Struktur Data

dengan C++, Java, dan Python



Irfan Subakti

Profitina
profitina.com

Struktur Data

dengan C++, Java, dan Python

Irfan Subakti



Hak cipta © 2026 Irfan Subakti. Hak cipta dilindungi undang-undang.

Diterbitkan oleh Profitina · profitina.com

Edisi pertama, September 2026.

Setiap program dalam buku ini sudah dikompilasi dan dijalankan sebelum keluarannya dicetak. Kode sumber lengkapnya ada di folder Code yang menyertai buku ini. Program-program tersebut disediakan untuk keperluan belajar-mengajar; Irfan Subakti dan penerbit tidak bertanggung jawab atas penggunaannya di luar keperluan itu.

Daftar Isi

Kata Pengantar ..... **lix**
Untuk siapa buku ini ditulis lix
Satu gagasan, tiga bahasa lix
Isi buku ini lix
Cara membacanya lx
Tentang penulis lxi
Ucapan terima kasih lxi

BAB 1 Fondasi: Penyegaran C++, Memori, dan Array **1**
1.1 Titik Berangkat: Perpustakaan Tanpa Rak 1
1.1.1 Pustaka Ceria 1
1.1.2 Apa itu struktur data 2
1.1.3 Mengapa pilihan itu penting 2
1.1.4 Struktur data di balik setiap halaman web 3
1.1.5 Enam keluarga, dan peta jalan buku ini 4
1.2 Tiga Bahasa, Satu Gagasan 5
1.2.1 Mengapa C++ lebih dahulu, dan mengapa juga Java dan Python 5
1.2.2 Bagaimana program menjadi sesuatu yang berjalan 6
1.2.3 Bagaimana kode disajikan 7
1.3 Bagaimana Memori Menyimpan Data 7
1.3.1 Bit dan byte 8
1.3.2 Alamat 8
1.3.3 Tipe memberi makna pada byte 8
1.3.4 Melihatnya sendiri 9
1.3.5 Stack dan heap 12
1.4 Penyegaran C++ bagi Pemula 14
1.4.1 Bentuk sebuah program C++ 14
1.4.2 Tipe dan variabel 14
1.4.3 Masukan dan keluaran 15
1.4.4 Mengambil keputusan dan mengulang 17
1.4.5 Fungsi 18
1.4.6 Struct: banyak nilai di bawah satu nama 19
1.4.7 Operator alamat, dan pandangan pertama pada pointer 23
1.5 Array Secara Mendalam 26
1.5.1 Satu deret slot bernomor 26
1.5.2 Mengapa indeks dimulai dari 0 27
1.5.3 Array berisi rekaman: katalog Pustaka Ceria 27
1.5.4 Tetap di dalam batas 33
1.5.5 Menelusuri array: penelusuran sehari-hari 35
1.5.6 Array dua dimensi dan tata letak row-major 39
1.5.7 Tabel bilangan 44
1.5.8 `std::array` dan `std::vector` 47
1.5.9 Array Java dan `ArrayList` 50
1.5.10 List Python dan modul `array` 51
1.5.11 Ketiga bahasa berdampingan 54
1.6 Cicipan Pertama Kompleksitas: Akses $O(1)$ lawan Penyisipan $O(n)$ 55
1.6.1 Menghitung langkah, bukan detik 55
1.6.2 Akses: biaya yang sama untuk setiap indeks 55
1.6.3 Penyisipan dan penghapusan: biaya menjaga deretan tetap rapat 56
1.6.4 Big-O, secara informal 61

1.6.5 Sekilas tentang biaya teramortisasi 62
1.7 Menyiapkan VS Code untuk C++, Java, dan Python di Windows, macOS, dan Linux 62
1.7.1 Tiga kata lebih dahulu 62
1.7.2 Visual Studio Code 63
1.7.3 Kompilator dan debugger C++ 63
1.7.4 Java: JDK 25 64
1.7.5 Python 3.14 65
1.7.6 Ekstensi VS Code 65
1.7.7 Memeriksa seluruh meja kerja 66
1.7.8 Program pertama Anda, dalam tiga bahasa 66
1.7.9 Menjalankan dari VS Code, mengumpankan masukan, dan debugging 68
1.8 Profitina dalam Praktik 69
1.9 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 70
1.10 Ringkasan Bab 71
1.11 Soal Latihan 72
1.12 Penyelesaian Soal Latihan 73
Daftar Pustaka 82

BAB 2 Pencarian 83
2.1 Dari Mana Kita Melanjutkan 83
2.2 Masalah Pencarian 84
2.2.1 Rekaman, kunci, dan kontrak ditemukan/tidak ditemukan 84
2.2.2 Cara kita mengukur biaya pencarian 84
2.2.3 Rak komputer Pustaka Ceria 85
2.3 Pencarian Sekuensial 85
2.3.1 Gagasannya 85
2.3.2 Dari penanda ke indeks 86
2.3.3 Programnya 86
2.3.4 Berapa biaya pencarian sekuensial? 96
2.4 Pencarian Sekuensial dengan Sentinel 96
2.4.1 Dua uji padahal satu cukup 96
2.4.2 Menghitung uji 97
2.4.3 Apa yang dibeli sentinel, dan apa yang tidak 101
2.5 Pencarian Biner 101
2.5.1 Memanfaatkan urutan 101
2.5.2 Jejak dengan tangan 102
2.5.3 Prosedur dan invariannya 102
2.5.4 Berapa banyak pemeriksaan? 103
2.5.5 Pencarian biner secara rekursif 104
2.5.6 Programnya 105
2.6 Jebakan Meleset Satu, *Lower Bound*, dan *Upper Bound* 111
2.6.1 Interval tertutup dan setengah terbuka 111
2.6.2 Empat jebakan 111
2.6.3 *Lower bound* dan *upper bound* 111
2.6.4 Titik tengah yang meluap 112
2.6.5 Programnya 113
2.7 Pencarian Interpolasi 120
2.7.1 Menebak di mana kunci seharusnya berada 120
2.7.2 Rak itu, dengan tangan 120
2.7.3 Programnya 121
2.7.4 Cemerlang atau buruk: bergantung pada datanya 125
2.8 Pencarian Lompat 126
2.8.1 Melompat maju per blok 126
2.8.2 Memilih ukuran blok 127
2.8.3 Programnya 127

2.9 Pencarian pada Larik Dua Dimensi 131
2.9.1 Letak larik dua dimensi di memori 131
2.9.2 Tiga cara mencari di sebuah kisi 132
2.9.3 Programnya 133
2.10 Mengukur pada Data Nyata 142
2.10.1 Sebuah katalog nyata 142
2.10.2 Percobaannya 142
2.10.3 Apa kata angka-angka itu 156
2.10.4 Bagaimana hitungan pemeriksaan tumbuh 157
2.11 Pencarian dengan Pustaka Standar 167
2.11.1 C++ 167
2.11.2 Java 169
2.11.3 Python 171
2.12 Memilih Metode Pencarian, dan Batas Pencarian dengan Perbandingan 173
2.12.1 Gambaran utuhnya 173
2.12.2 Mengapa tidak ada algoritma berbasis perbandingan yang dapat mengalahkan pencarian biner 173
2.13 Profitina dalam Praktik 174
2.13.1 Di mana isi bab ini muncul dalam produk nyata 174
2.13.2 Studi kasus yang dikerjakan: Profitina Laundry 175
2.14 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 182
2.15 Ringkasan Bab 183
2.15.1 Menatap ke depan 184
2.16 Soal Latihan 184
2.17 Penyelesaian Soal Latihan 185
Daftar Pustaka 194

BAB 3 Pengurutan 196
3.1 Titik Berangkat 196
3.2 Mengapa Mengurutkan, dan Istilah yang Kita Perlukan 197
3.2.1 Apa itu pengurutan 197
3.2.2 Mengapa sepadan dengan usahanya 197
3.2.3 Enam keluarga metode pengurutan 198
3.2.4 Satu gerakan yang diperlukan setiap pengurutan: menukar dua nilai 199
3.2.5 Menghitung kerja: perbandingan dan pertukaran 202
3.2.6 Tiga sifat: di tempat, stabil, adaptif 203
3.3 Bubble Sort 204
3.3.1 Gagasannya 204
3.3.2 Keluar dini 204
3.3.3 Jejak lengkap 204
3.3.4 Programnya 205
3.3.5 Tepatnya berapa banyak kerja 209
3.3.6 Bentuk-bentuk lain yang akan Anda jumpai 210
3.4 Exchange Sort 210
3.5 Selection Sort 214
3.5.1 Cari dulu, tukar sekali 214
3.5.2 Programnya 215
3.5.3 Pemindahan paling sedikit, tetapi perbandingan tak pernah berkurang 218
3.6 Insertion Sort 218
3.6.1 Mengurutkan kartu di tangan 218
3.6.2 Jebakan pada syarat gelung 219
3.6.3 Programnya 220
3.6.4 Mengapa insertion sort adalah yang terbaik di antara pengurutan sederhana 222
3.7 Shell Sort 223
3.7.1 Lompatan jauh lebih dahulu 223

3.7.2 Jejak dengan celah 3 dan 1 223
3.7.3 Barisan celah menentukan segalanya 224
3.8 Merge Sort 230
3.8.1 Bagi dan kuasai 230
3.8.2 Jejaknya 231
3.8.3 Programnya 232
3.8.4 Mengapa merge sort selalu $n\lg n$ 236
3.9 Quicksort 236
3.9.1 Partisi, lalu rekursi 236
3.9.2 Partisi Lomuto 237
3.9.3 Partisi Hoare 238
3.9.4 Programnya, dan pemilihan pivot 239
3.9.5 Kasus terbaik, terburuk, dan rata-rata 249
3.10 Heap Sort: Pandangan Pertama 250
3.10.1 Pohon yang tersembunyi di dalam larik 250
3.10.2 Jejaknya 251
3.11 Counting Sort dan Radix Sort 258
3.11.1 Batas yang tidak dapat dilintasi pengurutan berbasis perbandingan 258
3.11.2 Counting sort 259
3.11.3 Radix sort 260
3.12 Kestabilan, Di Tempat, dan Tabel Lengkapnya 264
3.12.1 Kestabilan yang dapat dilihat 264
3.12.2 Tabel lengkapnya 269
3.13 Mengukur Pengurutan 270
3.13.1 Program tolok ukur 270
3.13.2 Membaca angka-angkanya 276
3.13.3 Urutan masukan berpengaruh pada sebagian pengurutan 278
3.13.4 Tolok ukur yang sama dalam Java dan Python 279
3.14 Pengurutan Pustaka Masa Kini 290
3.14.1 C++: introsort dan merge sort 291
3.14.2 Java: quicksort dua-pivot dan TimSort 291
3.14.3 Python: Timsort dan Powersort 292
3.14.4 Memakainya 292
3.15 Profitina dalam Praktik 296
3.16 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 299
3.17 Ringkasan Bab 300
3.17.1 Menatap ke depan 302
3.18 Soal Latihan 302
3.19 Penyelesaian Soal Latihan 303
Daftar Pustaka 311

BAB 4 Evaluasi dan Pembahasan I: Array, Pencarian, dan Pengurutan, Diselesaikan Langkah demi Langkah 313
4.1 Dari Mana Kita Berangkat 313
4.2 Cara Menggarap Soal Struktur Data 314
4.2.1 Delapan langkah 315
4.2.2 Memperkirakan biaya sebelum menulis kode 316
4.2.3 Susunan setiap soal, dan tempat kodenya 317
4.3 Soal 4.1: Mengurutkan Kata Menurut Abjad 318
4.3.1 Soalnya 318
4.3.2 Memahaminya dengan tangan 318
4.3.3 Merancangnya di atas kertas 319
4.3.4 Penyelesaian C++ 320
4.3.5 Penyelesaian Java dan Python 322
4.3.6 Penyelesaian keliru yang menggoda 324

4.4 Soal 4.2: Karakter Pertama yang Berulang 326
4.4.1 Soalnya 326
4.4.2 Memahaminya dengan tangan 326
4.4.3 Merancangnya di atas kertas 326
4.4.4 Penyelesaian C++ 327
4.4.5 Penyelesaian Java dan Python 329
4.4.6 Penyelesaian keliru yang menggoda 330
4.5 Soal 4.3: Mencari Uang Menurut Nama 332
4.5.1 Soalnya 332
4.5.2 Memahaminya dengan tangan 332
4.5.3 Merancangnya di atas kertas 332
4.5.4 Penyelesaian C++ 333
4.5.5 Penyelesaian Java dan Python 335
4.5.6 Penyelesaian keliru yang menggoda 336
4.6 Soal 4.4: Mengurutkan Menurut Kepentingan 338
4.6.1 Soalnya 338
4.6.2 Memahaminya dengan tangan 338
4.6.3 Merancangnya di atas kertas 339
4.6.4 Penyelesaian C++ 339
4.6.5 Penyelesaian Java dan Python 341
4.6.6 Penyelesaian keliru yang menggoda 343
4.7 Soal 4.5: Alamat Larik dengan Tangan 344
4.7.1 Soalnya 344
4.7.2 Memahaminya dengan tangan 344
4.7.3 Merancangnya di atas kertas 345
4.7.4 Penyelesaian C++ 346
4.7.5 Penyelesaian Java dan Python 348
4.7.6 Penyelesaian keliru yang menggoda 349
4.8 Soal 4.6: Menghitung Kunci dalam Larik Terurut 349
4.8.1 Soalnya 349
4.8.2 Memahaminya dengan tangan 350
4.8.3 Merancangnya di atas kertas 350
4.8.4 Penyelesaian C++ 351
4.8.5 Penyelesaian Java dan Python 353
4.8.6 Penyelesaian keliru yang menggoda 354
4.9 Soal 4.7: Pencarian Interpolasi pada Kode Buku 356
4.9.1 Soalnya 356
4.9.2 Memahaminya dengan tangan 356
4.9.3 Merancangnya di atas kertas 356
4.9.4 Penyelesaian C++ 357
4.9.5 Penyelesaian Java dan Python 359
4.9.6 Penyelesaian keliru yang menggoda 363
4.10 Soal 4.8: Menelusuri Pengurutan Dasar 364
4.10.1 Soalnya 364
4.10.2 Memahaminya dengan tangan 364
4.10.3 Merancangnya di atas kertas 364
4.10.4 Penyelesaian C++ 365
4.10.5 Penyelesaian Java dan Python 368
4.10.6 Penyelesaian keliru yang menggoda 372
4.11 Soal 4.9: Menggabungkan Dua Daftar Terurut 373
4.11.1 Soalnya 373
4.11.2 Memahaminya dengan tangan 373
4.11.3 Merancangnya di atas kertas 373
4.11.4 Penyelesaian C++ 374
4.11.5 Penyelesaian Java dan Python 375

4.11.6 Penyelesaian keliru yang menggoda 377
4.12 Soal 4.10: Menghapus Nilai di Tempat 378
4.12.1 Soalnya 378
4.12.2 Memahaminya dengan tangan 379
4.12.3 Merancangnya di atas kertas 379
4.12.4 Penyelesaian C++ 380
4.12.5 Penyelesaian Java dan Python 382
4.12.6 Penyelesaian keliru yang menggoda 383
4.13 Soal 4.11: Dua Harga yang Cocok dengan Voucher 384
4.13.1 Soalnya 384
4.13.2 Memahaminya dengan tangan 384
4.13.3 Merancangnya di atas kertas 385
4.13.4 Penyelesaian C++ 386
4.13.5 Penyelesaian Java dan Python 388
4.13.6 Penyelesaian keliru yang menggoda 390
4.14 Soal-Soal Berdampingan 391
4.15 Profitina dalam Praktik 391
4.16 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 392
4.17 Ringkasan Bab 393
4.18 Soal Latihan 394
4.19 Penyelesaian Soal Latihan 395
Daftar Pustaka 396

BAB 5 Stack dan Queue 398

5.1 Posisi Kita Sejauh Ini 398
5.1.1 Dari akses bebas ke akses berdisiplin 398
5.1.2 Tipe data abstrak 399
5.2 Kisah Pembuka: Menara Hanoi 400
5.2.1 Tiga cakram, dengan tangan 400
5.2.2 Berapa banyak langkah? 401
5.2.3 Menyelesaikan Hanoi dengan tiga stack tanpa rekursi 402
5.3 Stack sebagai Tipe Data Abstrak 405
5.3.1 Operasi-operasinya 405
5.3.2 Operasi dalam pseudokode 406
5.4 Stack Berbasis Larik 407
5.4.1 Mengamati larik 407
5.4.2 Stack dalam C++ 408
5.4.3 Stack yang sama dalam Java dan Python 411
5.4.4 Setiap operasi bernilai $O(1)$ 415
5.5 Stack yang Tumbuh dan Stack Bertaut 416
5.5.1 Menggandakan larik 416
5.5.2 Sekilas: stack bertaut 417
5.6 Aplikasi: Mencocokkan Tanda Kurung 420
5.6.1 Algoritmanya 420
5.6.2 Programnya 421
5.7 Aplikasi: Undo dan Redo 423
5.8 Aplikasi: Infiks ke Postfiks dengan Algoritma Shunting-Yard 426
5.8.1 Tiga cara menulis jumlah yang sama 426
5.8.2 Presedensi dan asosiativitas 426
5.8.3 Algoritmanya 427
5.8.4 Menelusuri `3+2*5` 427
5.8.5 Menelusuri `a+b*c-d` 428
5.9 Aplikasi: Mengevaluasi Ekspresi Postfiks 428
5.9.1 Kalkulator lengkap 429
5.10 Aplikasi: Call Stack Fungsi 435

5.10.1 Yang terjadi ketika fungsi dipanggil 435
5.10.2 Menyaksikannya terjadi 435
5.11 Queue sebagai Tipe Data Abstrak 437
5.11.1 Operasi-operasinya 438
5.12 Queue dalam Larik Linear, dan Masalah Pergeseran 439
5.12.1 Percobaan pertama: geser semuanya pada setiap *dequeue* 439
5.12.2 Percobaan kedua: biarkan indeksnya bergerak 440
5.13 Queue Sirkular 443
5.13.1 Aritmetika modular 443
5.13.2 Rancangan pertama: front ditambah pencacah 443
5.13.3 Queue sirkular dalam C++, Java, dan Python 445
5.13.4 Rancangan kedua: sisakan satu slot kosong 450
5.14 Deque 453
5.15 Priority Queue: Sekilas 457
5.16 Stack dan Queue dalam Pustaka Standar 457
5.16.1 C++: *container adaptor* 457
5.16.2 Java: `ArrayDeque` untuk keduanya 459
5.16.3 Python: `list`, `collections.deque`, `queue`, dan `heapq` 460
5.17 Memilih di Antara Mereka: Ringkasan Biaya 462
5.18 Profitina dalam Praktik 463
5.19 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 464
5.20 Ringkasan Bab 466
5.21 Soal Latihan 466
5.22 Penyelesaian Soal Latihan 468
Daftar Pustaka 472

BAB 6 Pointer, Referensi, dan Fungsi 474

6.1 Dari Mana Kita Melanjutkan 474
6.2 Memori, Alamat, dan Apa Itu Pointer 475
6.2.1 Memori sebagai jalan dengan rumah-rumah bernomor 475
6.2.2 Operator alamat `&` dan pointer pertama 476
6.2.3 Operator dereferensi `*` 477
6.2.4 Tipe pointer itu penting 478
6.2.5 Menggambar memori: konvensi kotak dan panah 479
6.3 Operasi pada Pointer 479
6.3.1 Penugasan: menyalin alamat versus menyalin nilai 479
6.3.2 Perbandingan 481
6.3.3 Aritmetika: bergerak di dalam larik 481
6.3.4 Yang boleh dan yang tidak 484
6.4 Pointer dan Larik 485
6.4.1 Nama larik menjadi pointer 485
6.4.2 Menyusuri larik dengan pointer 485
6.4.3 String adalah larik `char`, dan terminatornya penting 486
6.4.4 Larik sebagai parameter fungsi 488
6.4.5 Larik dua dimensi 490
6.5 Pointer ke Struktur dan Operator Panah 490
6.6 Memori Dinamis: `new` dan `delete` 493
6.6.1 Dua tempat tinggal: *stack* dan *heap* 493
6.6.2 `new`, `delete`, `new[]`, dan `delete[]` 494
6.6.3 Larik tumbuh dan aturan penggandaan 495
6.6.4 Ketika `new` gagal 497
6.7 Ketika Pointer Salah Jalan 497
6.7.1 Lima bug, satu akar masalah 497
6.7.2 Bug-bug itu, dengan sengaja 498
6.7.3 AddressSanitizer 501

6.7.4 Valgrind 502
6.7.5 Alat yang mana, di komputer yang mana 503
6.8 `nullptr`: Pointer ke Ketiadaan 504
6.9 Referensi 506
6.9.1 Nama kedua untuk objek yang sudah ada 506
6.9.2 Referensi versus pointer 507
6.10 Fungsi dan Tiga Cara Mengirim Argumen 508
6.10.1 Fungsi: tinjauan singkat 508
6.10.2 Apa yang terjadi selama pemanggilan 510
6.10.3 Kisah *swap* 510
6.10.4 Biaya pengiriman berdasarkan nilai 513
6.10.5 Memilih cara 513
6.11 Ketepatan `const` 514
6.12 Mengembalikan Pointer dan Referensi dengan Aman 516
6.13 *Smart Pointer*: Kepemilikan yang Tertulis dalam Tipe 519
6.13.1 Masalah dengan `new` dan `delete` mentah 519
6.13.2 `std::unique_ptr`: tepat satu pemilik 519
6.13.3 `std::shared_ptr` dan `std::weak_ptr`: pemilik yang dihitung 521
6.14 Referensi Java dan Nama Python 525
6.14.1 Satu aturan untuk masing-masing 525
6.14.2 Mutasi versus pengikatan ulang, dalam Java 526
6.14.3 Mutasi versus pengikatan ulang, dalam Python 528
6.14.4 Kisah *swap* dalam Java dan Python 530
6.14.5 Siapa yang membebaskan memori? 532
6.14.6 Tiga bahasa berdampingan 533
6.15 Biaya Setiap Operasi 534
6.16 Profitina dalam Praktik 534
6.17 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 539
6.18 Ringkasan Bab 540
6.19 Soal Latihan 541
6.20 Penyelesaian Soal Latihan 542
Daftar Pustaka 548

BAB 7 Single Linked List 549
7.1 Dari Mana Kita Melanjutkan 549
7.2 Rantai, Bukan Barisan: Sejarah dan Gagasan 550
7.2.1 Perburuan harta karun 550
7.2.2 Asal-usul gagasannya 550
7.2.3 Kosakatanya 551
7.3 Larik versus Linked List 552
7.3.1 Dua cara menyimpan lima bilangan 552
7.3.2 Perbandingannya, ciri demi ciri 553
7.3.3 Memori yang dimakan setiap elemen 554
7.4 Simpul dan Pointer dalam C++ 554
7.4.1 Mendeklarasikan simpul 554
7.4.2 Membuat simpul dengan `new` 555
7.4.3 Tiga simpul dengan tangan 555
7.4.4 Cara lain: `malloc` dan `free` 557
7.4.5 Cara membaca diagram pointer dalam bab ini 558
7.5 Single Linked List dengan Pointer Head 558
7.5.1 `init` dan `isEmpty` 559
7.5.2 `frontInsertion`: simpul pertama yang baru 559
7.5.3 `backInsertion`: simpul terakhir yang baru 560
7.5.4 `show`: menyusuri list 562
7.5.5 `frontDeletion`: membuang simpul pertama 563

7.5.6 `backDeletion`: membuang simpul terakhir 564
7.5.7 `clear`: melepaskan setiap simpul 565
7.5.8 Program lengkapnya 565
7.5.9 List yang sama dalam Java dan Python 569
7.6 Operasi Lain: Pencarian, Panjang, Penyisipan di Tengah, Penghapusan Berdasarkan Kunci, Pembalikan 575
7.6.1 `search` 575
7.6.2 `length`, secara iteratif dan rekursif 575
7.6.3 `insertAfter`: penyisipan di tengah 576
7.6.4 `insertSorted`: menjaga list tetap terurut 577
7.6.5 `deleteKey`: penghapusan berdasarkan nilai 578
7.6.6 Membalik list secara iteratif 579
7.6.7 Membalik list secara rekursif 581
7.6.8 `destroy` 582
7.6.9 Program lengkapnya 582
7.7 Simpul Kepala Dummy: Lebih Sedikit Kasus Khusus 586
7.7.1 Dari mana kasus-kasus khusus berasal 586
7.7.2 Operasi tanpa kasus khusus 587
7.7.3 Cara ketiga: pointer ke tautan 590
7.7.4 Varian mana yang dipakai 592
7.8 Pointer Head dan Tail 592
7.8.1 Gagasannya 592
7.8.2 Mengapa `tail` tidak dapat mempercepat penghapusan belakang 593
7.8.3 Programnya 594
7.9 Single Linked List Melingkar 597
7.9.1 Menutup rantai 597
7.9.2 Penelusuran memerlukan tanda berhenti yang baru 598
7.9.3 Layanan bergiliran di meja baca 598
7.10 Stack dan Queue yang Dibangun di Atas Linked List 601
7.10.1 Kecocokannya 601
7.10.2 Troli pengembalian dan antrean pemesanan Pustaka Ceria 602
7.10.3 Berantai atau berbasis larik? 606
7.11 Keamanan Memori: Membuktikan Tidak Ada Kebocoran 606
7.11.1 Mengapa “hasilnya sudah benar” belum cukup 606
7.11.2 Valgrind 609
7.11.3 AddressSanitizer 612
7.11.4 Daftar periksa untuk setiap fungsi linked list 613
7.12 Linked List dalam Pustaka Baku 614
7.12.1 C++: `std::forward_list` dan `std::list` 614
7.12.2 Java: `java.util.LinkedList` 616
7.12.3 Python: mengapa tidak ada linked list bawaan, dan `collections.deque` 617
7.12.4 Mengukur: larik melawan list pada mesin sungguhan 619
7.13 Profitina dalam Praktik 622
7.14 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 623
7.15 Ringkasan Bab 624
7.16 Soal Latihan 625
7.17 Penyelesaian Soal Latihan 627
Daftar Pustaka 634

BAB 8 Evaluasi dan Pembahasan II: Stack, Pointer, dan Linked List dalam Penerapan 635
8.1 Titik Tolak Kita 635
8.2 Cara Menghadapi Soal Pointer 637
8.3 Soal 8.1 — Melengkapi Stack Larik 639
8.3.1 Soalnya 639
8.3.2 Dengan kata-kata kita sendiri 640

8.3.3 Dengan tangan, selangkah demi selangkah 641
8.3.4 Penyelesaian C++ 641
8.3.5 Penyelesaian Java dan Python 644
8.3.6 Penyelesaian keliru yang menggoda 648
8.3.7 Uji diri 648
8.4 Soal 8.2 — Menelusuri List Dua Arah Menurut Petunjuk 648
8.4.1 Soalnya 648
8.4.2 Secukupnya tentang simpul bertaut dua 649
8.4.3 Penelusuran, dengan tangan 650
8.4.4 Apa yang dapat salah pada masukan 651
8.4.5 Penyelesaian C++ 651
8.4.6 Penyelesaian Java dan Python 654
8.4.7 Penyelesaian keliru yang menggoda 658
8.4.8 Uji diri 658
8.5 Soal 8.3 — Genap Naik, Ganjil Turun, Hanya dengan Linked List 659
8.5.1 Soalnya 659
8.5.2 Dengan kata-kata kita sendiri 659
8.5.3 Dengan tangan, selangkah demi selangkah 659
8.5.4 Penyelesaian C++ 661
8.5.5 Penyelesaian Java dan Python 663
8.5.6 Penyelesaian keliru yang menggoda 666
8.5.7 Uji diri 667
8.6 Soal 8.4 — Membalik Urutan Kata Sebuah Kalimat 667
8.6.1 Soalnya 667
8.6.2 Dengan kata-kata kita sendiri 667
8.6.3 Dengan tangan, selangkah demi selangkah 668
8.6.4 Penyelesaian C++ 668
8.6.5 Penyelesaian Java dan Python 669
8.6.6 Penyelesaian keliru yang menggoda 671
8.6.7 Uji diri 672
8.7 Soal 8.5 — Membuang Nilai Ganda dari Sebuah List 672
8.7.1 Soalnya 672
8.7.2 Dengan kata-kata kita sendiri 672
8.7.3 Dengan tangan, selangkah demi selangkah 673
8.7.4 Penyelesaian C++ 674
8.7.5 Penyelesaian Java dan Python 677
8.7.6 Penyelesaian keliru yang menggoda 681
8.7.7 Uji diri 682
8.8 Soal Baru yang Dikerjakan: Stack dan Queue 682
8.8.1 Soal 8.6 — Mencocokkan tanda kurung 682
8.8.2 Soal 8.7 — Queue sirkular untuk permintaan peminjaman 688
8.8.3 Soal 8.8 — Dari infiks ke postfiks, dan nilai sebuah ekspresi 695
8.9 Soal Baru yang Dikerjakan: Pointer dan Fungsi 702
8.9.1 Soal 8.9 — By value, by reference, by pointer 702
8.10 Soal Baru yang Dikerjakan: Singly Linked List 710
8.10.1 Soal 8.10 — Membalik list di tempat 710
8.10.2 Soal 8.11 — Menggabungkan dua list terurut 716
8.10.3 Soal 8.12 — Queue dari simpul bertaut 723
8.11 Perbandingan Soal-Soal dalam Bab Ini 731
8.12 Profitina dalam Praktik 732
8.13 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 733
8.14 Ringkasan Bab 734
8.15 Soal Latihan 734
8.16 Penyelesaian Soal Latihan 735
Daftar Pustaka 747

BAB 9 Double Linked List dan Linked List Sirkular 749
9.1 Dari Mana Kita Melanjutkan 749
9.2 Dua Panah di Setiap Simpul 750
9.2.1 Kereta yang dapat disusuri ke dua arah 750
9.2.2 Definisi dan kosakatanya 751
9.2.3 Mendeklarasikan simpul, dan membuat simpul baru 752
9.2.4 Harga pointer kedua 754
9.2.5 Cara membaca diagram pointer dalam bab ini 754
9.3 Double Linked List dengan Pointer Head 755
9.3.1 `init` dan `isEmpty` 755
9.3.2 `frontInsertion`: simpul pertama yang baru 755
9.3.3 `backInsertion`: simpul terakhir yang baru 756
9.3.4 Menyusuri ke dua arah 757
9.3.5 `frontDeletion`: membuang simpul pertama 758
9.3.6 `backDeletion`: membuang simpul terakhir 759
9.3.7 `clear` 760
9.3.8 Program lengkapnya 760
9.3.9 List yang sama dalam Java dan Python 765
9.4 Head dan Tail: Setiap Ujung dalam O(1) 771
9.4.1 Menambahkan `tail`, dan akhirnya penghapusan di belakang dalam O(1) 771
9.4.2 `insertAfter` dan `insertBefore`: menyisipkan di samping simpul yang dipegang 773
9.4.3 `insertSorted`: menjaga list tetap terurut 774
9.4.4 `search`, `deleteNode`, dan `deleteKey` 775
9.4.5 Memeriksa tautan 776
9.4.6 Program lengkapnya 776
9.5 Double Linked List dengan Simpul Kepala 781
9.5.1 Dari mana sisa kasus khusus berasal 781
9.5.2 Programnya 782
9.6 Double Linked List Sirkular 785
9.6.1 Menutup kedua rantai 785
9.6.2 Penyusuran butuh tanda berhenti yang baru 786
9.6.3 Korsel di lobi 786
9.7 Sentinel: Melenyapkan Setiap Kasus Khusus 789
9.7.1 Gagasannya 789
9.7.2 Mengapa kasus khusus lenyap 790
9.7.3 Mencari dengan kunci di dalam sentinel 792
9.7.4 Programnya 792
9.7.5 Kasus khusus, dihitung 795
9.8 Deque untuk Meja Pengembalian 795
9.8.1 Apa itu deque 795
9.8.2 Meja pengembalian Pustaka Ceria 796
9.9 Daftar “Baru Dipinjam”: Pola LRU 799
9.9.1 Masalahnya 799
9.9.2 Dua struktur yang bekerja bersama 799
9.9.3 Programnya 801
9.10 Perbandingan Ongkos: Single Linked, Double Linked, Larik 803
9.10.1 Tabelnya 803
9.10.2 Mengukur apa yang dibeli pointer kedua 804
9.11 Keamanan Memori untuk Tautan Dua Arah 808
9.11.1 Apa yang berubah dengan dua tautan 808
9.11.2 Apa kata alat-alatnya 812
9.11.3 Daftar periksa untuk tautan dua arah 814
9.12 Double Linked List dalam Pustaka Standar 814
9.12.1 C++: `std::list` 814
9.12.2 Java: `LinkedList` dan `LinkedHashMap` 816

9.12.3 Python: `collections.deque`, `OrderedDict`, dan `lru_cache` 818
9.13 Profitina dalam Praktik 820
9.14 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 822
9.15 Ringkasan Bab 823
9.16 Soal Latihan 824
9.17 Penyelesaian Soal Latihan 826
Daftar Pustaka 833

BAB 10 Rekursi 835
10.1 Dari Mana Kita Melanjutkan 835
10.2 Berpikir Secara Rekursif 836
10.2.1 Pertanyaan yang diteruskan sepanjang antrean 836
10.2.2 Definisi rekursif 837
10.2.3 Membuktikan definisi rekursif: induksi 838
10.2.4 Dari definisi ke program 838
10.2.5 Tiga aturan, dan sepatah kata jujur tentang ongkos 840
10.3 Call Stack, Frame demi Frame 841
10.3.1 Frame, sekali lagi 841
10.3.2 Faktorial, frame demi frame 841
10.3.3 Jejak itu sebagai program 843
10.3.4 Bila base case hilang 847
10.4 Faktorial: Iteratif dan Rekursif 848
10.4.1 Dua algoritma untuk satu fungsi 848
10.4.2 Ongkos masing-masing 848
10.5 Fibonacci dan Kelinci-Kelincinya 849
10.5.1 Masalah kelinci 849
10.5.2 Rekursi naif, dan mengapa ia meledak 850
10.5.3 Memoisasi: ingat apa yang sudah dihitung 852
10.5.4 Bottom-up: tanpa rekursi sama sekali 852
10.5.5 Rekursi ekor: gelung yang ditulis sebagai rekursi 853
10.5.6 Fast doubling: $O(\log n)$ 853
10.5.7 Bentuk tertutup Binet 853
10.5.8 Keenam cara dalam satu program 854
10.5.9 Diukur: panggilan dan waktu 859
10.6 Faktor Persekutuan Terbesar Euklides 861
10.6.1 Masalah dan gagasannya 861
10.6.2 Dengan tangan, dan sebagai gambar 862
10.6.3 Programnya 862
10.6.4 Berapa banyak pembagian? Teorema Lamé 865
10.7 Pemangkatan Cepat 865
10.7.1 Mengurangi satu: n perkalian 865
10.7.2 Membagi dua: sekitar $2\lg n$ perkalian 866
10.7.3 Pemangkatan modular 867
10.8 Menara Hanoi, Diselesaikan Secara Rekursif 869
10.8.1 Gagasan rekursifnya 869
10.8.2 Programnya 870
10.8.3 Urutan panggilan: sebuah pohon rekursi 876
10.8.4 Mengapa tepat $2n - 1$ langkah, dan mengapa tidak kurang 876
10.9 Rekursi pada Larik dan Linked List 877
10.9.1 Dua cara membuat larik lebih kecil 877
10.9.2 Linked list adalah struktur rekursif 877
10.9.3 Ongkos, dan sebuah peringatan 882
10.10 Pohon Rekursi 882
10.10.1 Menggambar panggilan-panggilannya 882
10.10.2 Merge sort dan Hanoi, aras demi aras 883

10.10.3 Tabel rekurens 883
10.11 Rekursi Ekor dan Apa yang Sesungguhnya Dilakukan Compiler 884
10.11.1 Tail call 884
10.11.2 Apa yang dilakukan GCC: sebuah eksperimen 885
10.11.3 Mengapa Anda tidak boleh bergantung padanya di C++ 888
10.11.4 JVM dan CPython tidak menghapus tail call 888
10.12 Seberapa Dalam Rekursi Boleh Berjalan? Batas di C++, Java, dan Python 890
10.12.1 C++: ukuran stack 890
10.12.2 Java: `-Xss`, dan thread dengan stack sendiri 892
10.12.3 Python: batas rekursi 893
10.12.4 Batas-batas itu sekilas 895
10.13 Dari Rekursi ke Iterasi dengan Stack Eksplisit 896
10.13.1 Kapan harus diubah 896
10.13.2 Metode umumnya 896
10.13.3 Hanoi dengan stack berisi frame 897
10.13.4 List panjang yang ditelusuri mundur 897
10.14 Profitina dalam Praktik 900
10.15 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 904
10.16 Ringkasan Bab 905
10.17 Soal Latihan 906
10.18 Penyelesaian Soal Latihan 907
Daftar Pustaka 913

BAB 11 Tree dan Binary Search Tree 916
11.1 Dari Mana Kita Melanjutkan 916
11.2 Apa Itu Pohon 917
11.2.1 Empat pohon dari kehidupan sehari-hari 917
11.2.2 Definisinya 918
11.2.3 Kosakatanya 918
11.2.4 Contoh kerja pada pohon kecil 920
11.2.5 Pohon statis dan pohon dinamis 920
11.3 Menggambar dan Menuliskan Pohon 921
11.4 Merepresentasikan Pohon dalam Memori 921
11.4.1 Representasi statis: larik 922
11.4.2 Representasi dinamis: simpul bertaut 924
11.4.3 Pohon dengan banyak anak sembarang 925
11.5 Pohon Biner 926
11.5.1 Definisi 926
11.5.2 Full, perfect, complete, dan condong 926
11.5.3 Berapa banyak simpul muat, dan seberapa rendah pohon dapat dibuat 928
11.6 Menelusuri Pohon Biner 928
11.6.1 Tiga urutan telusur-dalam 929
11.6.2 Berjalan mengelilingi pohon 930
11.6.3 Urutan tingkat dengan queue 931
11.6.4 Penelusuran tanpa rekursi 932
11.6.5 Ongkos penelusuran 937
11.6.6 Pohon ekspresi 937
11.7 Binary Search Tree 941
11.7.1 Sifat BST 941
11.7.2 Penyisipan: `insert(65, 5, 70, 3, 10)` 942
11.7.3 Pencarian 944
11.7.4 Minimum dan maksimum 945
11.7.5 Menghitung simpul, daun, dan tinggi 945
11.7.6 Mencetak pohon secara menyamping 946
11.7.7 Program lengkapnya 946

11.7.8 Pohon yang sama dalam Java dan Python 952
11.7.9 Successor dan predecessor 960
11.7.10 Penghapusan: tiga kasus 962
11.7.11 Dua varian: successor atau predecessor 964
11.7.12 Program penghapusan 965
11.7.13 Ongkos 971
11.8 Pohon Seimbang dan Pohon Merosot 972
11.8.1 Urutan penyisipan menentukan bentuk 972
11.8.2 Tinggi yang terukur 972
11.8.3 Arti "seimbang" 976
11.9 Sekilas tentang Pohon Penyeimbang-Diri 976
11.9.1 Rotasi 976
11.9.2 Pohon AVL 977
11.9.3 Pohon red-black 981
11.9.4 Pohon seimbang lainnya 982
11.10 Pohon Umum: Konversi dan Rekonstruksi 983
11.10.1 Anak pertama, saudara berikutnya 983
11.10.2 Membangun kembali pohon dari preorder dan derajat 984
11.10.3 Programnya 985
11.11 Pohon Pencarian dalam Pustaka Baku 989
11.11.1 C++: `std::set` dan `std::map` 990
11.11.2 Java: `TreeMap` dan `TreeSet` 992
11.11.3 Python: tanpa pohon pencarian bawaan 994
11.12 Profitina dalam Praktik 996
11.13 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 1003
11.14 Ringkasan Bab 1004
11.15 Soal Latihan 1005
11.16 Penyelesaian Soal Latihan 1006
Daftar Pustaka 1012

BAB 12 Evaluasi dan Pembahasan III: Rekursi Bekerja pada List dan Pohon 1014
12.1 Titik Tolak Kita 1014
12.2 Cara Menghadapi Soal Rekursif pada Struktur Bertaut 1016
12.3 Pohon yang Kita Kerjakan 1018
12.3.1 Titik awalnya 1018
12.3.2 *Toolkit*, dalam tiga bahasa 1020
12.3.3 Mengamati pohonnya 1026
12.4 Soal 12.1 — Penelusuran *Depth First*, Cabang Kiri Lebih Dulu 1028
12.4.1 Soalnya 1028
12.4.2 Dengan kata-kata kita sendiri 1029
12.4.3 Dengan tangan, selangkah demi selangkah 1029
12.4.4 Penyelesaian C++ 1030
12.4.5 Penyelesaian Java dan Python 1031
12.4.6 Penyelesaian keliru yang menggoda 1032
12.4.7 Uji diri 1033
12.5 Soal 12.2 — Mencetak Satu Tingkat 1033
12.5.1 Soalnya 1033
12.5.2 Dengan kata-kata kita sendiri 1033
12.5.3 Dengan tangan, selangkah demi selangkah 1034
12.5.4 Penyelesaian C++ 1034
12.5.5 Penyelesaian Java dan Python 1035
12.5.6 Penyelesaian keliru yang menggoda 1036
12.5.7 Uji diri 1037
12.6 Soal 12.3 — Jumlah Semua Simpul 1037
12.6.1 Soalnya 1037

12.6.2 Dengan kata-kata kita sendiri 1037
12.6.3 Dengan tangan, selangkah demi selangkah 1037
12.6.4 Penyelesaian C++ 1038
12.6.5 Penyelesaian Java dan Python 1039
12.6.6 Penyelesaian keliru yang menggoda 1040
12.6.7 Uji diri 1041
12.7 Soal 12.4 — Melipattigakan Setiap Simpul 1041
12.7.1 Soalnya 1041
12.7.2 Dengan kata-kata kita sendiri 1041
12.7.3 Penyelesaian C++ 1042
12.7.4 Penyelesaian Java dan Python 1043
12.7.5 Penyelesaian keliru yang menggoda 1044
12.7.6 Uji diri 1044
12.8 Soal 12.5 — Bayangan Cermin 1044
12.8.1 Soalnya 1044
12.8.2 Dengan kata-kata kita sendiri 1044
12.8.3 Dengan tangan, selangkah demi selangkah 1045
12.8.4 Penyelesaian C++ 1045
12.8.5 Penyelesaian Java dan Python 1046
12.8.6 Penyelesaian keliru yang menggoda 1048
12.8.7 Uji diri 1048
12.9 Soal 12.6 — Apakah Pohonnya Seimbang Tingginya? 1049
12.9.1 Soalnya 1049
12.9.2 Dengan kata-kata kita sendiri 1049
12.9.3 Dengan tangan, selangkah demi selangkah 1049
12.9.4 Penyelesaian C++ 1050
12.9.5 Penyelesaian Java dan Python 1052
12.9.6 Penyelesaian keliru yang menggoda 1054
12.9.7 Uji diri 1055
12.10 Soal 12.7 — Menghapus Sebuah Nilai 1055
12.10.1 Soalnya 1055
12.10.2 Dengan kata-kata kita sendiri 1055
12.10.3 Dengan tangan, selangkah demi selangkah 1056
12.10.4 Penyelesaian C++ 1057
12.10.5 Penyelesaian Java dan Python 1059
12.10.6 Penyelesaian keliru yang menggoda 1062
12.10.7 Uji diri 1062
12.11 Soal 12.8 — Mencari Induk 1062
12.11.1 Soalnya 1062
12.11.2 Dengan kata-kata kita sendiri 1063
12.11.3 Dengan tangan, selangkah demi selangkah 1063
12.11.4 Penyelesaian C++ 1064
12.11.5 Penyelesaian Java dan Python 1065
12.11.6 Penyelesaian keliru yang menggoda 1066
12.11.7 Uji diri 1067
12.12 Soal 12.9 — Melaporkan Anak-Anak 1067
12.12.1 Soalnya 1067
12.12.2 Dengan kata-kata kita sendiri 1067
12.12.3 Penyelesaian C++ 1067
12.12.4 Penyelesaian Java dan Python 1068
12.12.5 Penyelesaian keliru yang menggoda 1070
12.12.6 Uji diri 1070
12.13 Soal 12.10 — Dari Pohon Menjadi *Double Linked List*, di Tempat 1071
12.13.1 Soalnya 1071
12.13.2 Dengan kata-kata kita sendiri 1071

12.13.3 Gagasannya, dengan tangan 1072
12.13.4 Penyelesaian C++ 1073
12.13.5 Penyelesaian Java dan Python 1075
12.13.6 Penyelesaian keliru yang menggoda 1078
12.13.7 Uji diri 1079
12.14 Soal Baru yang Dikerjakan: *Double Linked List* dan List Sirkular 1079
12.14.1 Soal 12.11 — Membalik *double linked list* di tempat 1079
12.14.2 Soal 12.12 — Lingkaran Josephus 1087
12.15 Soal Baru yang Dikerjakan: Rekursi 1092
12.15.1 Soal 12.13 — Memilih buku 1092
12.16 Soal Baru yang Dikerjakan: Pohon 1097
12.16.1 Soal 12.14 — Benarkah ini *binary search tree*? 1097
12.16.2 Soal 12.15 — Leluhur bersama terendah 1101
12.16.3 Soal 12.16 — Mencetak pohon tingkat demi tingkat 1104
12.17 Perbandingan Soal-Soal dalam Bab Ini 1108
12.18 Profitina dalam Praktik 1109
12.19 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 1109
12.20 Ringkasan Bab 1110
12.21 Soal Latihan 1111
12.22 Penyelesaian Soal Latihan 1112
Daftar Pustaka 1119

BAB 13 Graf 1120
13.1 Dari Mana Kita Melanjutkan 1120
13.2 Kisah Pembuka: Tujuh Jembatan dan Sebuah Peta Kampus 1121
13.2.1 Jembatan-jembatan Königsberg 1121
13.2.2 Dari jembatan ke perpustakaan cabang 1122
13.3 Kosakata Graf 1123
13.3.1 Simpul dan sisi 1123
13.3.2 Berarah dan tak berarah 1124
13.3.3 Graf berbobot 1124
13.3.4 Derajat dan lema jabat tangan 1124
13.3.5 Lintasan dan siklus 1125
13.3.6 Keterhubungan dan komponen terhubung 1125
13.3.7 Graf asiklik berarah 1126
13.3.8 Berapa banyak sisi yang dapat dimiliki graf? Jarang dan padat 1126
13.4 Contoh Berjalan: Peta Kampus Pustaka Ceria 1127
13.5 Merepresentasikan Graf di Memori 1128
13.5.1 Matriks ketetanggaan 1128
13.5.2 List ketetanggaan 1129
13.5.3 List sisi 1129
13.5.4 Ongkos berdampingan 1129
13.5.5 Programnya dalam C++ 1130
13.5.6 Program yang sama dalam Java dan Python 1133
13.6 Penelusuran Melebar (Breadth-First Search) 1137
13.6.1 Gagasannya: riak di kolam 1137
13.6.2 Prosedurnya 1137
13.6.3 Menelusuri BFS dari perpustakaan pusat 1138
13.6.4 Mengapa BFS menemukan rute dengan langkah paling sedikit 1140
13.6.5 Waktu jalannya 1141
13.6.6 Programnya dalam C++ 1142
13.6.7 Program yang sama dalam Java dan Python 1145
13.7 Penelusuran Mendalam (Depth-First Search) 1149
13.7.1 Gagasannya: menjelajahi labirin dengan segulung benang 1149
13.7.2 DFS rekursif dengan waktu penemuan dan waktu selesai 1150

13.7.3 Menelusuri DFS pada peta kampus 1151
13.7.4 Struktur tanda kurung 1152
13.7.5 DFS tanpa rekursi 1153
13.7.6 Menemukan siklus dalam graf tak berarah 1154
13.7.7 Programnya 1154
13.8 Keterhubungan dan Komponen Terhubung 1163
13.8.1 Satu penelusuran untuk setiap komponen 1163
13.8.2 Jalan shuttle 1163
13.8.3 Programnya 1164
13.8.4 Graf berarah: catatan tentang keterhubungan kuat 1170
13.9 Graf Asiklik Berarah dan Pengurutan Topologis 1170
13.9.1 Alur kerja buku baru 1170
13.9.2 Algoritma Kahn 1171
13.9.3 Pengurutan topologis dengan DFS 1173
13.9.4 Mendeteksi siklus dalam graf berarah 1173
13.9.5 Programnya 1174
13.10 Lintasan Terpendek Berbobot: Dijkstra dengan Perlahan 1182
13.10.1 Mengapa BFS tidak cukup 1182
13.10.2 Relaksasi 1182
13.10.3 Algoritma Dijkstra 1182
13.10.4 Menelusuri Dijkstra dari perpustakaan pusat 1183
13.10.5 Mengapa algoritma ini benar, dan kapan tidak 1184
13.10.6 Waktu jalan, dan sekilas ke Bab 14 1185
13.10.7 Programnya 1185
13.11 Graf pada Skala Dunia Nyata 1191
13.11.1 Jaringan jalan 1191
13.11.2 Graf sosial 1192
13.11.3 Apa yang berubah pada skala besar, dan apa yang tidak 1193
13.12 Memilih Algoritma: Ongkos Sekilas 1193
13.13 Profitina dalam Praktik 1194
13.14 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 1198
13.15 Ringkasan Bab 1199
13.16 Soal Latihan 1200
13.17 Penyelesaian Soal Latihan 1202
Daftar Pustaka 1213

BAB 14 Heap dan Priority Queue 1214
14.1 Dari Mana Kita Melanjutkan 1214
14.2 Melayani yang Paling Mendesak Lebih Dulu 1215
14.2.1 Priority queue sebagai tipe data abstrak 1215
14.2.2 Mengapa struktur yang paling jelas tidak cukup 1215
14.2.3 Asal-usul heap 1216
14.3 Pohon Biner Complete di dalam Larik 1217
14.3.1 Pohon biner complete 1217
14.3.2 Menomori simpul level demi level 1217
14.3.3 Mengapa rumus-rumus itu benar 1219
14.3.4 Tinggi, dan letak daun-daun 1219
14.4 Sifat Heap 1222
14.4.1 Max-heap dan min-heap 1222
14.4.2 Apa yang dijamin oleh sifat itu 1222
14.4.3 Apa yang tidak dikatakan oleh sifat itu 1223
14.5 Sift-Up: Memperbaiki Kunci yang Terlalu Besar 1223
14.5.1 Prosedurnya 1224
14.5.2 Mengapa ia benar 1225
14.5.3 Berapa ongkosnya 1226

14.6 Sift-Down: Memperbaiki Kunci yang Terlalu Kecil ... 1226
14.6.1 Prosedurnya ... 1226
14.6.2 Mengapa ia benar, dan berapa ongkosnya ... 1227
14.7 Insert, GetMax, dan ExtractMax ... 1228
14.7.1 `getMax` ... 1228
14.7.2 `insert` ... 1228
14.7.3 `extractMax` ... 1228
14.8 GetMin pada Max-Heap ... 1229
14.9 IncreaseKey dan DecreaseKey ... 1230
14.10 Membangun Heap dari Bawah ke Atas dalam Waktu Linear ... 1232
14.10.1 Dua cara membangun ... 1232
14.10.2 Mengapa ongkosnya hanya linear ... 1234
14.10.3 Membangun lewat penyisipan memang lebih lambat ... 1235
14.11 Memeriksa Sifat Heap: `isMaxHeap` ... 1238
14.12 Kelas `MaxHeap` yang Lengkap ... 1238
14.12.1 Antarmukanya ... 1238
14.12.2 Kelas C++ ... 1239
14.12.3 Kelas yang sama dalam Java dan Python ... 1245
14.13 Heap Sort secara Utuh ... 1253
14.13.1 Algoritmanya ... 1253
14.13.2 Kebenaran ... 1257
14.13.3 Ongkos ... 1257
14.13.4 Mengapa heap sort tidak stabil ... 1258
14.14 Priority Queue dalam Pustaka Standar ... 1259
14.14.1 C++: `std::priority_queue` ... 1259
14.14.2 Java: `java.util.PriorityQueue` ... 1260
14.14.3 Python: `heapq`, termasuk fungsi max-heap yang baru ... 1262
14.15 Aplikasi: Top-k, Menggabung k Senarai Terurut, dan Dijkstra ... 1263
14.15.1 k butir terbesar dari sebuah aliran data ... 1264
14.15.2 Menggabung k senarai terurut ... 1264
14.15.3 Kait ke Bab 13: lintasan terpendek Dijkstra ... 1267
14.16 Profitina dalam Praktik ... 1270
14.17 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya ... 1271
14.18 Ringkasan Bab ... 1272
14.19 Soal Latihan ... 1273
14.20 Penyelesaian Soal Latihan ... 1274
Daftar Pustaka ... 1280

BAB 15 Hashing dan Tabel Hash ... 1282
15.1 Dari Mana Kita Melanjutkan ... 1282
15.2 Dari $O(n)$ ke $O(\lg n)$ ke $O(1)$: Gagasan Hashing ... 1283
15.2.1 Pustakawan yang tidak pernah mencari ... 1283
15.2.2 Pengalamatan langsung ... 1283
15.2.3 Hashing: kosakatanya ... 1284
15.2.4 Perjalanan itu dalam satu tabel ... 1285
15.3 Fungsi Hash ... 1285
15.3.1 Apa yang harus dilakukan fungsi hash yang baik ... 1285
15.3.2 Metode pembagian ... 1286
15.3.3 Metode perkalian ... 1287
15.3.4 Meng-hash *string* ... 1287
15.3.5 Pembaruan bergulir ... 1289
15.3.6 *Avalanche*: seberapa baik sebuah fungsi mengaduk? ... 1289
15.3.7 Programnya ... 1290
15.4 Tabrakan dan Batas Ulang Tahun ... 1295
15.4.1 Tabrakan tidak dapat dihindari ... 1295

15.4.2 Soal ulang tahun 1295
15.4.3 Bilangan-bilangannya, dihitung 1296
15.4.4 Artinya bagi tabel hash 1299
15.5 *Separate Chaining* 1299
15.5.1 Sebuah *linked list* di setiap *bucket* 1299
15.5.2 Programnya 1300
15.5.3 Tabel yang sama dalam Java dan Python 1306
15.5.4 Seberapa panjang rantainya? 1313
15.6 Pengalamatan Terbuka 1314
15.6.1 Kunci yang berpindah, bukan rantai yang memanjang 1314
15.6.2 *Linear probing* 1314
15.6.3 *Quadratic probing* 1315
15.6.4 *Double hashing* 1315
15.6.5 Penghapusan dan nisan 1316
15.6.6 Programnya 1317
15.6.7 *Chaining* atau pengalamatan terbuka? 1328
15.7 Faktor Muat dan *Rehashing*, Diukur 1329
15.7.1 Satu-satunya bilangan yang penting 1329
15.7.2 Apa yang diramalkan teori 1329
15.7.3 Pengukurannya 1330
15.7.4 Bagaimana panjang rantai tersebar 1336
15.7.5 *Rehashing*, dan mengapa penggandaan itu murah 1337
15.8 Kompleksitas, Kasus Terburuk, dan *Hash Flooding* 1338
15.8.1 Tagihan lengkapnya 1338
15.8.2 *Hash flooding* 1338
15.8.3 Hashing teracak: pertahanannya 1342
15.9 `std::unordered_map` versus `std::map` 1343
15.9.1 Antarmuka sama, mesin berbeda 1343
15.9.2 Programnya 1344
15.10 Di Dalam `HashMap` Milik Java dan `dict` Milik Python 1348
15.10.1 Java: `HashMap` dan *bin*-nya yang menjadi pohon 1348
15.10.2 Python: `dict` yang ringkas dan berurutan menurut penyisipan 1353
15.11 Penerapan 1356
15.11.1 Menghitung kata dalam katalog 1360
15.11.2 Mendeteksi duplikat 1360
15.11.3 *Two-sum* 1360
15.11.4 *Cache* 1361
15.11.5 *Map* terbalik 1361
15.12 Profitina dalam Praktik 1361
15.13 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 1362
15.14 Ringkasan Bab 1363
15.15 Soal Latihan 1364
15.16 Penyelesaian Soal Latihan 1366
Daftar Pustaka 1372

BAB 16 Evaluasi dan Pembahasan IV: Graf, Heap, dan Tabel Hash dalam Praktik 1373
16.1 Titik Tolak Kita 1373
16.2 Cara Menghadapi Soal tentang Sebuah Struktur 1375
16.3 Soal 16.1 — Melengkapi Kelas Max-Heap 1378
16.3.1 Soalnya 1378
16.3.2 Dengan kata-kata kita sendiri 1379
16.3.3 Dengan tangan, selangkah demi selangkah 1380
16.3.4 Penyelesaian C++ 1383
16.3.5 Penyelesaian Java dan Python 1390
16.3.6 Penyelesaian keliru yang menggoda 1398

16.3.7 Uji diri 1399
16.4 Soal 16.2 — Menyusun Tiket Menurut Urutan Perjalanan 1399
16.4.1 Soalnya 1399
16.4.2 Dengan kata-kata kita sendiri 1400
16.4.3 Dengan tangan, selangkah demi selangkah 1400
16.4.4 *Map* mana yang merupakan *hash map*? 1401
16.4.5 Penyelesaian C++ 1403
16.4.6 Penyelesaian Java dan Python 1406
16.4.7 Penyelesaian keliru yang menggoda 1410
16.4.8 Uji diri 1410
16.5 Soal Baru yang Dikerjakan: Graf 1410
16.5.1 Soal 16.3 — Lompatan paling sedikit antarcabang 1410
16.5.2 Soal 16.4 — Urutan membaca topik-topik 1420
16.5.3 Soal 16.5 — Rute tercepat untuk mobil buku 1427
16.6 Soal Baru yang Dikerjakan: Heap 1436
16.6.1 Soal 16.6 — k nilai terbesar dengan heap berukuran k 1436
16.6.2 Soal 16.7 — Median sejauh ini, dengan dua heap 1446
16.7 Soal Baru yang Dikerjakan: Hashing 1453
16.7.1 Soal 16.8 — Tabel hash dengan separate chaining, dengan tangan 1453
16.7.2 Soal 16.9 — Open addressing, dan menghapus dengan aman 1464
16.8 Memilih Struktur Data yang Tepat 1473
16.8.1 Mulailah dari operasinya, bukan dari strukturnya 1473
16.8.2 Percobaannya 1475
16.8.3 Membaca angka-angkanya 1502
16.8.4 Pilihannya, dalam satu paragraf 1504
16.9 Perbandingan Soal-Soal dalam Bab Ini 1505
16.10 Profitina dalam Praktik 1505
16.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 1506
16.12 Ringkasan Bab 1507
16.13 Soal Latihan 1508
16.14 Penyelesaian Soal Latihan 1509
Daftar Pustaka 1525

Daftar Gambar

Gambar 1.1 Enam keluarga struktur data dalam buku ini: array, *linked list*, *stack* dan *queue*, *tree* (bersama *heap*), graf, dan tabel hash, masing-masing diberi label bab yang membangunnya. 4

Gambar 1.2 Dari berkas sumber ke program yang berjalan: C++ dikompilasi menjadi kode mesin, Java menjadi *bytecode* untuk Java Virtual Machine, dan Python menjadi *bytecode* yang langsung dijalankan mesin virtualnya sendiri. 7

Gambar 1.3 Dua belas byte memori yang menyimpan sebuah `int`, sebuah `char`, sebuah `short`, dan sebuah `float`; setiap tipe menentukan berapa byte yang dipakai sebuah nilai dan bagaimana byte itu dibaca, dan byte beralamat terendah datang lebih dahulu. 9

Gambar 1.4 Memori program C++ yang sedang berjalan: kode dan data global di bawah, *heap* tumbuh ke atas, *stack* tumbuh ke bawah, dan, di kanan, *frame* milik `main` yang pointer `onHeap`-nya merujuk ke sebuah `int` di *heap*. 13

Gambar 1.5 Array `loans` berisi delapan `int` di memori: setiap elemen menempati 4 byte, sehingga elemen i bermula di $1000 + 4i$. 27

Gambar 1.6 Katalog sebagai delapan rekaman `Book` berukuran 196 byte yang disambung ujung ke ujung, dan isi satu rekaman: keempat field-nya bermula di offset 0, 6, 106, dan 166. 30

Gambar 1.7 Array `shelf` berukuran 2×4 sebagai kisi, dan sebagaimana ia sungguh-sungguh terbaring di memori: baris 0 lebih dahulu, lalu baris 1, setiap elemen berjarak 196 byte dari elemen sebelumnya. 40

Gambar 1.8 `std::vector` berkapasitas 8 yang sudah penuh menerima elemen kesembilan: blok berisi 16 dipesan, kedelapan elemen disalin, blok lama dibebaskan, dan 90 ditambahkan. 48

Gambar 1.9 `list` Python menyimpan referensi ke objek-objek terpisah; `array('i')` dari modul `array` menyimpan nilai mentah 4 byte itu sendiri, berdempetan. 51

Gambar 1.10 Menyisipkan 25 di indeks 2: elemen-elemen di indeks 4, 3, dan 2 bergeser satu tempat ke kanan, dalam urutan itu, lalu 25 ditulis ke slot yang dikosongkan. 56

Gambar 2.1 Pencarian sekuensial untuk kode 088 pada rak dalam urutan kedatangan. Lingkaran bernomor memberi urutan pemeriksaan; pencarian berhenti pada pemeriksaan kelima, dan tiga sel terakhir tidak pernah diperiksa. 86

Gambar 2.2 Pencarian sekuensial dengan sentinel. Kunci 5 disalin ke sel cadangan di indeks 6, sehingga gelung pasti berhenti paling lambat di sana; berhenti di indeks $6 = n$ berarti kunci tidak ada di antara elemen sesungguhnya. 97

Gambar 2.3 Pencarian biner untuk 17 dalam [3,9,11,12,15,17,23,31,35]. Sel abu-abu sudah dikesampingkan; sel kuning adalah sel yang diperiksa. Tiga pemeriksaan menemukan kuncinya. 102

Gambar 2.4 Invarian gelung pencarian biner. Semua yang di kiri *low* diketahui lebih kecil daripada kunci, semua yang di kanan *high* diketahui lebih besar, dan kunci, jika ada, berada di wilayah tengah. 103

Gambar 2.5 *Lower bound* dan *upper bound* dari 2005 dalam daftar tahun terbit yang terurut. Tiga salinan 2005 menempati indeks 3 sampai 5, yaitu rentang setengah terbuka 3,6. Untuk 2003 yang tidak ada, *lower bound* dan *upper bound* sama-sama 3, tempat 2003 akan disisipkan. 112

Gambar 2.6 Pencarian interpolasi pada rak komputer. Garis putus-putus membentang lurus dari code[0] = 025 ke code[7] = 096. Untuk kunci 088 ia menaksir posisi 6,21, dan pemeriksaan di indeks 6 langsung menemukan 088; untuk kunci 060 ia menaksir 3,45 dan memeriksa indeks 3. 120

Gambar 2.7 Pencarian lompat untuk 52 dengan blok berisi empat. Lompatan memeriksa elemen terakhir setiap blok (15, 34, 58); karena $58 > 52$, kunci pasti berada di blok 3, dan penelusuran melalui 41, 47, 52 menemukannya. 126

Gambar 2.8 Larik 3×5 dan tata letak baris-mayornya di memori. Setiap sel memori diberi label baris dan kolom elemen yang dimuatnya; angka kecil adalah alamatnya bila sebuah `int` menempati 2 byte dan larik dimulai di 0x1000. 131

Gambar 2.9 Metode tangga pada kisi 4×5 yang baris dan kolomnya menaik. Untuk 37, lintasan 50, 40, 30, 35, 37 menemukan kunci dalam lima pemeriksaan; untuk 34, lintasan bergeser ke kiri dan ke bawah sampai keluar dari kisi di bawah 33. 133

Gambar 2.10 Nanodetik per pencarian pada katalog 10.000 buku, menurut metode pencarian dan bahasa, pada skala logaritmik. Diukur oleh **Kode 2.22**, **Kode 2.23**, dan **Kode 2.24** pada September 2026.....156
Gambar 2.11 Rata-rata pemeriksaan per pencarian berhasil ketika n tumbuh dari 103 sampai 106, pada sumbu logaritmik. Pencarian sekuensial naik dengan kemiringan 1, pencarian lompat dengan kemiringan 12; pencarian biner naik sekitar 3,3 pemeriksaan untuk setiap kenaikan sepuluh kali lipat; pencarian interpolasi pada kunci seragam nyaris tidak naik, tetapi pada kunci kubik ia mendaki curam.....166
Gambar 2.12 Pohon keputusan pencarian biner pada tujuh elemen terurut. Setiap lingkaran membandingkan kunci dengan satu elemen; setiap kotak adalah satu dari delapan celah tempat kunci yang tidak ada dapat jatuh. Setiap lintasan dari akar ke kotak memuat tiga perbandingan.....174
Gambar 3.1 Keluarga metode pengurutan dalam bab ini. Lima yang pertama membandingkan kunci; yang terakhir langsung memakai digit-digit kunci.....198
Gambar 3.2 Menukar dua nilai dengan variabel cadangan. Setiap kotak yang disorot adalah kotak yang berubah pada langkah itu.....199
Gambar 3.3 *Bubble sort* pada $A = [22,10,15,3,8,2]$, satu baris per perbandingan. Lintasan 1 membawa 22 ke ujung, lintasan 2 membawa 15, dan seterusnya; lima lintasan dan lima belas perbandingan seluruhnya.....205
Gambar 3.4 *Exchange sort* pada A. Pada putaran 1 nilai di posisi 0 berganti tiga kali, $22 \rightarrow 10 \rightarrow 3 \rightarrow 2$, sebelum nilai terkecil menetap di sana.....211
Gambar 3.5 *Selection sort* pada A. Putaran 1 menemukan minimum 2 di indeks 5 dan melakukan satu pertukaran; putaran 4 dan 5 mendapati minimumnya sudah di tempat dan tidak menukar apa pun.....215
Gambar 3.6 *Insertion sort* pada A. Setiap langkah mengambil sebuah kunci, menggeser elemen yang lebih besar di awalan terurut satu tempat ke kanan, lalu menjatuhkan kunci ke celahnya.....219
Gambar 3.7 *Shell sort* pada A dengan celah 3 dan 1. Mengurutkan tiga pasangan yang berselang-seling menghapus 9 dari 13 inversi sekaligus; *insertion sort* kemudian menghapus 4 sisanya.....224
Gambar 3.8 *Merge sort* pada A. Pembagian (atas) membelah larik sampai setiap potongan berisi satu elemen; penggabungan (bawah) menyatukan *run* terurut, dan perbandingan pada setiap tingkat dicatat di sebelah kanan.....232
Gambar 3.9 Partisi Lomuto. Atas: mempartisi [10,15,3,8] di sekitar pivot 8. Bawah: empat partisi yang mengurutkan A dengan elemen terakhir sebagai pivot.....237
Gambar 3.10 Partisi Hoare pada A dengan pivot 15. Dua pertukaran sudah cukup; kedua indeks bersilangan dengan $j = 3$, membelah A menjadi [2,10,8,3] dan [15,22].....238
Gambar 3.11 *Heap sort* pada A. Setiap panel memperlihatkan *heap* sebagai pohon (akar berwarna kuning tua) dan seluruh larik di bawahnya; ekor berwarna hijau toska adalah bagian yang sudah terurut, yang bertambah satu elemen pada setiap langkah.....252
Gambar 3.12 *Counting sort* pada A dengan kunci 0 sampai 22. Hitungan (kuning tua) menjadi jumlah kumulatif (biru), dan setiap jumlah kumulatif menyatakan di mana kuncinya berakhir dalam keluaran.....259
Gambar 3.13 *Radix sort* LSD pada A. Lintasan pertama menyebar menurut digit satuan; lintasan kedua, menurut digit puluhan, menjaga setiap ember dalam urutan lintasan pertama, dan larik keluar dalam keadaan terurut.....260
Gambar 3.14 Mengurutkan katalog menurut kategori. Hanya pengurutan stabil yang mempertahankan keempat buku *Fiksi* dalam urutan id aslinya; *selection sort* memindahkan B1001 ke belakang B1002 dan B1003.....268
Gambar 3.15 Waktu terukur program-program C++, pada sumbu log–log. (a) Ketiga pengurutan kuadratik naik dua dekade waktu untuk setiap dekade n; `std::sort` ditampilkan sebagai pembanding skala. (b) Pengurutan cepat hanya naik sedikit lebih dari satu dekade per dekade; *radix sort* yang tercepat.....277
Gambar 3.16 Perbandingan per elemen. Pengurutan kuadratik menanjak curam; yang lain tetap dekat dengan lg$n$, dengan *merge sort* paling hemat dan *heap sort* paling boros.....278
Gambar 4.1 Delapan langkah yang dipakai untuk setiap soal dalam bab ini; panah putus-putus adalah putaran antara menjalankan program dan memperbaikinya.....315
Gambar 4.2 Insertion sort atas delapan kata: setelah langkah i, $i + 1$ kata pertama (hijau toska) sudah menurut urutan kamus, dan sel kuning adalah tempat kata kunci mendarat.....320

Gambar 4.3 (a) Lintasan 1 mengisi pencacah, lintasan 2 berhenti pada karakter pertama yang pencacahnya sedikitnya 2; (b) berhenti pada karakter pertama yang pernah terlihat memberi jawaban keliru b. 327
Gambar 4.4 (a) Pencarian sekuensial menemukan Pujo pada pemeriksaan keempat; (b) pencarian biner pada data yang sama yang tidak terurut membuang bagian yang memuat Pujo dan melaporkan Not found. 333
Gambar 4.5 Keenam string dengan kedua kuncinya, menurut urutan masukan (kiri) dan terurut dari yang terpenting ke yang paling tidak penting (kanan); warna menandai tiga kelompok kunci pertama....339
Gambar 4.6 (a) Sepuluh kotak A beserta alamatnya; (b) kisi B, sel-selnya diberi nomor menurut urutan di memori, dan sel-sel yang sama sebagaimana sungguh-sungguh terletak di memori. 346
Gambar 4.7 Batas bawah dan batas atas 2001 dan 2023, dengan posisi lewat-ujung 9 digambar putus-putus, dan penelusuran LOWER-BOUND(2001) langkah demi langkah. 351
Gambar 4.8 Pencarian interpolasi memproyeksikan kunci dari sumbu nilai kode ke sumbu indeks; 88 mendarat di 6,21 dan ditemukan di indeks 6, sedangkan 41 mendarat di 1,58 dan memerlukan pemeriksaan kedua. 357
Gambar 4.9 Enam perbandingan pada lintasan pertama bubble sort; pasangan kuning ditukar, pasangan hijau toska sudah berurutan. 365
Gambar 4.10 Menggabungkan A dan B ke dalam C: setiap elemen C dihubungkan ke asalnya, bilangan kecil menghitung perbandingan, dan ekor B yang putus-putus disalin tanpa perbandingan. 374
Gambar 4.11 (a) Indeks baca r mengunjungi setiap kotak sekali dan indeks tulis w merapatkan bilangan yang disimpan ke depan; (b) menggeser ke kiri lalu selalu memajukan i melewatkan 3 kedua dari dua 3 yang bersebelahan. 380
Gambar 4.12 Jalan dua indeks untuk voucher 37000 pada harga yang terurut, dalam ribuan rupiah: setiap langkah menemukan sebuah pasangan atau membuang harga termurah atau termahal yang tersisa. 385
Gambar 5.1 Dua disiplin dalam bab ini. Stack menambah dan mengambil di ujung yang sama (LIFO); queue menambah di belakang dan mengambil di depan (FIFO). 400
Gambar 5.2 Tujuh langkah yang memindahkan tiga cakram dari tiang A ke tiang C. Panah merah menandai langkah yang menghasilkan setiap keadaan; pada setiap langkah hanya cakram teratas yang berpindah, dan cakram itu tidak pernah mendarat di atas cakram yang lebih kecil. 401
Gambar 5.3 Stack larik berkapasitas 4 melalui delapan operasi. Sel kuning adalah elemen puncak dan panah merah adalah indeks `top`; setelah *push* keempat stack penuh dan *push* kelima ditolak....408
Gambar 5.4 Stack bertaut setelah *push* 10, 20, dan 30 (atas), dan setelah satu *pop* (bawah): pointer puncak melompati simpul pertama yang lama, yang kemudian dibebaskan. 418
Gambar 5.5 Memeriksa `{[()()]}`: setiap pembuka di-*push*, setiap penutup mengambil pasangannya dari puncak, dan stack kosong di akhir, sehingga teksnya seimbang. 421
Gambar 5.6 Stack undo dan redo pada empat saat dalam **Kode 5.8**. Suntingan baru (panel 4) melakukan *push* ke stack undo dan mengosongkan stack redo. 425
Gambar 5.7 Mengubah `3+2*5` menjadi postfiks. `*` di-*push* di atas `+` karena mengikat lebih kuat; di akhir stack dikosongkan dari puncak, menghasilkan `3 2 5 * +`....428
Gambar 5.8 Mengevaluasi `3 2 5 * +`. Operan di-*push*; setiap operator mengambil y (puncak) lalu x, dan melakukan *push* x op y. Satu nilai yang tersisa, 13, adalah hasilnya....429
Gambar 5.9 Call stack selama `lateFee(7, "reference")`: setiap pemanggilan melakukan *push* sebuah *frame* di puncak, dan setiap *return* melakukan *pop*, menyerahkan nilainya kepada *frame* di bawahnya....437
Gambar 5.10 Sebuah queue melalui tujuh operasi, digambar dengan bagian depan di kiri. Elemen keluar persis dalam urutan kedatangannya: 70, lalu 80....439
Gambar 5.11 Queue bergeser: *dequeue* mengambil slot 0 (langkah 1), menggeser setiap elemen lain satu tempat ke kiri (langkah 2), dan mengurangi `tail` (langkah 3). Queue berisi n elemen memerlukan $n - 1$ pergeseran per *dequeue*. 439
Gambar 5.12 Queue yang bergeser. Kedua indeks hanya bergerak ke kanan. Setelah dua *dequeue*, slot 0 dan 1 kosong, tetapi `tail` sudah mencapai slot terakhir, sehingga `isFull()` melaporkan benar dan *enqueue* berikutnya ditolak: *penuh palsu*. 440

Gambar 5.13 Queue sirkular berkapasitas $N = 8$ yang memuat A sampai E. Queue dimulai pada `front` = 5 dan berjalan searah jarum jam melalui slot 6, 7, 0, dan 1: setelah slot 7 ia melingkar ke slot 0. 443
Gambar 5.14 Queue sirkular pada **Kode 5.12** dengan kapasitas 4. Dua *enqueue* terakhir melingkar ke slot 0 dan 1, sehingga bagian belakang kini berada di kiri bagian depan — dan queue tetap berada dalam urutan FIFO yang sempurna. 444
Gambar 5.15 Tanpa pencacah, `front = rear` akan berarti "kosong" (atas) sekaligus "kelima slot terpakai" (bawah). Menyisakan satu slot yang selalu kosong (tengah) membuat keadaan penuh dapat dibedakan: queue penuh bila memajukan `rear` akan mencapai `front`. 451
Gambar 5.16 Deque berbasis larik sirkular berkapasitas 6. `pushFront` memindahkan `front` mundur dengan pelingkaran, dari 0 ke 5 lalu ke 4; `popFront` memajukannya kembali. 454
Gambar 6.1 Enam belas byte memori. Setiap variabel menempati byte-byte yang berurutan; pointer `addressX` menyimpan alamat `0x1003`, sehingga mengikutinya membawa kita kembali ke `x`. 476
Gambar 6.2 Dua cara menugasi dengan pointer. `*q = *p` memindahkan nilai antarkotak; `q = p` memindahkan panah. 481
Gambar 6.3 Aritmetika pointer pada `int a[5]`. Setiap `+1` bergeser satu elemen, empat byte; `p + 5` menunjuk satu lewat ujung, posisi yang boleh dihitung dan dibandingkan tetapi tidak pernah boleh didereferensi. 482
Gambar 6.4 Atas: sebuah `Book*` dan dua cara menulis akses field. Bawah: tiga simpul yang ditautkan oleh pointer `next`-nya, bentuk *linked list* di Bab 7. 492
Gambar 6.5 Variabel otomatis tinggal di *frame stack* fungsinya; objek yang dibuat dengan `new` tinggal di *heap* dan hanya dapat dijangkau melalui pointer. 493
Gambar 6.6 Satu langkah pertumbuhan larik yang menggandakan diri. Elemen-elemen disalin ke blok baru dan blok lama dibebaskan, sehingga pointer yang diambil sebelum pertumbuhan kini menunjuk ke memori yang sudah bebas. 497
Gambar 6.7 Tiga bug umur objek yang digambar sebagai kotak dan panah. Blok yang diarsir sudah dibebaskan. 498
Gambar 6.8 Satu *swap* yang ditulis dengan tiga cara. Berdasarkan nilai, fungsi menukar salinannya sendiri; melalui pointer, fungsi menukar variabel pemanggil lewat alamatnya; melalui referensi, parameternya adalah variabel pemanggil dengan nama baru. 512
Gambar 6.9 Kiri: `unique_ptr` mempunyai satu pemilik, dan *move* mengalihkannya. Kanan: tiga `shared_ptr` berbagi satu `Book` melalui blok kendali yang menghitungnya; `weak_ptr` mengamati tanpa ikut dihitung. 522
Gambar 6.10 Kiri: fungsi memutasi *list* yang dirujuk parameternya, dan pemanggil melihat perubahannya. Kanan: fungsi mengikat ulang parameternya pada *list* baru, dan nama milik pemanggil masih merujuk *list* yang lama. 526
Gambar 7.1 *Single linked list* yang menyimpan 13, 3, 86, dan 74. Setiap simpul memiliki medan data dan medan `next`; `head` menunjuk simpul pertama, dan `next` simpul terakhir bernilai NULL. 551
Gambar 7.2 Lima nilai yang sama dalam larik (satu blok bersebelahan, alamat berurutan) dan dalam *linked list* (simpul di mana pun alokator menaruhnya, disambung oleh pointer `next`). Alamatnya hanya ilustrasi. 552
Gambar 7.3 `frontInsertion(3)` pada list 13, 86: (a) sebelum, (b) simpul baru ditautkan ke simpul pertama lama, (c) `head` pindah ke simpul baru. 560
Gambar 7.4 `backInsertion(74)` pada list 3, 13, 86: (a) `aux` mulai dari `head`, (b) `aux` berhenti pada simpul yang `next`-nya NULL, (c) `next` simpul itu menjadi simpul baru. 561
Gambar 7.5 Menelusuri list 3, 13, 86, 74: `aux` mengunjungi setiap simpul bergiliran dan gelung berakhir ketika `aux` menjadi NULL; `head` tidak pernah bergerak. 562
Gambar 7.6 `frontDeletion()` pada list 3, 13, 86: (a) `del` mengingat simpul pertama, (b) `head` pindah ke simpul kedua, (c) simpul pertama lama dihapus. 563
Gambar 7.7 `backDeletion()` pada list 3, 13, 86, 74: (a) `aux` berhenti di simpul kedua dari belakang, (b) simpul terakhir dilepas dari rantai, (c) simpul itu dihapus. 564
Gambar 7.8 `insertAfter(p, 35)` dengan `p` di 30: (a) sebelum, (b) simpul baru menunjuk penerus `p`, (c) `p` menunjuk simpul baru. 577
Gambar 7.9 `insertSorted`: (a) untuk 20, `aux` berhenti di simpul terakhir yang lebih kecil daripada 20, (b) simpul baru disisipkan sesudahnya, (c) 5 lebih kecil daripada setiap nilai dan menjadi simpul pertama. 578

Gambar 7.10 `deleteKey(35)`: (a) `prev` dan `cur` berhenti dengan `cur` di 35, (b) `prev->next` melompatinya, (c) simpul itu dihapus. 579
Gambar 7.11 Pembalikan iteratif atas 20, 30, 40, 50: sesudah setiap iterasi satu panah lagi menunjuk ke belakang; ketika `cur` mencapai NULL, `prev` adalah simpul pertama yang baru. 580
Gambar 7.12 Pembalikan rekursif atas 20, 30, 40: pemanggilan terdalam mengembalikan 40; setiap pemanggilan yang kembali membalik satu panah dan memutus tautan maju simpulnya sendiri. 581
Gambar 7.13 List dengan simpul kepala *dummy*: (a) list kosong berupa satu simpul *dummy*, (b) list berisi 10, 20, 30, (c) menyisipkan 5 sebelum 10 adalah penyisipan sesudah *dummy*, tanpa kasus khusus. 587
Gambar 7.14 List dengan `head` dan `tail`: (a) sebelum `backInsertion(74)`, (b) simpul terakhir lama menunjuk simpul baru, (c) `tail` pindah ke sana; (d) `backDeletion` tetap harus menjalankan `aux` ke simpul sebelum `tail`. 593
Gambar 7.15 *Single linked list* melingkar: (a) empat simpul, dicapai lewat satu pointer `last`, dengan simpul pertama di `last->next`; (b) lingkaran satu simpul menunjuk dirinya sendiri; (c) menyisipkan di belakang adalah penyisipan sesudah `last` yang diikuti pemindahan `last`. 597
Gambar 7.16 (a) *Stack* berantai: `push` dan `pop` bekerja pada simpul pertama, `top_`. (b) *Queue* berantai: `enqueue` menautkan simpul baru sesudah `rear_`, `dequeue` membuang simpul pada `front_`. 602
Gambar 7.17 Penyisipan depan dengan kedua penugasannya dalam urutan yang salah: (a) `head` berpindah lebih dulu, (b) simpul baru lalu menunjuk dirinya sendiri dan simpul lama 13 dan 86 tidak akan pernah dapat dicapai lagi. 607
Gambar 7.18 Hasil pengukuran **Kode 7.16**: penyisipan depan kuadratik untuk larik dan linear untuk list (kiri); menjumlahkan dua juta nilai paling cepat pada larik dan sekitar seratus kali lebih lambat pada list yang simpulnya tersebar (kanan). 622
Gambar 8.1 Stack larik Soal 8.1 sesudah setiap operasi *driver*: `top` adalah indeks butir teratas, dan `pop` hanya menggeser `top`; nilai 30 tetap berada di larik tetapi bukan lagi bagian dari stack. 641
Gambar 8.2 List dua arah yang dibangun dari contoh masukan Soal 8.2: sembilan simpul, masing-masing dengan pointer `prev` dan `next`. Simpul 8 memuat baris akhir `0 0 0`, yang juga ikut disimpan oleh perulangan pembacaan yang diberikan. 650
Gambar 8.3 Delapan gerakan penelusuran: gerakan maju di atas list, gerakan mundur di bawahnya, diberi nomor menurut urutan terjadinya. Setiap simpul berhuruf dikunjungi tepat sekali. 650
Gambar 8.4 Menyisipkan 98 ke list naik `evens`: `cur` berhenti di simpul terakhir yang tetap di depan (34); simpul baru ditautkan ke 654 lebih dulu, dan baru sesudah itu 34 ditautkan ke simpul baru. 660
Gambar 8.5 Membangun list Soal 8.4 dengan penyisipan depan: setiap kata baru menjadi simpul pertama, sehingga list membaca kalimatnya dari belakang. 668
Gambar 8.6 Menghapus salinan yang datang belakangan: `runner` berdiri di pendahulu duplikat, menautkan melompatinya, dan baru sesudah itu membebaskannya; `runner` tidak bergeser, karena penerusnya yang baru belum diperiksa. 673
Gambar 8.7 Queue sirkular Soal 8.7 yang digambar sebagai cincin. Panah jingga adalah `front`, panah hijau toska adalah slot kosong berikutnya, `rear`. Sesudah dua kali *dequeue* (kiri), permintaan berikutnya masuk ke sel 0; sesudah dua kali *enqueue* lagi (kanan), cincin penuh, dan urutan queue dimulai di sel 2 lalu berputar. 689
Gambar 8.8 Tiga cara mengirim `balance`, digambar pada akhir setiap pemanggilan. By value, fungsi bekerja pada salinan yang lenyap ketika fungsi selesai. By reference, `fine` adalah nama lain untuk `balance` itu sendiri. By pointer, `fine` menyimpan alamat `balance`, dan `*fine` mengikutinya. 703
Gambar 8.9 Satu langkah pembalikan di tempat, yaitu langkah ketiga: simpan `cur->next` di `after`, balik panah `cur` ke `prev`, lalu majukan `prev` dan `cur` satu simpul. Simpul berwarna jingga adalah bagian yang sudah dibalik. 711
Gambar 8.10 Menggabungkan list kedua cabang: hasilnya bergantung pada simpul dummy, `tail` menandai simpul terakhirnya, dan perbandingan berikutnya adalah antara simpul depan `a` dan `b`. 717
Gambar 8.11 Queue bertaut Soal 8.12. *Enqueue* bekerja di `tail` dan *dequeue* di `head`; ketika simpul terakhir pergi, `tail` juga harus disetel ulang menjadi `null`, atau ia tertinggal menunjuk ke memori yang sudah dibebaskan. 724

Gambar 9.1 *Double linked list* yang berisi 13, 3, 86, dan 74. Setiap simpul punya medan `prev`, medan data, dan medan `next`; `head` menunjuk simpul pertama dan `tail` simpul terakhir, dan kedua ujung rantai bernilai NULL. 751
Gambar 9.2 Tata letak memori simpul yang berisi `int` pada mesin 64-bit: 16 bita untuk simpul *single linked*, 24 bita untuk simpul *double linked*. Pengisi menjaga setiap pointer berada di alamat yang habis dibagi 8. 754
Gambar 9.3 `frontInsertion(3)` pada list 13, 86: (a) sebelum, (b) simpul baru menunjuk maju ke simpul pertama lama, (c) simpul pertama lama menunjuk balik ke simpul baru, (d) `head` pindah ke simpul baru. 756
Gambar 9.4 `backInsertion(74)` pada list 3, 13, 86: (a) `aux` berhenti di simpul terakhir, (b) dua tautan dipasang, satu untuk setiap arah, (c) sesudahnya. 757
Gambar 9.5 Menyusuri list 3, 13, 86, 74: (a) maju dari `head`, mengikuti `next`; (b) mundur, mengikuti `prev` dari simpul terakhir. Angka-angka menunjukkan urutan kunjungan. 758
Gambar 9.6 `frontDeletion()` pada list 3, 13, 86: (a) `del` mengingat simpul pertama, (b) `head` pindah ke 13 dan `prev` milik 13 menjadi NULL, (c) simpul pertama lama dihapus. 759
Gambar 9.7 `backDeletion()` dengan `head` saja pada list 3, 13, 86, 74: (a) `del` berjalan ke simpul terakhir, (b) pendahulunya, `del->prev`, berhenti menunjuknya, (c) simpul itu dihapus. 760
Gambar 9.8 Penghapusan di belakang dengan pointer *tail* pada 3, 13, 86, 74: (a) `del` = `tail`, (b) `tail` mundur menyusuri `prev` dan `next` milik simpul terakhir baru menjadi NULL, (c) sesudahnya: $O(1)$, tanpa berjalan. 772
Gambar 9.9 `insertAfter(p, 35)` dengan `p` di 30: (a) sebelum, (b) simpul baru menunjuk kedua tetangganya, (c) kedua tetangga menunjuk simpul baru, (d) sesudahnya. 773
Gambar 9.10 `insertSorted`: (a) untuk 40, `aux` berhenti di 50, nilai pertama yang lebih besar, (b) simpul baru ditautkan sebelum `aux` dalam empat langkah, (c) list sesudah 5 dan 60 juga disisipkan, di depan dan di belakang. 775
Gambar 9.11 `deleteKey(35)`: (a) `search` mengembalikan `p`, (b) pendahulu dan penerus ditautkan satu sama lain, (c) `p` dihapus. 776
Gambar 9.12 *Double linked list* dengan simpul kepala: (a) list kosong berupa satu simpul *dummy*, dan `tail` juga menunjuknya, (b) list 10, 20, 40, (c) penyisipan di depan adalah `insertAfter(head, 5)`, dengan empat langkah yang sama seperti penyisipan mana pun. 782
Gambar 9.13 *Double linked list* sirkular yang dicapai lewat `head`: (a) empat simpul; simpul terakhir adalah `head->prev`, (b) lingkaran satu simpul, (c) `insertBack(9)` menautkan simpul baru di antara simpul terakhir dan `head`. 785
Gambar 9.14 *Double linked list* sirkular dengan *sentinel*: (a) list kosong adalah *sentinel* seorang diri, menunjuk dirinya ke dua arah, (b) list 10, 20, 30; `nil.next` adalah simpul pertama dan `nil.prev` yang terakhir. 790
Gambar 9.15 Dua kasus yang dulu butuh kode khusus, kini ditangani kode biasa: (a) menyisipkan sesudah *sentinel* ke list kosong, (b) menghapus satu-satunya simpul. 791
Gambar 9.16 Meja pengembalian sebagai *deque* di atas *double linked list* sirkular dengan *sentinel*: depan adalah `nil.next`, belakang adalah `nil.prev`, dan keempat operasi bernilai $O(1)$. 796
Gambar 9.17 Panel "baru dipinjam": *double linked list* sirkular dengan *sentinel* menyimpan judul dari yang terbaru sampai yang terlama, dan *hash map* menunjuk dari setiap judul ke simpulnya. (a) Sesudah tiga peminjaman; (b) sesudah *Laskar* dipinjam lagi dan dipindah ke depan; *map*-nya sendiri tidak berubah. 800
Gambar 9.18 Hasil pengukuran **Kode 9.11** pada sumbu logaritmik: waktu *single linked list* tumbuh empat kali per penggandaan n (kuadratik), waktu *double linked list* dua kali (linear). 808
Gambar 9.19 Tiga galat tautan dua arah: (a) `insertAfter` dengan urutan langkah yang salah membuat simpul baru menunjuk dirinya sendiri, (b) tautan mundur yang terlupa membuat list terbaca berbeda di kedua arah, (c) tautan mundur yang terlupa pada penghapusan meninggalkan pointer ke memori yang sudah dibebaskan. 809
Gambar 10.1 Menghitung orang di depan dengan bertanya: setiap pertanyaan maju satu tempat (biru tua), orang pertama menjawab 0 tanpa bertanya kepada siapa pun, dan setiap jawaban kembali dengan tambahan satu (kuning). 836
Gambar 10.2 *Call stack* selama `fact(4)` berjalan. Baris atas: setiap panggilan menunggu panggilan yang lebih kecil sampai *base case* `fact(0)` berada di puncak. Baris bawah: setiap *return* melakukan *pop* satu *frame* dan menuntaskan satu perkalian di *frame* di bawahnya. 843

Gambar 10.3 Enam bulan pertama kelinci Fibonacci. Panah abu-abu menunjukkan sepasang kelinci yang terus hidup ke bulan berikutnya; panah kuning menunjukkan pasangan dewasa yang melahirkan pasangan muda baru. 850
Gambar 10.4 Panggilan-panggilan untuk menghitung $F(5)$. (a) Rekursi naif: 15 panggilan, sebagian besar menghitung ulang nilai yang sudah diketahui. (b) Dengan memoisasi: 9 panggilan; nilai yang sudah ada di tabel dikembalikan tanpa panggilan apa pun di bawahnya. 851
Gambar 10.5 (a) Banyaknya panggilan untuk menghitung $F(n)$ dengan ketiga metode rekursif, skala logaritmik. (b) Waktu rekursi naif yang diukur dalam C++, Java, dan Python, September 2026, skala logaritmik....860
Gambar 10.6 Algoritma Euklides pada 228 dan 90 sebagai pemotongan persegi dari persegi panjang: setiap pembagian membuang sebanyak mungkin persegi bersisi pendek; persegi terakhir, 6×6, tepat menutup sisanya, jadi $\gcd(228,90) = 6$. 862
Gambar 10.7 Menghitung 313. (a) Mengurangi pangkat dengan satu: 14 panggilan. (b) Membagi dua pangkatnya: 5 panggilan; di bawah setiap panggilan tertera cara hasilnya dibangun dari panggilan berikutnya....866
Gambar 10.8 Penyelesaian rekursif Menara Hanoi: singkirkan $n-1$ cakram yang lebih kecil, pindahkan cakram terbesar, lalu pindahkan kembali $n-1$ cakram itu ke atasnya. 870
Gambar 10.9 Pohon rekursi `hanoi(3, A, C, B)`; `h(k, X→Y)` memindahkan k cakram dari X ke Y. Setiap panggilan membuat langkahnya sendiri di antara langkah-langkah kedua anaknya; angka merah memberikan urutan langkah-langkahnya. 876
Gambar 10.10 Pohon rekursi dengan ongkos setiap aras di sebelah kanan. (a) *Merge sort*: $\lg n + 1$ aras yang masing-masing berongkos cn. (b) Hanoi: aras i berongkos $2i$, dan jumlah semua aras $2n-1$. 883
Gambar 10.11 Kiri: tanpa optimisasi, `sumPlain(5)` memiliki enam *frame* miliknya sendiri di *stack* pada titik terdalam. Kanan: dengan `-O2`, *tail call* `sumAcc` dikompilasi sebagai lompatan, dan satu *frame* melayani seluruh rekursi (GCC melakukan hal yang sama untuk `sumPlain` setelah memperkenalkan akumulator). 888
Gambar 10.12 *Stack* eksplisit selama `hanoiIter(2)` berjalan; setiap *frame* memperlihatkan (n, tiang asal dan tujuan, tahap), dan *frame* teratas disorot. Setiap kolom adalah *stack* setelah satu putaran gelung; langkah-langkahnya tertera di bawah putaran yang membuatnya. 897
Gambar 10.13 Menu layanan Profitina Laundry: kategori (hijau kebiruan) memuat layanan (kuning) dan kategori yang lebih kecil, sampai kedalaman berapa pun. 901
Gambar 11.1 Empat pohon: (a) pohon keluarga yang memuat para leluhur, (b) pohon urai yang dibangun kompilator untuk sebuah ekspresi aritmetika, (c) folder dan berkas pada sebuah diska, (d) bagan pertandingan sistem gugur. 917
Gambar 11.2 Sebuah pohon dengan sebelas simpul. Simpul internal berwarna abu-abu, daun berwarna hijau-biru; segitiga kuning menandai subpohon berakar C, dan kedalaman setiap tingkat tertera di kanan. 919
Gambar 11.3 Satu pohon dalam tiga notasi: (a) gambar biasa beserta tingkat-tingkatnya, (b) diagram Venn, yang menggambar setiap subpohon sebagai himpunan di dalam himpunan induknya, (c) notasi tingkat, yang menggambar setiap simpul sebagai batang yang menjorok sesuai kedalamannya....921
Gambar 11.4 Representasi statis: (a) setiap simpul dinomori tingkat demi tingkat, (b) simpul i disimpan di slot i, dan anak-anaknya ditemukan di slot $2i+1$ dan $2i+2$ menurut Persamaan (11.2). 922
Gambar 11.5 Representasi dinamis pohon biner: setiap simpul memiliki pointer kiri, medan data, dan pointer kanan; `root` menunjuk simpul teratas, dan setiap anak yang tidak ada adalah pointer NULL. 925
Gambar 11.6 Pohon biner dengan empat belas simpul: simpul 43 adalah induk dari anak kirinya 33 dan anak kanannya 97, dan 64, yang tidak memiliki anak, adalah daun. 926
Gambar 11.7 Bentuk-bentuk pohon biner: (a) *full*, (b) *perfect*, (c) *complete*, (d) tidak *complete*, karena posisi 4 pada tingkat terakhir kosong sementara posisi 5 terisi, (e) condong ke kanan, (f) condong ke kiri. 927
Gambar 11.8 Urutan setiap penelusuran mengunjungi simpul-simpul pohon yang sama; lencana kuning pada sebuah simpul adalah posisinya dalam barisan kunjungan. 929

Gambar 11.9 Perjalanan mengelilingi pohon menyentuh setiap simpul tiga kali: mendaftar simpul ketika dilewati di sisi kiri menghasilkan *preorder*, di sisi bawah menghasilkan *inorder*, dan di sisi kanan menghasilkan *postorder*. 930
Gambar 11.10 Urutan tingkat pada pohon *perfect* bersimpul tujuh: (a) urutan kunjungan, (b) isi *queue* tepat sebelum setiap kunjungan, dengan simpul yang akan keluar berwarna kuning. 931
Gambar 11.11 *Stack* eksplisit dari **Pseudokode 11.3** pada setiap kunjungan dari lima kunjungan: simpul hijau-biru berada di *stack*, simpul kuning sedang dikunjungi, dan simpul abu-abu sudah dikunjungi. 933
Gambar 11.12 Pohon ekspresi dari `3 + 2 * 5 - 4` (a) dan tiga penelusuran telusur-dalamnya (b): *preorder* menghasilkan notasi prefiks, *inorder* menghasilkan notasi infiks, dan *postorder* menghasilkan notasi postfiks. 938
Gambar 11.13 *Stack* berisi pohon selama `3 2 5 * + 4 -` dibaca: operan mendorong pohon bersimpul satu, dan setiap operator menggabungkan dua pohon terbaru menjadi satu. 939
Gambar 11.14 (a) Sifat BST di akar: setiap kunci subpohon kiri lebih kecil daripada 65 dan setiap kunci subpohon kanan lebih besar; (b) dan (c) probe-probe *binary search* pada larik terurut membentuk sebuah *binary search tree*. 941
Gambar 11.15 Pohon sesudah setiap panggilan `insert(65, 5, 70, 3, 10)` dan satu `insert(70)` ulangan: simpul hijau-biru dan sisi kuning adalah lintasan pencarian, simpul kuning adalah daun baru, dan 70 yang kembar ditemukan lalu ditolak. 943
Gambar 11.16 Dua pencarian: (a) `search(8)` mengikuti 6, 10, 7 dan menemukan 8; (b) `search(18)` mengikuti 6 dan 10, jatuh keluar dari pohon pada subpohon kanan 10 yang kosong, dan melaporkan bahwa 18 tidak ada. 944
Gambar 11.17 Minimum, 1, dicapai dengan terus ke kiri dari akar sampai tidak ada anak kiri; maksimum, 10, dengan terus ke kanan sampai tidak ada anak kanan. 945
Gambar 11.18 Dua kasus *successor*: (a) 30 memiliki subpohon kanan, dan *successor*-nya adalah minimum subpohon itu, 35; (b) 45 tidak memiliki subpohon kanan, dan *successor*-nya adalah leluhur tempat lintasan dari akar terakhir kali berbelok ke kiri, 50. 961
Gambar 11.19 Tiga kasus penghapusan, diterapkan satu per satu pada pohon `insert(65, 5, 70, 3, 10)`: daun dilepas (a, b); simpul beranak satu digantikan oleh anaknya (c, d); simpul beranak dua mengambil kunci *successor*-nya, yang kemudian dihapus dari subpohon kanan (e, f). 962
Gambar 11.20 Menghapus simpul beranak dua dengan kedua cara: (a) akar 50 memiliki *successor* 60 dan *predecessor* 45; (b) dengan *successor*, 60 naik dan anak kanannya 65 menempati posisi lamanya (kasus 2); (c) dengan *predecessor*, 45 naik dan sebuah daun hilang (kasus 1). 964
Gambar 11.21 Tujuh kunci yang sama dalam dua urutan penyisipan: (a) dalam urutan terurut kunci-kunci itu membentuk rantai bertinggi 6; (b) bila yang tengah disisipkan lebih dahulu, terbentuk pohon *perfect* bertinggi 2. 972
Gambar 11.22 Hasil pengukuran **Kode 11.8**: dalam urutan acak, tinggi dan kedalaman rata-rata tumbuh seiring lg$n$ (kiri); dalam urutan terurut pohon menjadi rantai, dan waktu membangunnya menjadi empat kali lipat pada setiap penggandaan n (kanan, skala logaritmik). 975
Gambar 11.23 Rotasi kanan dan rotasi kiri: tiga pointer berubah, subpohon di satu sisi turun satu tingkat dan di sisi lain naik satu tingkat, dan barisan *inorder* $\alpha, x, \beta, y, \gamma$ tidak berubah. 977
Gambar 11.24 Menyisipkan 1 sampai 7 dalam urutan terurut ke pohon AVL: (a) sesudah 3, simpul 1 berfaktor keseimbangan -2; (b) rotasi kiri di 1 memperbaikinya; (c) sampai (e) tiga rotasi kiri lagi menjaga pohon tetap seimbang, dan hasilnya *perfect*. 978
Gambar 11.25 Pohon **Gambar 11.18** yang diwarnai sebagai pohon *red-black*: akar hitam, simpul merah beranak hitam, dan setiap lintasan akar-ke-NIL memuat dua simpul hitam. 982
Gambar 11.26 Mengubah pohon umum menjadi pohon biner: (a) pohon umum; (b) setiap simpul hanya menyimpan tautan ke anak pertamanya (hijau-biru) dan ke saudara berikutnya (kuning); (c) tautan-tautan yang sama digambar sebagai pohon biner, anak pertama di kiri dan saudara berikutnya di kanan. 983
Gambar 11.27 Dua pohon yang dibangun kembali dari daftar *preorder* dan derajat setiap simpulnya: U V W X Y dengan derajat 2 2 0 0 0 (a, b), serta A B E F C D G dengan derajat 3 2 0 0 0 1 0 (c, d). 984
Gambar 11.28 Pohon papan pengambilan, berkunci waktu siap: kueri rentang 10:00 sampai 12:59 mengunjungi lima simpul (hijau-biru dan kuning) dan melaporkan tiga (kuning); tujuh simpul lainnya tidak pernah disentuh. 1002

Gambar 12.1 Pohon bab ini, dibangun dengan menyisipkan 6, 3, 10, 1, 5, 7, 8, 9 ke dalam *binary search tree* kosong. Tingkat dihitung dari 0 di akar, daun diberi arsiran, dan h adalah tinggi subpohon yang berakar di setiap simpul. 1019

Gambar 12.2 Urutan *depth first* pada Soal 12.1: setiap lencana menunjukkan saat simpul itu dicetak. Seluruh subpohon kiri 6 (lencana 2 sampai 4) dituntaskan sebelum subpohon kanan (lencana 5 sampai 8) dimasuki. 1029

Gambar 12.3 Pemanggilan `printKthLevel(root, 2)`: label setiap simpul adalah k yang diterima pemanggilannya. Simpul-simpul yang menerima $k = 0$ mencetak dirinya lalu berhenti, sehingga 8 dan 9, di bawahnya, tidak pernah dikunjungi. 1034

Gambar 12.4 Nilai yang dikembalikan `sumOfNodes` pada setiap subpohon. Masing-masing adalah nilai simpul itu sendiri ditambah dua nilai yang dikembalikan di bawahnya; akar mengembalikan 49. 1038

Gambar 12.5 Soal 12.4: pohon sebelum dan sesudah `triple(root)`. Bentuknya tidak berubah, dan karena $a < b$ mengakibatkan $3a < 3b$, pohon yang dilipattigakan tetap *binary search tree*. 1042

Gambar 12.6 Soal 12.5: pohon sebelum dan sesudah `mirror(root)`. Setiap simpul telah menukar kedua subpohonnya, sehingga nilai-nilai yang lebih kecil kini tergantung di kanan; hasilnya bukan lagi *binary search tree*. 1045

Gambar 12.7 Soal 12.6: di atas setiap simpul, tinggi subpohon kirinya dan subpohon kanannya. Ketiga simpul merah melanggar aturan, sehingga pohonnya tidak seimbang tingginya. 1049

Gambar 12.8 Tiga kasus Soal 12.7 pada pohon kita: (a) daun 5 cukup dibuang; (b) 10 memiliki satu anak, dan 7 menggantikan tempatnya; (c) akar 6 memiliki dua anak, sehingga penerusnya, 7, disalin ke akar dan 7 yang lama dihapus di bawah. 1056

Gambar 12.9 `findParent(root, 8)` mengikuti lintasan pencarian 8. Di setiap simpul ia melihat satu langkah ke depan; di 7, simpul berikutnya pada lintasan memuat 8, sehingga 7 adalah induknya. 1063

Gambar 12.10 Soal 12.10: delapan simpul yang sama, disambung ulang. Di dalam list, `right` menunjuk ke nilai berikutnya yang lebih besar dan `left` ke nilai sebelumnya yang lebih kecil; garis pandu putus-putus memperlihatkan bahwa urutan list adalah *inorder* pohon itu. 1072

Gambar 12.11 Soal 12.11: setiap simpul tetap di tempatnya dalam memori dan tetap memuat bukunya; hanya kedua panahnya yang bertukar peran, dan `head` serta `tail` bertukar. 1080

Gambar 12.12 Lingkaran Josephus untuk $n = 7$, $k = 3$: lencana merah di samping setiap anggota adalah putaran saat anggota itu keluar. Anggota 4 tidak pernah terhitung ketiga dan mendapatkan bukunya. 1087

Gambar 12.13 Rekursi `choose` untuk dua buku dari A, B, C, D. Setiap pemanggilan entah mengambil atau melewati buku berikutnya; keenam daun kuning adalah keenam pilihan, dan daun kelabu adalah cabang yang ditinggalkan karena buku yang tersisa terlalu sedikit. 1093

Gambar 12.14 Interval yang diterima setiap pemanggilan `inRange`. Setiap pasangan induk–anak berurutan, tetapi nilai 8 berada di luar intervalnya (3,6), sehingga pohon itu bukan *binary search tree*. 1097

Gambar 12.15 Lintasan dari akar ke 5 (hijau toska) dan ke 9 (biru tua) hanya berbagi akar: karena $5 < 6 < 9$, kedua nilai berpisah di 6, yang merupakan leluhur bersama terendahnya. 1101

Gambar 12.16 Soal 12.16: pada awal setiap tingkat, queue memuat tepat tingkat itu; ukurannya adalah lebar tingkat tersebut. Simpul yang diambil dari queue dicetak, dan anak-anaknya dimasukkan untuk tingkat berikutnya. 1104

Gambar 13.1 Tujuh jembatan Königsberg: (a) peta skematis dengan empat daratan A sampai D dan tujuh jembatan berwarna kuning tua; (b) keadaan yang sama sebagai graf, dengan banyaknya jembatan yang menyentuh setiap daratan. 1122

Gambar 13.2 Tiga graf kecil: (a) graf tak berarah dengan derajat setiap simpul tertulis di sampingnya, termasuk simpul terasing E; (b) graf berarah dengan derajat-masuk dan derajat-keluar setiap simpul; (c) graf berbobot, sepotong peta kampus dengan panjang jalan setapak dalam meter. 1123

Gambar 13.3 Peta kampus Pustaka Ceria, contoh berjalan bab ini: sepuluh gedung (perpustakaan berwarna hijau toska) dan sebelas jalan setapak beserta panjangnya dalam meter. Annex kota dan gudang berada di luar kampus dan belum terhubung dengannya. 1127

Gambar 13.4 Peta kampus yang disimpan dengan tiga cara: (a) matriks ketetanggaan, dengan panjang jalan setapak pada sel-sel yang diarsir; (b) list ketetanggaan, satu rantai entri (tetangga, panjang) untuk setiap simpul; (c) list sisi, satu rekaman untuk setiap jalan setapak. 1128

Gambar 13.5 Isi *queue* BFS sesudah setiap langkah **Tabel 13.2**: simpul yang dikeluarkan di kiri, isi *queue* di kanan dengan simpul yang baru dimasukkan berwarna kuning tua. 1140
Gambar 13.6 Peta kampus yang disusun menurut jarak langkah dari LIB. Sisi kuning tua membentuk pohon BFS; sisi putus-putus sudah diperiksa tetapi menuju simpul yang telah ditemukan; ANX dan WHS tidak tercapai. 1140
Gambar 13.7 Penelusuran mendalam pada peta kampus. Setiap simpul diberi label waktu penemuan dan waktu selesainya d/f; sisi kuning tua adalah sisi pohon dan sisi putus-putus adalah sisi balik. 1152
Gambar 13.8 Selang dari penemuan sampai selesai untuk setiap simpul pada DFS di **Gambar 13.7**: setiap dua selang bersarang atau saling lepas, seperti tanda kurung yang berpasangan dengan benar. 1152
Gambar 13.9 Komponen terhubung peta kampus: (a) sebagaimana dibangun, dua komponen (hijau toska dan kuning tua); (b) sesudah jalan *shuttle* PRK-ANX sepanjang 400 meter ditambahkan, satu komponen. 1164
Gambar 13.10 Alur kerja buku baru: (a) DAG ketergantungannya, dengan panah dari setiap tugas ke tugas yang bergantung padanya; (b) DAG yang sama dalam urutan topologis hasil algoritma Kahn, dengan semua panah menunjuk ke kanan. 1171
Gambar 13.11 Alur kerja dengan panah keliru dari shelve ke catalog (putus-putus). Ketika DFS memeriksa panah dari label ke cover, setiap tugas pada lintasan saat ini berwarna abu-abu dan cover salah satunya: sebuah sisi balik, yang membuktikan adanya siklus merah. 1174
Gambar 13.12 Jarak berjalan terpendek dalam meter dari LIB. Sisi kuning tua membentuk pohon lintasan terpendek; sisi merah putus-putus LIB-ENG berada pada rute dengan langkah paling sedikit ke PRK, tetapi tidak pada rute terpendek. 1184
Gambar 13.13 Memori untuk matriks ketetanggaan (satu bit per sel) dan list ketetanggaan (8 bita per entri) tiga graf jalan: peta kampus, jaringan DIMACS AS (tolok ukur 2006), dan OpenStreetMap sedunia (*node* dan ruas per 23 September 2026). Skala logaritmik. 1191
Gambar 13.14 Situs web Profitina Laundry sebagai graf berarah. Angka di atas setiap halaman adalah kedalaman kliknya dari halaman beranda; /faq (kuning tua) berada empat klik di dalam, dan dua halaman bergaris tepi merah adalah halaman yatim yang tidak dicapai lintasan tautan mana pun. 1194
Gambar 14.1 (a) Pohon biner *complete* dengan sepuluh simpul; lingkaran putus-putus menandai tempat kosong di level terakhir, yang terisi dari kiri. (b) dan (c) tidak *complete*: (b) bercelah di level terakhirnya, dan (c) kekurangan simpul di atas level terakhir. 1217
Gambar 14.2 Pohon *complete* yang disimpan level demi level di dalam larik. Setiap simpul membawa indeks berbasis 0 (teal) dan indeks berbasis 1 (kuning tua); busur-busurnya menunjukkan bahwa anak-anak indeks 1 berada di indeks 3 dan 4, dan induknya di indeks 0. 1218
Gambar 14.3 Kunci 25, 30, 40, 55, 65, 70, dan 90 yang ditata (a) sebagai *max-heap*, dengan maksimum di akar, dan (b) sebagai *min-heap*, dengan minimum di akar. 1222
Gambar 14.4 *Sift-up* kunci 80 dari indeks 8. Setiap panel menampilkan pohon dan lariknya berdampingan; kunci yang bergerak berwarna kuning tua dan induk pembandingnya berwarna teal. 1225
Gambar 14.5 *Sift-down* kunci 55 dari akar selama `extractMax`: di setiap level ia dibandingkan dengan kedua anaknya (teal) dan ditukar dengan yang lebih besar, sampai tidak ada anak yang lebih besar. 1227
Gambar 14.6 Daun-daun *max-heap* sembilan kunci adalah indeks 4 sampai 8, paruh kedua larik (teal); minimumnya, 10 (kuning tua), adalah salah satunya. 1230
Gambar 14.7 (a) sampai (c) `increaseKey(6, 75)`: kunci yang dinaikkan naik lewat *sift-up*. (d) sampai (f) `decreaseKey(0, 20)`: kunci yang diturunkan tenggelam lewat *sift-down*, selalu ke arah anak yang lebih besar. 1232
Gambar 14.8 `buildMaxHeap` pada 12, 30, 45, 7, 60, 18, 52, 25, 3, 41: *sift-down* diterapkan pada simpul internal 4, 3, 2, 1, dan 0 dengan urutan itu; kuning tua menandai lintasan tempat setiap kunci tenggelam. 1234
Gambar 14.9 Setiap simpul *heap* 15 simpul diberi label tingginya, yaitu banyak level terbesar yang dapat ditempuh *sift-down* darinya. Menjumlahkan level demi level memberi paling banyak 11 pertukaran untuk seluruh pembangunan, lebih sedikit daripada 15 simpulnya. 1234
Gambar 14.10 *Heap sort* pada *heap* dari **Gambar 14.8**: setelah setiap pertukaran dan *sift-down*, *heap* (pohonnya, akar berwarna kuning tua) menyusut satu dan ekor terurut (teal) bertambah

satu. Panel (e) dan (f) melompati langkah-langkah antara yang tampak pada keluaran **Kode 14.6**. 1254
Gambar 14.11 *Heap sort* atas 5a, 5b, 3 menurut kuncinya: pertukaran pertama memindahkan 5a dari depan ke belakang, melompati 5b, dan keluarannya berakhir dengan 5b sebelum 5a. 1258
Gambar 14.12 (a) 3 teratas dari sebuah aliran data: ketika 33 datang, *min-heap* memuat 27, 31, dan 40; karena $33 > 27$, ia menggantikan akar. Teal menandai 3 teratas di akhir aliran. (b) Menggabung tiga senarai terurut: *min-heap* memuat elemen terdepan setiap senarai (kuning tua), dan akarnya selalu menjadi keluaran berikutnya. 1265
Gambar 14.13 Jalan-jalan pengantaran dengan waktu tempuh dalam menit, dan waktu pengantaran terpendek d dari Library ke setiap gedung. Sisi-sisi teal membentuk pohon lintasan terpendek yang ditemukan *priority queue*. 1268
Gambar 15.1 (a) Tabel beralamat langsung memiliki satu slot untuk setiap kunci yang mungkin, sehingga kunci itu sendiri menjadi indeks. (b) Tabel hash memiliki slot jauh lebih sedikit daripada kunci yang mungkin; fungsi hash h memetakan setiap kunci ke sebuah slot, dan dua kunci berbeda dapat dikirim ke slot yang sama, yaitu sebuah tabrakan. 1284
Gambar 15.2 Peluang sedikitnya satu tabrakan di antara n kunci acak dalam 365 slot: 0,507 pada $n = 23$ dan 0,990 pada $n = 57$. Kurva putus-putus adalah hampiran Persamaan (15.8). 1299
Gambar 15.3 *Separate chaining*: tujuh judul dalam tujuh *bucket* dengan $h(k) = \text{poly31}(k) \bmod 7$. Setiap *bucket* menyimpan kepala sebuah *single linked list*; *bucket* 2, 4, dan 6 masing-masing berisi dua judul yang bertabrakan, dan *bucket* 0, 1, dan 5 kosong. 1300
Gambar 15.4 Delapan kunci yang sama dalam 16 slot dengan (a) *linear probing*, (b) *quadratic probing*, dan (c) *double hashing*. Panah-panahnya adalah *probe* yang dilakukan untuk menyisipkan (dan kelak menemukan) 1051: empat, empat, dan dua. 1316
Gambar 15.5 Menghapus 1019 dari tabel *linear probing*. (a) Sebelumnya, pencarian 1035 memeriksa 11, 12, 13. (b) Dengan nisan di slot 12 pencarian melewatinya dan tetap menemukan 1035. (c) Bila slot 12 sekadar dikosongkan, pencarian berhenti di sana dan keliru melaporkan 1035 tidak ada. 1316
Gambar 15.6 Rata-rata *probe* per pencarian pada tabel berisi 16.384 slot, diukur oleh **Kode 15.9** (penanda) dan diramalkan oleh Persamaan (15.12), Persamaan (15.11), Persamaan (15.20), Persamaan (15.19), dan Persamaan (15.18) (kurva). *Chaining* tetap datar; *linear probing* meledak lebih dulu; *double hashing* mengikuti kurva *uniform hashing* 1336
Gambar 15.7 Panjang rantai untuk 16.384 kunci acak dalam 16.384 *bucket* (hijau toska) dibandingkan ramalan Poisson Persamaan (15.22) dengan $\alpha = 1$ (kuning). Lebih dari sepertiga *bucket* kosong; tidak ada yang berisi lebih dari 7 kunci. 1337
Gambar 15.8 Delapan kode disisipkan dalam urutan B1005, B1001, B1008, B1003, B1002, B1007, B1004, B1006. (a) `std::map` menyimpan pohon merah-hitam, yang penelusuran *in-order*-nya terurut. (b) `std::unordered_map` menyimpan 13 *bucket*; B1002 dan B1004 berbagi *bucket* 9, dan tidak ada urutan yang bertahan. 1343
Gambar 15.9 Tabel `HashMap` dengan *bin* list dan satu *bin* yang telah menjadi pohon. Sebuah *bin* menjadi pohon merah-hitam berisi `TreeNode` ketika mengumpulkan 8 entri dalam tabel berisi sedikitnya 64 *bin*, sehingga pencarian di dalamnya memerlukan $O(\lg \ell)$ alih-alih $O(\ell)$ langkah untuk *bin* berisi ℓ entri. 1349
Gambar 15.10 Tata letak `dict` ringkas dengan tiga butir: tabel `indices` kecil berisi 8 slot satu-*byte* di-*probe* menurut hash, dan setiap slot terpakai menyimpan posisi sebuah entri dalam larik padat `entries`, yang dijaga dalam urutan penyisipan. (Posisi slot yang diperlihatkan hanya ilustrasi; nilai hash sebenarnya berubah dari satu jalannya program ke jalannya program berikutnya.) 1354
Gambar 16.1 *Max-heap* setelah sebelas penyisipan oleh *driver*, digambar sebagai pohon (angka abu-abu kecil adalah indeks larik) dan sebagai larik yang sebenarnya. 1380
Gambar 16.2 `extractMax` dalam tiga langkah: simpan akar, pindahkan daun terakhir ke akar, lalu biarkan ia tenggelam di sepanjang lintasan anak yang lebih besar sampai invarian *max-heap* berlaku kembali. 1381
Gambar 16.3 `increaseKey(2, 8)`: kunci yang dinaikkan hanya dapat menjadi terlalu besar bagi induknya, sehingga ia naik; satu pertukaran memulihkan invarian. 1381
Gambar 16.4 Minimum sebuah *max-heap* selalu berupa daun, dan daun-daun itu adalah separuh akhir larik: indeks $\lfloor n/2 \rfloor$ sampai $n - 1$. 1382

Gambar 16.5 `buildMaxHeap` pada {10, 7, 6, 5, 1, 8, 2}: hanya indeks 2 yang melanggar invarian, dan satu pertukaran memperbaikinya. 1383
Gambar 16.6 Soal 16.2 dengan tangan: tiket-tiket sebagaimana diberikan (a), *map* dari asal ke tujuan (b) dan kebalikannya (c), pencarian satu-satunya kota asal yang tidak pernah dituju (d), dan penelusuran dari kota itu (e). 1400
Gambar 16.7 Kunci-kunci yang sama di `std::map`, *red-black tree* yang dijaga terurut (simpul merah adalah simpul merah dalam skema *red-black*), dan di `std::unordered_map`, tabel hash dengan 13 *bucket* tempat Cianjur dan Denpasar bertabrakan. 1403
Gambar 16.8 Peta cabang ditelusuri melebar dari Pusat: simpul-simpul dalam cincin berjarak sama, pohon BFS dalam garis gelap, dua jalan yang tidak dipakai berwarna pucat, dan Benowo tak terjangkau. 1412
Gambar 16.9 Kesebelas topik dan keenam belas busur "baca sebelum", setiap topik bernomor menurut posisinya dalam urutan yang ditemukan algoritma Kahn; busur merah putus-putus "graf sebelum pointer" menutup siklus Pointer, List, Graf, Pointer. 1421
Gambar 16.10 Waktu tercepat dari Pusat. Jalan-jalan gelap membentuk pohon lintasan terpendek; jalan langsung Pusat–Rungkut yang merah putus-putus (20 menit) kalah oleh jalan memutar melalui Gubeng (11 menit). 1428
Gambar 16.11 Tiga buku yang paling sering dipinjam, ditemukan dengan *min-heap* berukuran 3. Setiap hitungan baru dibandingkan dengan akar, yang terkecil dari tiga yang disimpan; jika lebih besar, akarnya dibuang. 1436
Gambar 16.12 Kedua *heap* setelah kesepuluh waktu tunggu. Akar-akarnya adalah dua nilai tengah daftar terurut, sehingga median dibaca dalam waktu konstan. 1447
Gambar 16.13 Tabel berantai dengan 5 *bucket* (tiga judul bertabrakan di *bucket* 2) dan setelah *rehash* menjadi 11 *bucket*, tempat tidak ada rantai yang lebih panjang daripada dua. 1456
Gambar 16.14 *Linear probing* dengan $m = 11$: lintasan *probe* 59 melalui slot 4 sampai 8. Menghapus 15 meninggalkan *tombstone* di slot 5, yang dilewati pencarian dan dipakai ulang penyisipan. 1465
Gambar 16.15 Memilih struktur: mulailah dari operasi yang paling sering dikerjakan tugas, lalu jawab paling banyak dua pertanyaan lanjutan. Bab yang mengajarkan setiap struktur tertulis di dalam kurung. 1475
Gambar 16.16 Enam struktur pada katalog Pustaka Ceria (C++, mikrodetik per operasi, terbaik dari lima kali jalan, sumbu logaritmik). Tidak ada struktur yang unggul di setiap kolom. 1503

Daftar Tabel

Tabel 1.1 Banyaknya kali lihat dalam kasus terburuk untuk menemukan satu buku di antara n buku, menurut tiga cara menyimpan buku. 3
Tabel 1.2 Keenam belas bab buku ini dan apa yang ditambahkan masing-masing. 5
Tabel 1.3 Sepuluh bahasa pemrograman terpopuler menurut Indeks TIOBE September 2026. 6
Tabel 1.4 Tipe-tipe dasar C++ yang dipakai buku ini, beserta ukurannya pada mesin 64-bit. 15
Tabel 1.5 Array dan array yang dapat tumbuh di C++, Java, dan Python. 55
Tabel 1.6 Biaya operasi-operasi dasar pada array berisi n elemen. 62
Tabel 1.7 Memasang VS Code di setiap sistem operasi. 63
Tabel 1.8 Memasang dan memakai kompilator C++ di setiap sistem operasi. Fedora memakai `sudo dnf install gcc-c++ gdb` dan Arch `sudo pacman -S gcc gdb` pada langkah 1. 64
Tabel 1.9 Memasang JDK 25 di setiap sistem operasi. 65
Tabel 1.10 Memasang Python di setiap sistem operasi. 65
Tabel 1.11 Ekstensi VS Code yang dipakai buku ini. 66
Tabel 1.12 Perintah-perintah pemeriksaan, apa yang semestinya dicetak masing-masing, dan perbaikan yang lazim bila tidak. 66
Tabel 2.1 Rak komputer Pustaka Ceria, terurut menurut kode. Pencarian sekuensial juga akan menjumpai buku-buku ini menurut urutan kedatangannya: 072, 025, 096, 041, 088, 034, 063, 056. 85

Tabel 2.2 Dua konvensi interval. Setiap baris harus diambil dari kolom yang sama; program yang mengambil syarat gelung dari satu kolom dan pembaruan dari kolom lain adalah program yang salah. 111
Tabel 2.3 Biaya terukur kelima pencarian pada katalog 10.000 buku: rata-rata pemeriksaan per pencarian, dan nanodetik per pencarian dirata-ratakan atas seluruh 1.905 kueri (terbaik dari sembilan pengukuran). Angka desimal ditulis dengan titik, sama seperti keluaran program. 156
Tabel 2.4 Pencarian dengan pustaka standar C++, Java, dan Python. 167
Tabel 2.5 Metode-metode pencarian di bab ini, apa yang diperlukan masing-masing, dan berapa biayanya. 173
Tabel 3.1 Pertumbuhan $n\lg n$ dan $n2/2$. Pada satu miliar butir, komputer yang mengerjakan 109 langkah per detik memerlukan 30 detik untuk $n\lg n$ langkah dan 16 tahun untuk $n2/2$ langkah. 203
Tabel 3.2 Waktu jalan (setiap isian adalah Θ dari ungkapannya, kecuali rata-rata *Shell sort*, yang merupakan taksiran empiris), memori tambahan, dan ketiga sifat setiap pengurutan di bab ini. 269
Tabel 3.3 Kerja setiap algoritma pada $A = [22,10,15,3,8,2]$, sebagaimana dicetak program-program pada Subbab 3.3 sampai 3.11. 269
Tabel 3.4 Pengukuran historis: mengurutkan 10.000 bilangan pada Pentium II 450 MHz (akhir 1990-an). Waktu menurun *selection sort* mengulang waktu *bubble sort* dan hampir pasti merupakan salah salin; waktu *selection sort* tidak bergantung pada urutan masukan. 278
Tabel 3.5 Eksperimen yang sama pada tahun 2026 (dari keluaran **Kode 3.34**). Waktu absolut turun dengan faktor 10 sampai 60; hubungan antaralgoritmanya tidak berubah. 278
Tabel 3.6 Pengurutan standar C++, Java, dan Python per September 2026, algoritma di dalamnya, dan jaminannya. 291
Tabel 4.1 Kesebelas soal dalam bab ini dan perkakas dari Bab 1 sampai Bab 3 yang paling banyak dilatih oleh masing-masing soal. 314
Tabel 4.2 Perkiraan kasar banyak langkah untuk laju pertumbuhan dari Bab 2 dan Bab 3; dengan sekitar 108 sampai 109 langkah sederhana per detik, 1012 langkah memakan waktu bermenit-menit dan 1016 langkah memakan waktu bertahun-tahun. 316
Tabel 4.3 Kedua kunci setiap string pada Soal 4.4, dihitung dengan tangan. 338
Tabel 4.4 Hitungan Soal 4.8 untuk larik 5, 6, 8, 1, 3, 25, 10, yang memiliki 7 inversi. 368
Tabel 4.5 Kesebelas soal dalam bab ini, gagasan yang diajarkan masing-masing, biayanya, dan jebakan yang diperagakan penyelesaian keliru yang menggodanya. 391
Tabel 5.1 Operasi ADT stack, prasyaratnya, dan biayanya pada implementasi larik di Subbab 5.4. 406
Tabel 5.2 Ekspresi yang sama dalam notasi infiks, prefiks, dan postfiks. Postfiks tidak memerlukan tanda kurung, dan operatornya muncul persis dalam urutan penerapannya. 426
Tabel 5.3 Presedensi dan asosiativitas operator yang dipakai kalkulator. 426
Tabel 5.4 Mengubah `a+b*c-d` menjadi postfiks, satu token per baris. 428
Tabel 5.5 Operasi ADT queue dan biayanya pada implementasi larik sirkular di Subbab 5.13. 438
Tabel 5.6 Dua cara membedakan queue sirkular yang penuh dari yang kosong. 451
Tabel 5.7 Tiga cara mengimplementasikan priority queue. *Heap* pada Bab 14 adalah satu-satunya yang membuat kedua operasi cepat. 457
Tabel 5.8 Operasi yang sama dalam ketiga pustaka standar. 462
Tabel 5.9 Biaya setiap struktur dalam bab ini. 463
Tabel 6.1 Tiga makna `*` dan `&` dalam C++. Konteks selalu menentukan makna mana yang dimaksud. 478
Tabel 6.2 Variabel biasa dan pointer, berdampingan. 478
Tabel 6.3 Operasi yang didefinisikan C++ pada pointer `p` bertipe `T*`. 484
Tabel 6.4 Kelemahan keamanan memori dalam 2025 CWE Top 25 Most Dangerous Software Weaknesses, yang diterbitkan MITRE pada Desember 2025 [4]. 498
Tabel 6.5 Perbandingan AddressSanitizer dan Valgrind, per September 2026 [5], [6], [7], [8]. 503
Tabel 6.6 Perbandingan pointer dan referensi. 507
Tabel 6.7 Memilih cara fungsi C++ menerima argumennya. 513
Tabel 6.8 Empat kombinasi `const` dengan pointer. 514
Tabel 6.9 *Raw pointer*, referensi, dan *smart pointer*, serta kapan memakai masing-masing. 524
Tabel 6.10 Pointer, referensi, dan pengiriman parameter dalam C++, Java, dan Python. 533
Tabel 6.11 Waktu eksekusi operasi-operasi dalam bab ini. 534

Tabel 7.1 Larik dan *single linked list* dibandingkan. 553
Tabel 7.2 Waktu jalan setiap operasi list ber-*head* dengan n simpul. 569
Tabel 7.3 Ongkos operasi-operasi Subbab 7.6 pada list berisi n simpul. 586
Tabel 7.4 Varian-varian *single linked list* dibandingkan, untuk list berisi n simpul. 597
Tabel 7.5 *Linked list* pada pustaka baku C++, Java, dan Python. 619
Tabel 8.1 Dua belas soal dalam bab ini, subbab yang menyelesaikan masing-masing, dan bab yang menyumbangkan gagasan utamanya. 636
Tabel 8.2 Metode tujuh langkah untuk soal tentang stack, queue, dan linked list. Langkah 1 dan 2 memahami soal, Langkah 3 sampai 6 merancang pergerakan pointer, dan Langkah 7 memeriksanya. 638
Tabel 8.3 Gagasan yang sama dalam tiga bahasa buku ini. Referensi Java dan nama Python berperilaku seperti pointer C++ yang hanya dapat diikuti, tidak pernah dinaikkan, dan yang sasarannya dibebaskan secara otomatis. 639
Tabel 8.4 Penelusuran Soal 8.2 pada contoh masukan. Setiap baris mencetak huruf simpul saat ini, lalu bergerak menurut petunjuk simpul itu sendiri. 650
Tabel 8.5 Sepuluh penyisipan pada contoh masukan Soal 8.3. Mencetak `evens` lalu `odds` menghasilkan keluaran yang diharapkan. 661
Tabel 8.6 Mengubah `a+b*c-d` ke postfiks dengan aturan-aturan Soal 8.8. Stack menampung operator yang menunggu operand kanannya lengkap. 696
Tabel 8.7 Apa yang dapat diubah sebuah fungsi di setiap bahasa. Inilah sebabnya versi Java dan Python pada Soal 8.3 sampai 8.5 mengembalikan simpul pertama yang baru, sedangkan versi C++ menerima `Node *&head`. 709
Tabel 8.8 Kedua belas soal dalam bab ini: struktur yang dipakai masing-masing, biayanya, dan masukan yang membuat penyelesaian kelirunya gagal. n adalah banyaknya simpul, L panjang dalam karakter. 731
Tabel 9.1 Waktu jalan dengan pointer `head` saja, untuk list berisi n simpul: pointer kedua mengubah penyusuran mundur tetapi belum mengubah ujung belakang. 765
Tabel 9.2 Ongkos dengan pointer *head* dan *tail*, untuk list berisi n simpul. *Double linked list* mengubah setiap operasi pada posisi yang diketahui menjadi $O(1)$, dengan harga pekerjaan pointer dua kali lipat. 781
Tabel 9.3 Uji kasus khusus dalam dua operasi dasar, menurut representasinya. Hanya *sentinel* yang menghapus semuanya. 795
Tabel 9.4 Larik dinamis, *single linked list*, dan *double linked list* dibandingkan, untuk n elemen. 803
Tabel 9.5 Struktur *double linked* dalam pustaka standar C++, Java, dan Python. 820
Tabel 10.1 Tujuh cara menghitung $F(n)$, dengan setiap operasi pada *word* mesin dihitung satu langkah. 861
Tabel 10.2 Rekurens yang lazim, penyelesaiannya, dan algoritma dalam buku ini yang menghasilkannya. 884
Tabel 10.3 Batas kedalaman rekursi di C++, Java, dan Python, sebagaimana diukur pada mesin buku ini pada September 2026. 895
Tabel 10.4 Rekursi atau iterasi: keputusan untuk setiap algoritma bab ini. 896
Tabel 11.1 Kosakata pohon, dievaluasi pada pohon **Gambar 11.2**. 920
Tabel 11.2 Empat bentuk pohon biner, beserta nama-nama lain yang dipakai untuknya. 927
Tabel 11.3 Ongkos operasi-operasi *binary search tree* dengan n simpul dan tinggi h. 971
Tabel 11.4 *Binary search tree* biasa, AVL, dan *red-black* dibandingkan. 982
Tabel 11.5 Himpunan dan *map* terurut dalam pustaka baku C++, Java, dan Python, beserta alternatif berbasis *hash*-nya. 996
Tabel 12.1 Enam belas soal bab ini, subbab yang menyelesaikan masing-masing, dan bab yang menyumbangkan gagasan utamanya. 1015
Tabel 12.2 Metode tujuh langkah untuk soal rekursif pada list atau pohon. Langkah 1 dan 2 memastikan jawabannya, Langkah 3 sampai 5 merancang rekursinya, dan Langkah 6 dan 7 memeriksanya. 1017
Tabel 12.3 Gagasan-gagasan pohon bab ini dalam tiga bahasa buku ini. Hanya C++ yang memungkinkan fungsi menerima alamat pointer milik pemanggil; Java dan Python

mengembalikan akar yang baru, atau menyimpan variabel yang harus diubah di dalam sebuah objek. 1018
Tabel 12.4 Tiga kasus penghapusan simpul dari *binary search tree*. Kasus 3 direduksi menjadi kasus 1 atau kasus 2, karena penerus tidak pernah memiliki anak kiri. 1056
Tabel 12.5 Keenam belas soal bab ini: struktur yang dipakai masing-masing, biayanya, dan masukan yang membuat penyelesaian keliru yang menggoda gagal. n adalah banyaknya simpul dan h tinggi pohon. 1108
Tabel 13.1 Ongkos ketiga representasi graf, untuk graf dengan V simpul dan E sisi. 1130
Tabel 13.2 Penelusuran melebar dari LIB pada peta kampus, langkah demi langkah. 1139
Tabel 13.3 Algoritma Kahn pada alur kerja buku baru. 1172
Tabel 13.4 Algoritma Dijkstra dari LIB pada peta kampus. Isi *priority queue* didaftar menurut jarak yang menaik. 1183
Tabel 13.5 Graf-graf nyata dan memori yang diperlukan dua representasi utamanya. Banyaknya ruas OpenStreetMap ditaksir sebagai banyaknya *node* dikurangi banyaknya *way*. 1192
Tabel 13.6 Algoritma-algoritma graf dalam bab ini, beserta struktur yang diandalkan masing-masing dan ongkosnya pada list ketetanggaan. 1193
Tabel 14.1 Operasi-operasi kelas `MaxHeap` dalam bab ini beserta ongkosnya. 1215
Tabel 14.2 Empat cara mengimplementasikan *max-priority queue* berisi n kunci. Hanya *binary heap* yang cepat pada setiap operasi tanpa memori tambahan. 1216
Tabel 14.3 Aritmetika indeks pohon biner *complete* berisi n simpul yang disimpan di dalam larik. 1219
Tabel 14.4 Antarmuka publik kelas `MaxHeap` dalam C++, Java, dan Python. 1239
Tabel 14.5 *Heap sort* dibandingkan dengan pengurutan perbandingan $O(n \lg n)$ lainnya dari Bab 3. 1257
Tabel 14.6 *Priority queue* dalam ketiga pustaka standar. Ketiganya adalah *binary heap* di dalam larik. 1263
Tabel 15.1 Mencari sebuah kunci: dari pencarian sekuensial hingga hashing. 1285
Tabel 15.2 Berapa banyak kunci acak yang membuat tabrakan mungkin terjadi, dihitung oleh **Kode 15.2** dari Persamaan (15.7); kolom tengah adalah hampiran Persamaan (15.9). 1298
Tabel 15.3 *Separate chaining* dan pengalamatan terbuka dibandingkan. 1328
Tabel 15.4 Ongkos operasi tabel hash dengan faktor muat Persamaan (15.17) yang dijaga di bawah suatu konstanta; bandingkan dengan $\Theta(\lg n)$ pohon pencarian seimbang untuk semuanya. 1338
Tabel 15.5 `std::map` dan `std::unordered_map` dibandingkan. 1348
Tabel 15.6 Tabel hash dalam pustaka baku C++, Java, dan Python (GCC 13, JDK 25, CPython 3.14). 1356
Tabel 16.1 Sembilan soal dalam bab ini, subbab yang menyelesaikan masing-masing, dan bab yang memasok gagasan utamanya. 1375
Tabel 16.2 Metode sembilan langkah untuk soal tentang sebuah struktur data. Langkah 1 sampai 3 memahami soal, Langkah 4 sampai 6 merancang operasi, Langkah 7 memeriksa bahwa rancangan itu sepadan, dan Langkah 8 serta 9 memeriksa programnya. 1377
Tabel 16.3 Struktur-struktur Bab 13 sampai 15 dalam tiga bahasa buku ini. *Priority queue* C++ secara bawaan adalah *max-heap*, sedangkan milik Java dan Python adalah *min-heap*; dan di C++ kata `map` berarti pohon pencarian, bukan tabel hash. 1378
Tabel 16.4 Memilih struktur data menurut operasi yang paling sering dikerjakan tugas: seluruh buku dalam satu halaman. n adalah banyaknya butir, r banyaknya hasil kueri rentang. 1474
Tabel 16.5 Mikrodetik per pencarian dan per kueri rentang pada katalog 10.000 buku, dari ketiga jalannya program di atas (angka ditulis seperti dicetak program, dengan titik desimal). Struktur terbaik untuk pencarian termasuk yang terburuk untuk rentang. 1503
Tabel 16.6 Kesembilan soal bab ini: struktur yang dipakai masing-masing, ongkosnya, dan masukan tempat penyelesaian kelirunya gagal. 1505

Daftar Kode

Kode 1.1 `memory_basics.cpp`: ukuran tipe-tipe dasar, byte-byte sebuah `int`, dan sebuah alamat di *stack* serta di *heap*. 9

Kode 1.2 `MemoryBasics.java`: ukuran tipe Java yang tetap dan byte-byte sebuah `int` dalam urutan milik mesin itu sendiri. 11
Kode 1.3 `memory_basics.py`: ukuran tipe C dilihat lewat `struct`, byte-byte 2026, dan ukuran sebenarnya objek-objek Python. 12
Kode 1.4 `io_sum.cpp`: membaca sebuah cacah dan bilangan sebanyak itu dengan `cin`, lalu meringkasnya. 15
Kode 1.5 `IoSum.java`: ringkasan yang sama dengan `Scanner`. 16
Kode 1.6 `io_sum.py`: ringkasan yang sama, dengan membaca semua token dari masukan standar sekaligus. 17
Kode 1.7 `late_fee.cpp`: konstanta, sebuah struct, fungsi dengan `if` dan `switch`, array berisi struct, dan perulangan `for` berbasis rentang dengan `continue`. 19
Kode 1.8 `LateFee.java`: biaya keterlambatan dengan `record` Java dan `switch`. 21
Kode 1.9 `late_fee.py`: biaya keterlambatan dengan `dataclass` dan pernyataan `match`. 22
Kode 1.10 `address_of.cpp`: `&` memberikan alamat, `*` mengikutinya, dan fungsi yang menerima alamat dapat mengubah variabel pemanggil. 24
Kode 1.11 `AddressOf.java`: Java menyalin nilai dasar ke dalam metode, tetapi berbagi objek lewat referensi. 25
Kode 1.12 `address_of.py`: nama-nama Python merujuk ke objek; `id()` memperlihatkan bahwa dua nama merujuk ke list yang sama. 25
Kode 1.13 `book.h`: rekaman `Book` milik Pustaka Ceria, dengan array karakter berukuran tetap, serta fungsi pembantu untuk mengisi dan mencetaknya. 28
Kode 1.14 `catalog_1d.cpp`: katalog Pustaka Ceria sebagai array berisi delapan `Book`, beserta offset byte setiap elemen. 28
Kode 1.15 `Book.java`: rekaman `Book` di Java, dengan field `String`. 30
Kode 1.16 `Catalog1D.java`: katalog sebagai array Java berisi referensi `Book`. 31
Kode 1.17 `book.py`: rekaman `Book` di Python, sebagai *dataclass*. 31
Kode 1.18 `catalog_1d.py`: katalog sebagai list Python. 32
Kode 1.19 `bounds.cpp`: pengindeksan di dalam batas, dan pengakses berpemeriksa `at()` yang menolak indeks di luar jangkauan. 33
Kode 1.20 `Bounds.java`: Java memeriksa setiap indeks, untuk array maupun `ArrayList`. 34
Kode 1.21 `bounds.py`: Python memeriksa setiap indeks, dan juga menerima indeks negatif. 34
Kode 1.22 `array_ops.cpp`: jumlah, posisi maksimum, pencarian linear, pencacahan, dan pembalikan array di tempat. 35
Kode 1.23 `ArrayOps.java`: kelima penelusuran di Java. 38
Kode 1.24 `array_ops.py`: kelima penelusuran di Python. 39
Kode 1.25 `shelves_2d.cpp`: array dua dimensi berisi `Book`, dan rumus row-major yang diperiksa untuk setiap elemen. 40
Kode 1.26 `Shelves2D.java`: array Java berisi array baris. 42
Kode 1.27 `shelves_2d.py`: list Python berisi list baris. 43
Kode 1.28 `loan_table.cpp`: tabel peminjaman 3 × 4 dengan total baris, total kolom, dan total keseluruhan. 44
Kode 1.29 `LoanTable.java`: tabel peminjaman di Java. 46
Kode 1.30 `loan_table.py`: tabel peminjaman di Python. 46
Kode 1.31 `aliasing_rows.py`: `[[0] * 4] * 3` menghasilkan tiga referensi ke satu baris; *list comprehension* menghasilkan tiga baris. 47
Kode 1.32 `vector_growth.cpp`: `std::array` berukuran tetap, dan `std::vector` yang menggandakan kapasitasnya ketika tumbuh. 48
Kode 1.33 `ListGrowth.java`: array Java berpanjang tetap, dan langkah-langkah tumbuh sebuah `ArrayList` dari kosong sampai 100 elemen. 50
Kode 1.34 `list_growth.py`: tuple tidak dapat tumbuh; list tumbuh bertahap, menyisakan ruang untuk `append` berikutnya. 51
Kode 1.35 `compact_array.py`: 1.000 bilangan dalam list berisi referensi dan dalam `array('i')` yang padat. 52

Kode 1.36 `compact_array.cpp`: 1.000 `int32_t` dalam array mentah dan dalam vector. 53
Kode 1.37 `CompactArray.java`: `int[]` menyimpan nilai, `Integer[]` menyimpan referensi. 54
Kode 1.38 `insert_shift.cpp`: penyisipan dan penghapusan dengan menggeser, beserta hitungan geserannya. 57
Kode 1.39 `InsertShift.java`: penyisipan dan penghapusan dengan menggeser, di Java. 58
Kode 1.40 `insert_shift.py`: penyisipan dan penghapusan dengan menggeser, di Python. 60
Kode 1.41 `hello.cpp`: program pertama; ia membuktikan bahwa kompilator, editor, dan terminal bekerja bersama. 67
Kode 1.42 `Hello.java`: program pertama di Java. 67
Kode 1.43 `hello.py`: program pertama di Python. 68
Kode 1.44 `sol_address3d.cpp`: Persamaan (1.9) diperiksa terhadap alamat sebenarnya dari ke-24 elemen. 74
Kode 1.45 `SolAddress3D.java`: rumus posisi tiga dimensi diperiksa di Java. 75
Kode 1.46 `sol_address3d.py`: rumus posisi tiga dimensi diperiksa di Python. 75
Kode 1.47 `sol_rotate.cpp`: rotasi dengan tiga pembalikan. 76
Kode 1.48 `SolRotate.java`: rotasi dengan tiga pembalikan, di Java. 77
Kode 1.49 `sol_rotate.py`: rotasi dengan tiga pembalikan, di Python. 78
Kode 1.50 `sol_transpose.cpp`: mentranspos tabel peminjaman. 79
Kode 1.51 `SolTranspose.java`: mentranspos tabel peminjaman, di Java. 79
Kode 1.52 `sol_transpose.py`: mentranspos tabel peminjaman, di Python. 79
Kode 1.53 `sol_padding.cpp`: offset dan ukuran `Book` yang mempunyai field `int` 81
Kode 2.1 `sequential_search.cpp`: pencarian sekuensial dengan penanda, dengan berhenti awal, untuk menghitung, dan pada rekaman. 86
Kode 2.2 `SequentialSearch.java`: pencarian yang sama dalam Java. 89
Kode 2.3 `sequential_search.py`: pencarian yang sama dalam Python. 92
Kode 2.4 `sentinel_search.cpp`: gelung biasa dan gelung sentinel, dengan setiap uji gelung dihitung 97
Kode 2.5 `SentinelSearch.java`: perbandingan yang sama dalam Java. 98
Kode 2.6 `sentinel_search.py`: perbandingan yang sama dalam Python. 99
Kode 2.7 `binary_search.cpp`: pencarian biner iteratif dengan jejak, dan bentuk rekursifnya. 105
Kode 2.8 `BinarySearch.java`: program yang sama dalam Java. 106
Kode 2.9 `binary_search.py`: program yang sama dalam Python. 108
Kode 2.10 `bounds.cpp`: *lower bound*, *upper bound*, pencarian biner yang tidak berhenti, dan titik tengah yang meluap. 113
Kode 2.11 `Bounds.java`: program yang sama dalam Java, tempat luapan terjadi pada `int` sungguhan. 115
Kode 2.12 `bounds.py`: program yang sama dalam Python. 117
Kode 2.13 `interpolation_search.cpp`: pencarian interpolasi pada rak komputer, mencetak setiap taksiran. 121
Kode 2.14 `InterpolationSearch.java`: program yang sama dalam Java. `Locale.ROOT` membuat tanda desimalnya berupa titik di komputer mana pun. 122
Kode 2.15 `interpolation_search.py`: program yang sama dalam Python. 123
Kode 2.16 `jump_search.cpp`: pencarian lompat pada enam belas nomor panggil, dan kasus terburuk untuk beberapa n. 127
Kode 2.17 `JumpSearch.java`: program yang sama dalam Java. 129
Kode 2.18 `jump_search.py`: program yang sama dalam Python. 130
Kode 2.19 `search_2d.cpp`: alamat pada larik dua dimensi, penelusuran, metode tangga, dan pencarian biner dalam urutan baris-mayor. 133
Kode 2.20 `Search2D.java`: program yang sama dalam Java. 136
Kode 2.21 `search_2d.py`: program yang sama dalam Python, dengan daftar berisi daftar sebagai kisinya. 139
Kode 2.22 `measure_catalog.cpp`: lima pencarian pada katalog sepuluh ribu buku sungguhan, dengan pemeriksaan dihitung dan waktu diukur. 142

Kode 2.23 `MeasureCatalog.java`: percobaan yang sama dalam Java. 147
Kode 2.24 `measure_catalog.py`: percobaan yang sama dalam Python. 151
Kode 2.25 `scaling.cpp`: rata-rata pemeriksaan ketika n tumbuh, pada kunci seragam yang dibangkitkan dan pada kunci kubik. 158
Kode 2.26 `Scaling.java`: percobaan yang sama dalam Java. Fungsi bantu `fixed` membulatkan nilai biner eksaknya, seperti yang dilakukan C dan Python, sehingga ketiga tabel sama sampai ke angka terakhir. 160
Kode 2.27 `scaling.py`: percobaan yang sama dalam Python. 163
Kode 2.28 `library_search.cpp`: `std::find`, `std::binary_search`, `std::lower_bound`, `std::upper_bound`, dan `std::equal_range`, termasuk pencarian rekaman menurut kode. 168
Kode 2.29 `LibrarySearch.java`: `indexOf`, `Arrays.binarySearch`, dan `Collections.binarySearch`, termasuk nilai kembalian negatif dan pencarian rekaman dengan pembanding. 169
Kode 2.30 `library_search.py`: `in`, `list.index`, dan modul `bisect`, termasuk pencarian rekaman dengan `key=`. 171
Kode 2.31 `laundry_orders.cpp`: pesanan Profitina Laundry dicari menurut rentang waktu, nomor, waktu terdekat, dan nama. 175
Kode 2.32 `LaundryOrders.java`: program yang sama dalam Java. 177
Kode 2.33 `laundry_orders.py`: program yang sama dalam Python. 179
Kode 2.34 `fibonacci_exponential.cpp`: pencarian Fibonacci dan pencarian eksponensial. 187
Kode 2.35 `FibonacciExponential.java`: kedua pencarian yang sama dalam Java. 189
Kode 2.36 `fibonacci_exponential.py`: kedua pencarian yang sama dalam Python. 191
Kode 3.1 `swap_demo.cpp`: menukar dua `int` melalui pointer, dan menukar dua elemen larik. 199
Kode 3.2 `SwapDemo.java`: metode tidak dapat menukar variabel `int` milik pemanggilnya, tetapi dapat menukar dua slot larik. 200
Kode 3.3 `swap_demo.py`: menukar dengan penugasan tuple, untuk dua nama dan untuk dua elemen list. 201
Kode 3.4 `bubble_sort.cpp`: *bubble sort* dengan keluar dini, menaik atau menurun, dengan hitungan perbandingan dan pertukaran. 205
Kode 3.5 `BubbleSort.java`: *bubble sort* yang sama dalam Java. 207
Kode 3.6 `bubble_sort.py`: *bubble sort* yang sama dalam Python. 207
Kode 3.7 `exchange_sort.cpp`: *exchange sort*, mencetak setiap pertukaran dan larik setelah setiap putaran. 211
Kode 3.8 `ExchangeSort.java`: *exchange sort* dalam Java. 212
Kode 3.9 `exchange_sort.py`: *exchange sort* dalam Python. 213
Kode 3.10 `selection_sort.cpp`: *selection sort*, melaporkan di mana setiap putaran menemukan minimumnya. 215
Kode 3.11 `SelectionSort.java`: *selection sort* dalam Java. 216
Kode 3.12 `selection_sort.py`: *selection sort* dalam Python. 217
Kode 3.13 `insertion_sort.cpp`: *insertion sort* dengan hitungan perbandingan dan pergeseran, pada masukan acak, terurut, dan terbalik. 220
Kode 3.14 `InsertionSort.java`: *insertion sort* dalam Java. 220
Kode 3.15 `insertion_sort.py`: *insertion sort* dalam Python. 221
Kode 3.16 `shell_sort.cpp`: *Shell sort* dengan tiga barisan celah, dibandingkan pada 100.000 bilangan acak. 225
Kode 3.17 `ShellSort.java`: *Shell sort* dalam Java. `>>>` adalah geseran tanpa tanda milik Java, yang diperlukan pembangkitnya. 226
Kode 3.18 `shell_sort.py`: *Shell sort* dalam Python. Bilangan bulat Python tidak pernah meluap, sehingga pembangkitnya mengambil sisa pembagian dengan 2 pangkat 64 secara manual. 228
Kode 3.19 `merge_sort.cpp`: *merge sort* atas-bawah dengan satu penyangga bersama, mencetak setiap penggabungan. 232
Kode 3.20 `MergeSort.java`: *merge sort* dalam Java. 233
Kode 3.21 `merge_sort.py`: *merge sort* dalam Python. 234

Kode 3.22 `quick_sort.cpp`: *quicksort* dengan partisi Lomuto dan Hoare, serta eksperimen dengan tiga aturan pivot. 239
Kode 3.23 `QuickSort.java`: *quicksort* dalam Java; eksperimen pivot berjalan pada *thread* dengan *stack* yang lebih besar. 242
Kode 3.24 `quick_sort.py`: *quicksort* dalam Python. 246
Kode 3.25 `heap_sort.cpp`: *heap sort* pada larik berjalan, dan pada larik yang harus dijadikan *heap* lebih dahulu. 252
Kode 3.26 `HeapSort.java`: *heap sort* dalam Java. 254
Kode 3.27 `heap_sort.py`: *heap sort* dalam Python. 256
Kode 3.28 `counting_radix.cpp`: *counting sort* stabil dengan parameter fungsi kunci, dan *radix sort* LSD yang dibangun di atasnya. 260
Kode 3.29 `CountingRadix.java`: *counting sort* dan *radix sort* dalam Java, dengan `IntUnaryOperator` sebagai fungsi kunci. 261
Kode 3.30 `counting_radix.py`: *counting sort* dan *radix sort* dalam Python, dengan sebuah `lambda` sebagai fungsi kunci. 262
Kode 3.31 `stability_demo.cpp`: katalog diurutkan menurut kategori dengan pengurutan stabil dan tidak stabil. Pembandingnya diteruskan sebagai pointer fungsi. 264
Kode 3.32 `StabilityDemo.java`: peragaan yang sama dalam Java, dengan `record` untuk buku dan `Comparator` untuk kuncinya. 265
Kode 3.33 `stability_demo.py`: peragaan yang sama dalam Python, dengan `dataclass` dan fungsi kunci. 266
Kode 3.34 `sort_benchmark.cpp`: setiap pengurutan dalam bab ini pada seribu sampai sejuta bilangan bulat acak, dengan hitungan dan waktu. 270
Kode 3.35 `SortBenchmark.java`: tolok ukur dalam Java, termasuk `Arrays.sort` pada tipe primitif dan pada objek. 279
Kode 3.36 `sort_benchmark.py`: tolok ukur dalam Python; pengurutan kuadratik berhenti pada sepuluh ribu elemen. 285
Kode 3.37 `library_sorts.cpp`: `std::sort` dengan dan tanpa komparator, `std::partial_sort`, dan `std::stable_sort` dengan *lambda*. 292
Kode 3.38 `LibrarySorts.java`: `Arrays.sort` pada `int[]` dan pada `Integer[]` dengan `Collections.reverseOrder()`, serta `List.sort` dengan `Comparator.comparing`. 293
Kode 3.39 `library_sorts.py`: `list.sort`, `sorted` dengan kunci, dan `heapq.nlargest`. 294
Kode 3.40 `laundry_orders.cpp`: antrean cuci Profitina Laundry dengan `std::stable_sort`, dan daftar pengambilannya dengan `std::sort`. 297
Kode 3.41 `LaundryOrders.java`: antrean yang sama dengan `List.sort`, yang stabil, dan `Comparator.comparingDouble(...).reversed()`. 297
Kode 3.42 `laundry_orders.py`: antrean yang sama dengan `sorted(..., reverse=True)`. 298
Kode 3.43 `cocktail_sort.cpp`: *cocktail shaker sort*, penyelesaian Soal 3.4. 304
Kode 3.44 `CocktailSort.java`: *cocktail shaker sort* dalam Java. 304
Kode 3.45 `cocktail_sort.py`: *cocktail shaker sort* dalam Python. 305
Kode 3.46 `count_inversions.cpp`: inversi dihitung dengan *merge sort*, diperiksa terhadap hitungan kuadratik menurut definisi. Penyelesaian Soal 3.14. 307
Kode 3.47 `CountInversions.java`: penghitungan inversi dalam Java. 308
Kode 3.48 `count_inversions.py`: penghitungan inversi dalam Python. 309
Kode 4.1 `sort_words.cpp`: insertion sort dengan perbandingan menurut urutan kamus. 320
Kode 4.2 `SortWords.java`: insertion sort yang sama dalam Java. 322
Kode 4.3 `sort_words.py`: insertion sort yang sama dalam Python. 323
Kode 4.4 `wrong_sort_words.cpp`: insertion sort yang sama dengan `<` biasa pada string. 324
Kode 4.5 `first_repeat.cpp`: dua lintasan atas string dengan 256 pencacah. 328
Kode 4.6 `FirstRepeat.java`: dua lintasan yang sama dalam Java, dengan satu pencacah per `char` Java. 329
Kode 4.7 `first_repeat.py`: dua lintasan yang sama dalam Python, dengan kamus pencacah. 329
Kode 4.8 `wrong_first_repeat.cpp`: berhenti pada karakter pertama yang sudah pernah terlihat. 330

Kode 4.9 `search_money.cpp`: pencarian sekuensial pada larik rekaman `Person`. 333
Kode 4.10 `SearchMoney.java`: pencarian yang sama dengan kelas kecil `Person` 335
Kode 4.11 `search_money.py`: pencarian yang sama atas list berisi pasangan (nama, uang). 336
Kode 4.12 `wrong_search_money.cpp`: pencarian biner pada data yang tidak terurut menurut nama 336
Kode 4.13 `importance_sort.cpp`: aturan perbandingan dua kunci di dalam insertion sort yang stabil. 339
Kode 4.14 `ImportanceSort.java`: aturan dan pengurutan yang sama dalam Java. 341
Kode 4.15 `importance_sort.py`: aturan dan pengurutan yang sama dalam Python. 342
Kode 4.16 `wrong_importance_sort.py`: dua pengurutan stabil yang dijalankan dengan urutan keliru. 343
Kode 4.17 `array_address.cpp`: rumus alamat, diperiksa terhadap jarak di memori nyata. 346
Kode 4.18 `ArrayAddress.java`: rumus alamat dalam Java. 348
Kode 4.19 `array_address.py`: rumus alamat dalam Python. 348
Kode 4.20 `wrong_array_address.py`: melupakan ukuran elemen, dan mengalikan dengan banyaknya baris. 349
Kode 4.21 `count_range.cpp`: batas bawah, batas atas, dan selisih keduanya. 351
Kode 4.22 `CountRange.java`: kedua pencarian batas dalam Java. 353
Kode 4.23 `count_range.py`: kedua pencarian batas dalam Python. 353
Kode 4.24 `WrongCountRange.java`: rentang pencarian dimulai sebagai $[0, n-1]$, sehingga jawaban n tidak pernah dapat dikembalikan. 354
Kode 4.25 `interpolation_search.cpp`: pencarian interpolasi dengan jejak yang dicetak, berdampingan dengan pencarian biner yang menghitung pemeriksaannya. 357
Kode 4.26 `InterpolationSearch.java`: pencarian yang sama dalam Java. 359
Kode 4.27 `interpolation_search.py`: pencarian yang sama dalam Python, dengan jejak yang dikumpulkan dalam sebuah string. 361
Kode 4.28 `wrong_interpolation.py`: pencarian interpolasi tanpa penjaga rentang. 363
Kode 4.29 `trace_sorts.cpp`: ketiga pengurutan dasar, masing-masing mencetak larik setelah setiap lintasan beserta hitungannya. 365
Kode 4.30 `TraceSorts.java`: ketiga pengurutan yang ditelusuri, dalam Java. 368
Kode 4.31 `trace_sorts.py`: ketiga pengurutan yang ditelusuri, dalam Python. 370
Kode 4.32 `WrongBubble.java`: gelung dalam bubble sort berjalan satu langkah terlalu jauh. 372
Kode 4.33 `merge_sorted.cpp`: penggabungan dua larik terurut, sambil menghitung perbandingan. 374
Kode 4.34 `MergeSorted.java`: penggabungan yang sama dalam Java, mengembalikan larik baru. 375
Kode 4.35 `merge_sorted.py`: penggabungan yang sama dalam Python, dengan ekornya ditambahkan melalui `extend` 376
Kode 4.36 `wrong_merge.cpp`: penggabungan yang melupakan elemen-elemen sisa. 377
Kode 4.37 `delete_value.cpp`: menghapus setiap kemunculan sebuah nilai dengan indeks baca dan indeks tulis. 380
Kode 4.38 `DeleteValue.java`: penghapusan dua indeks yang sama dalam Java. 382
Kode 4.39 `delete_value.py`: penghapusan dua indeks yang sama dalam Python. 382
Kode 4.40 `wrong_delete_value.py`: penghapusan dengan geser-kiri yang selalu memajukan indeks. 383
Kode 4.41 `pair_sum.cpp`: urutkan harga, lalu temukan setiap pasangan yang cocok dengan jalan dua indeks 386
Kode 4.42 `PairSum.java`: pengurutan dan jalan dua indeks yang sama dalam Java. 388
Kode 4.43 `pair_sum.py`: pengurutan dan jalan dua indeks yang sama dalam Python. 389
Kode 4.44 `wrong_pair_sum.cpp`: jalan dua indeks pada harga yang tidak pernah diurutkan. 390
Kode 5.1 `01_hanoi_stacks.cpp`: Menara Hanoi diselesaikan dengan tiga stack dan sebuah perulangan, disusul banyaknya langkah untuk 1 sampai 10 cakram. 402
Kode 5.2 `02_array_stack.cpp`: *template* stack berbasis larik dalam C++, dipakai sebagai troli pengembalian buku Pustaka Ceria. 408
Kode 5.3 `ArrayStackDemo.java`: stack larik yang sama sebagai kelas generik Java. 411

Kode 5.4 `02_array_stack.py`: stack larik yang sama dalam Python, dengan *list* berukuran tetap dan indeks puncak eksplisit. 413
Kode 5.5 `03_growing_stack.cpp`: stack yang menggandakan lariknya ketika penuh, sambil menghitung setiap penyalinan elemen. 416
Kode 5.6 `04_linked_stack.cpp`: pratinjau stack yang dibangun dari simpul-simpul bertaut. 418
Kode 5.7 `05_brackets.cpp`: pencocokan tanda kurung dengan `std::stack`, melaporkan posisi dan jenis galat pertama. 421
Kode 5.8 `06_undo_redo.cpp`: *undo* dan *redo* dengan dua stack berisi keadaan yang disimpan. 423
Kode 5.9 `07_calculator.cpp`: kalkulator yang mengubah infiks menjadi postfiks dengan algoritma *shunting-yard* dan mengevaluasi postfiks dengan stack operan. 429
Kode 5.10 `08_call_stack.cpp`: menampakkan call stack dengan melakukan *push* nama pada setiap masuk fungsi dan *pop* pada setiap kembali. 435
Kode 5.11 `09_linear_queue.cpp`: queue bergeser-elemen (benar tetapi $O(n)$ per *dequeue*) dan queue yang indeksnya bergeser ($O(1)$ tetapi penuh palsu). 440
Kode 5.12 `10_circular_queue.cpp`: queue sirkular dengan pencacah, sebagai antrean buku baru Pustaka Ceria. 445
Kode 5.13 `CircularQueueDemo.java`: queue sirkular sebagai kelas generik Java. 447
Kode 5.14 `10_circular_queue.py`: queue sirkular dalam Python. 448
Kode 5.15 `11_circular_queue_slot.cpp`: queue sirkular yang menyisakan satu slot kosong dan menghitung ukurannya dari kedua indeks. 451
Kode 5.16 `12_array_deque.cpp`: deque berbasis larik sirkular, dan uji palindrom yang mengambil dari kedua ujung. 454
Kode 5.17 `13_library.cpp`: `std::stack`, `std::queue`, `std::deque`, dan `std::priority_queue`. 458
Kode 5.18 `Library.java`: `ArrayDeque` sebagai stack, sebagai queue, dan sebagai deque, serta `PriorityQueue` dalam kedua urutan. 459
Kode 5.19 `13_library.py`: *list* sebagai stack, `collections.deque` sebagai queue dan deque, serta `heapq` sebagai priority queue. 461
Kode 6.1 `01_first_pointer.cpp`: mengambil alamat dengan `&`, menyimpannya dalam pointer, dan mengikuti pointer itu dengan `*`. 476
Kode 6.2 `02_pointer_assign.cpp`: `q = p` menyalin alamat, `*q = *p` menyalin nilai. 479
Kode 6.3 `03_pointer_arith.cpp`: setiap langkah `p + i` bergeser `sizeof(T)` byte, dan pengurangan pointer menghitung elemen. 482
Kode 6.4 `04_array_walk.cpp`: melompat, melangkah, dan menulis melalui pointer, lalu loop satu-lewat-ujung. 485
Kode 6.5 `05_danaya.cpp`: menyusuri string C maju dan mundur dengan pointer; `overrun` mengulang kesalahan klasik. 487
Kode 6.6 `06_array_param.cpp`: argumen larik tiba sebagai pointer; cacah, rentang pointer, atau kontainer melengkapi panjang yang hilang. 488
Kode 6.7 `07_book_pointer.cpp`: operator panah pada larik `Book`, dan rantai simpul yang pertama. 490
Kode 6.8 `08_dynamic.cpp`: objek tunggal dan larik di *heap*, serta larik tumbuh yang menggandakan kapasitasnya. 494
Kode 6.9 `09_mem_bugs.cpp`: tujuh bug pointer klasik, masing-masing dipilih dengan kata di baris perintah, beserta bentuk yang benar. 498
Kode 6.10 `10_nullptr.cpp`: `nullptr` memilih *overload* pointer, dan pencarian mengembalikan `nullptr` untuk "tidak ditemukan". 504
Kode 6.11 `11_references.cpp`: referensi adalah alias; menugasinya berarti menugasi objeknya, dan loop `for` atas referensi mengubah kontainernya. 506
Kode 6.12 `12_functions_scope.cpp`: prototipe, parameter formal dan aktual, variabel global, lokal, dan statis, serta perbandingan pertama antara pengiriman berdasarkan nilai dan melalui pointer. 508
Kode 6.13 `13_swap.cpp`: menukar berdasarkan nilai (gagal), melalui pointer dan melalui referensi (keduanya berhasil), serta `std::swap`. 510

Kode 6.14 `14_const_correct.cpp`: empat bentuk `const`, fungsi anggota const, dan biaya penyalinan yang diukur dengan penghitung salinan. 514
Kode 6.15 `15_find_book.cpp`: mengembalikan pointer ke dalam larik milik pemanggil atau `nullptr`, referensi ke dalam vector milik pemanggil, dan nilai baru. 516
Kode 6.16 `16_unique_ptr.cpp`: membuat, memindahkan, menyerahkan, dan mengumpulkan pemilik `std::unique_ptr`; tidak ada `delete` di mana pun. 519
Kode 6.17 `17_shared_ptr.cpp`: buku yang dibagi dua daftar dan dihancurkan ketika pemilik terakhir melepaskannya, serta sepasang anggota yang ditautkan dengan `weak_ptr` sehingga keduanya dapat dihancurkan. 522
Kode 6.18 `RefsDemo.java`: aliasing, mutasi melalui parameter, pengikatan ulang parameter, argumen primitif, salinan sungguhan, `==` versus `equals`, dan referensi null. 526
Kode 6.19 `refs_demo.py`: eksperimen yang sama dengan nama-nama Python, ditambah jebakan argumen bawaan yang dapat diubah. 528
Kode 6.20 `SwapDemo.java`: Java dapat menukar isi dua objek, atau dua slot sebuah larik, tetapi tidak pernah dua variabel milik pemanggil. 530
Kode 6.21 `swap_demo.py`: percobaan yang sama dalam Python, diakhiri dengan penugasan tuple yang menukar di pihak pemanggil. 531
Kode 6.22 `18_laundry.cpp`: antrean Profitina Laundry memiliki pesanannya melalui `unique_ptr`; pemindahan antrean memindahkan kepemilikan, bukan pesanannya. 535
Kode 6.23 `laundry.py`: antrean yang sama dalam Python, tempat memindahkan pesanan antarantrean berarti memindahkan referensi. 538
Kode 6.24 `19_sol_functions.cpp`: penyelesaian Soal 6.12. 544
Kode 6.25 `20_sol_pointers.cpp`: penyelesaian Soal 6.10 dan 6.15. 545
Kode 7.1 `node_basics.cpp`: tiga simpul dibuat dengan `new`, ditautkan dengan tangan, ditelusuri, dan dilepaskan dengan `delete`. 556
Kode 7.2 `malloc_node.cpp`: simpul yang dialokasikan dengan `malloc` dan dilepaskan dengan `free`, di samping simpul yang dibuat dengan `new` dan dilepaskan dengan `delete`. 557
Kode 7.3 `sll_head.cpp`: *single linked list* dengan pointer *head*, dengan setiap operasi dari Subbab 7.5. 565
Kode 7.4 `SllHead.java`: *single linked list* berreferensi *head* dalam Java. 569
Kode 7.5 `sll_head.py`: *single linked list* berreferensi *head* dalam Python. 572
Kode 7.6 `sll_ops.cpp`: pencarian, panjang, penyisipan sesudah simpul dan secara terurut, penghapusan berdasarkan kunci, pembalikan iteratif dan rekursif, serta pemusnahan. 582
Kode 7.7 `sll_dummy.cpp`: *single linked list* dengan simpul kepala *dummy*; penyisipan dan penghapusan tidak memerlukan kasus khusus untuk elemen pertama. 587
Kode 7.8 `sll_link.cpp`: menjalankan pointer ke tautan (`Node **`) menghapus kasus khusus untuk simpul pertama tanpa simpul *dummy*. 590
Kode 7.9 `sll_head_tail.cpp`: *single linked list* dengan `head` dan `tail`; penyisipan belakang $O(1)$, penghapusan belakang tetap berjalan. 594
Kode 7.10 `csll.cpp`: *single linked list* melingkar yang dicapai lewat `last`, dipakai untuk layanan bergiliran dengan jatah waktu 2 menit. 598
Kode 7.11 `linked_stack_queue.cpp`: `LinkedStack` dan `LinkedQueue` yang dibangun dari simpul bertaut tunggal, dengan destruktor yang melepaskan setiap simpul. 602
Kode 7.12 `memory_bugs.cpp`: satu eksekusi yang benar dan empat galat memori yang disengaja (`leak`, `uaf`, `dangle`, `double`). 607
Kode 7.13 `std_lists.cpp`: `std::forward_list` yang bertaut tunggal dan `std::list` yang bertaut ganda. 614
Kode 7.14 `LibraryLists.java`: `java.util.LinkedList` yang dipakai sebagai list dan sebagai *deque*. 616
Kode 7.15 `library_lists.py`: `collections.deque`, padanan bawaan Python yang paling dekat dengan *linked list*. 618
Kode 7.16 `bench.cpp`: penyisipan depan dan penelusuran diukur waktunya untuk larik dan *linked list*. 619
Kode 7.17 `exercises.cpp`: penyelesaian Soal Latihan 7.3 sampai 7.9 dan 7.11. 627

Kode 8.1 `p801_stack.cpp`: stack larik Soal 8.1, sudah dilengkapi, dengan *driver* yang diberikan dan pemeriksaan kasus tepi kita sendiri. 642
Kode 8.2 `P801Stack.java`: Soal 8.1 dalam Java. 644
Kode 8.3 `p801_stack.py`: Soal 8.1 dalam Python, dengan blok tetap berisi `MAX` sel agar logikanya sama dengan larik C++. 646
Kode 8.4 `p802_deque_walk.cpp`: Soal 8.2, list dua arah yang dibangun oleh `slist_pushBack` dan ditelusuri oleh `traverse`. 651
Kode 8.5 `P802DequeWalk.java`: Soal 8.2 dalam Java. 654
Kode 8.6 `p802_deque_walk.py`: Soal 8.2 dalam Python. `__slots__` menetapkan medan-medan sebuah simpul, sehingga setiap simpul lebih kecil dan salah ketik pada nama medan tertangkap. 656
Kode 8.7 `p803_evens_odds.cpp`: Soal 8.3 dengan dua linked list terurut dan satu fungsi penyisipan yang menerima urutannya sebagai pointer ke fungsi. 661
Kode 8.8 `P803EvensOdds.java`: Soal 8.3 dalam Java, dengan urutan yang dikirim sebagai ekspresi lambda. 663
Kode 8.9 `p803_evens_odds.py`: Soal 8.3 dalam Python, dengan urutan yang dikirim sebagai fungsi. 665
Kode 8.10 `p804_reverse_words.cpp`: Soal 8.4 dengan linked list yang dipakai sebagai stack. 668
Kode 8.11 `P804ReverseWords.java`: Soal 8.4 dalam Java. 670
Kode 8.12 `p804_reverse_words.py`: Soal 8.4 dalam Python. 670
Kode 8.13 `p805_remove_duplicates.cpp`: Soal 8.5 dengan pembuangan dua pointer, ditambah `slist_popBack` yang membuang penanda akhir. 674
Kode 8.14 `P805RemoveDuplicates.java`: Soal 8.5 dalam Java. 677
Kode 8.15 `p805_remove_duplicates.py`: Soal 8.5 dalam Python. 679
Kode 8.16 `p806_brackets.cpp`: Soal 8.6, pencocokan tanda kurung dengan stack larik yang kapasitasnya dipilih saat program berjalan. 683
Kode 8.17 `P806Brackets.java`: Soal 8.6 dalam Java. 684
Kode 8.18 `p806_brackets.py`: Soal 8.6 dalam Python. 686
Kode 8.19 `p807_circular_queue.cpp`: Soal 8.7, queue larik sirkular dengan `front` dan `count`. 689
Kode 8.20 `P807CircularQueue.java`: Soal 8.7 dalam Java. 691
Kode 8.21 `p807_circular_queue.py`: Soal 8.7 dalam Python. 693
Kode 8.22 `p808_postfix.cpp`: Soal 8.8, pengubahan *shunting-yard* dan evaluasi postfiks dengan `std::stack`. 696
Kode 8.23 `P808Postfix.java`: Soal 8.8 dalam Java, dengan `ArrayDeque` sebagai stack. 698
Kode 8.24 `p808_postfix.py`: Soal 8.8 dalam Python, dengan list sebagai stack. 700
Kode 8.25 `p809_three_ways.cpp`: Soal 8.9, tiga cara mengirim argumen di C++, dengan pemeriksaan memori mana yang dipakai setiap parameter. 703
Kode 8.26 `P809ThreeWays.java`: Soal 8.9 dalam Java, yang mengirim setiap argumen by value dan yang variabel objeknya menyimpan referensi. 705
Kode 8.27 `p809_three_ways.py`: Soal 8.9 dalam Python, yang parameternya adalah nama baru bagi objek yang sama. 707
Kode 8.28 `p810_reverse_list.cpp`: Soal 8.10, pembalikan di tempat dengan tiga pointer, ditelusuri langkah demi langkah. 711
Kode 8.29 `P810ReverseList.java`: Soal 8.10 dalam Java. 713
Kode 8.30 `p810_reverse_list.py`: Soal 8.10 dalam Python. 714
Kode 8.31 `p811_merge_sorted.cpp`: Soal 8.11, menggabungkan dua linked list terurut dengan simpul dummy. 717
Kode 8.32 `P811MergeSorted.java`: Soal 8.11 dalam Java. 719
Kode 8.33 `p811_merge_sorted.py`: Soal 8.11 dalam Python. 721
Kode 8.34 `p812_linked_queue.cpp`: Soal 8.12, queue yang dibangun dari linked list dengan `head` dan `tail` 724
Kode 8.35 `P812LinkedQueue.java`: Soal 8.12 dalam Java. 726
Kode 8.36 `p812_linked_queue.py`: Soal 8.12 dalam Python. 728
Kode 8.37 `ex08_solutions.cpp`: soal latihan pemrograman Bab 8 dalam C++. 735
Kode 8.38 `Ex08Solutions.java`: soal latihan pemrograman Bab 8 dalam Java. 739

Kode 8.39 `ex08_solutions.py`: soal latihan pemrograman Bab 8 dalam Python. 743
Kode 9.1 `dll_node.cpp`: tiga simpul *double linked* dibuat dengan `new`, ditautkan ke dua arah dengan tangan, disusuri ke dua arah, lalu dilepaskan. 752
Kode 9.2 `dll_head.cpp`: *double linked list* dengan pointer *head*, beserta setiap operasi dari Subbab 9.3. 760
Kode 9.3 `DllHead.java`: *double linked list* dengan referensi *head* dalam Java. 765
Kode 9.4 `dll_head.py`: *double linked list* dengan referensi *head* dalam Python. `__slots__` menetapkan ketiga medan sebuah simpul. 768
Kode 9.5 `dll_ops.cpp`: *double linked list* dengan *head* dan *tail*: penyisipan dan penghapusan di depan dan di belakang, penyisipan sesudah dan sebelum simpul, penyisipan terurut, pencarian, penghapusan berdasarkan pointer dan kunci, kedua penyusuran, serta pemeriksaan tautan. 776
Kode 9.6 `dll_headnode.cpp`: *double linked list* dengan simpul kepala; setiap operasi tinggal punya satu kasus khusus, di ujung belakang. 782
Kode 9.7 `cdll.cpp`: *double linked list* sirkular yang dicapai lewat `head`, dipakai sebagai korsel lobi; `next` dan `previous` berputar di kedua ujung. 786
Kode 9.8 `cdll_sentinel.cpp`: *double linked list* sirkular dengan *sentinel*; `insertAfter` dan `deleteNode` tidak memuat kasus khusus, dan setiap operasi lain dibangun di atasnya. 792
Kode 9.9 `deque.cpp`: *deque* berisi buku di atas *double linked list* sirkular dengan *sentinel*, dipakai di meja pengembalian Pustaka Ceria. 796
Kode 9.10 `recently_borrowed.cpp`: panel "baru dipinjam", daftar LRU yang dibangun dari *double linked list* sirkular dengan *sentinel* dan `std::unordered_map` dari judul ke simpul. 801
Kode 9.11 `bench.cpp`: mengosongkan list dari belakang, dan menghapus simpul lewat pointer yang sudah dipegang program, untuk *single* dan *double linked list*. 804
Kode 9.12 `memory_bugs.cpp`: satu jalan yang benar dan tiga galat *double linked list* yang disengaja (`halflink`, `stale`, `leak`) 809
Kode 9.13 `std_dll.cpp`: `std::list` disusuri ke dua arah, dengan penyisipan dan penghapusan $O(1)$ pada iterator, iterator yang stabil, dan `splice`. 815
Kode 9.14 `LibraryDll.java`: `LinkedList` disunting lewat `ListIterator` dan disusuri mundur, serta `LinkedHashMap` dalam urutan akses yang dipakai sebagai daftar LRU berkapasitas 3. 816
Kode 9.15 `library_dll.py`: `collections.deque` (dengan `rotate` dan `maxlen`), `OrderedDict` yang dipakai sebagai daftar LRU, dan `functools.lru_cache`. 818
Kode 9.16 `laundry_weeks.cpp`: total pesanan mingguan Profitina Laundry dalam *double linked list*, dijelajahi satu minggu mundur atau maju dari minggu mana pun. 821
Kode 9.17 `exercises.cpp`: penyelesaian Soal 9.3 sampai 9.9. 826
Kode 10.1 `01_recursive_f.cpp`: definisi rekursif Persamaan (10.1) sebagai fungsi C++, ditelusuri, dan diperiksa terhadap bentuk tertutupnya. 839
Kode 10.2 `02_factorial.cpp`: faktorial dengan gelung dan dengan rekursi, jejak `fact(4)`, dan batas aritmetika 64 bit. 843
Kode 10.3 `Factorial.java`: program yang sama dalam Java; `BigInteger` memberi 21! yang eksak. 845
Kode 10.4 `02_factorial.py`: program yang sama dalam Python, yang bilangan bulatnya tumbuh sesuai kebutuhan. 846
Kode 10.5 `03_fibonacci.cpp`: kelinci-kelinci, dan $F(n)$ dengan rekursi naif, memoisasi, tabel *bottom-up*, dua variabel, rekursi ekor, *fast doubling*, dan rumus Binet. 854
Kode 10.6 `05_gcd.cpp`: algoritma Euklides secara rekursif dan iteratif, ditelusuri, beserta kasus terburuknya. 863
Kode 10.7 `06_power.cpp`: pemangkatan dengan n perkalian dan dengan pengkuadratan berulang, beserta banyaknya panggilan, dan pemangkatan modular. 867
Kode 10.8 `07_hanoi.cpp`: Menara Hanoi secara rekursif, dengan setiap langkah diperiksa pada tiga *stack* sungguhan. 870
Kode 10.9 `Hanoi.java`: Menara Hanoi secara rekursif dalam Java. 872
Kode 10.10 `07_hanoi.py`: Menara Hanoi secara rekursif dalam Python. 874
Kode 10.11 `08_arrays_lists.cpp`: rekursi atas larik (mengupas satu elemen atau membelah dua) dan atas *singly linked list* (simpul pertama dan sisanya). 878

Kode 10.12 `09_tail.cpp`: rekursi bukan-ekor, rekursi ekor, dan rekursi yang *frame*-nya harus bertahan, dapat dipilih dari baris perintah. 885
Kode 10.13 `TailCalls.java`: di Java, metode rekursif ekor memakai satu *frame* per panggilan, persis seperti metode lainnya. 888
Kode 10.14 `09_tail.py`: di Python, fungsi rekursif ekor menabrak batas rekursi seperti fungsi lainnya. 889
Kode 10.15 `10_depth.cpp`: mengukur ukuran sebuah *stack frame* dan melakukan rekursi sampai kedalaman yang dipilih. 891
Kode 10.16 `Depth.java`: kedalaman rekursi yang dapat dicapai di *thread* utama Java, dan rekursi dalam di sebuah *thread* dengan *stack* yang lebih besar. 892
Kode 10.17 `10_depth.py`: batas rekursi Python, cara menaikkannya, dan rekursi dalam di sebuah *thread* dengan *stack* yang lebih besar. 894
Kode 10.18 `11_explicit_stack.cpp`: Menara Hanoi dengan *stack* eksplisit berisi *frame*, dan list berisi sejuta simpul yang ditelusuri mundur. 897
Kode 10.19 `12_laundry_menu.cpp`: fungsi-fungsi rekursif atas menu layanan Profitina Laundry yang bersarang, dan penghitungan yang sama dengan *stack* eksplisit. 901
Kode 10.20 `13_exercises.cpp`: penyelesaian soal-soal latihan pemrograman Bab 10. 907
Kode 11.1 `array_tree.cpp`: pohon biner yang disimpan dalam larik; anak dan induk ditemukan dengan aritmetika indeks. 923
Kode 11.2 `traversals_iter.cpp`: *preorder*, *inorder*, dan *postorder* dengan *stack* eksplisit, serta urutan tingkat dengan *queue* dari simpul bertaut, disertai jejak isi *stack* dan *queue* 933
Kode 11.3 `expr_tree.cpp`: pohon ekspresi yang dibangun dari notasi postfiks, dicetak dalam notasi prefiks, infiks, dan postfiks, serta dievaluasi dengan penelusuran *postorder* 939
Kode 11.4 `bst.cpp`: *binary search tree* dengan penyisipan, empat penelusuran, pencarian, hitungan simpul, hitungan daun, tinggi, minimum, maksimum, daftar daun, pencetakan menyamping, dan pelepasan semua simpul. 947
Kode 11.5 `Bst.java`: *binary search tree* dari **Kode 11.4** dalam Java; `insert` mengembalikan akar baru setiap subpohon. 952
Kode 11.6 `bst.py`: *binary search tree* yang sama dalam Python. 956
Kode 11.7 `bst_delete.cpp`: *successor* dan *predecessor* tanpa pointer induk, serta penghapusan dalam ketiga kasus dengan varian *successor* dan varian *predecessor* 965
Kode 11.8 `height_experiment.cpp`: tinggi dan kedalaman rata-rata *binary search tree* yang dibangun dari urutan penyisipan acak dan terurut, beserta waktu pembangunannya. 973
Kode 11.9 `avl_first_look.cpp`: penyisipan AVL dengan rotasi tunggal dan ganda, disuapi kunci terurut yang mengubah *binary search tree* biasa menjadi rantai. 978
Kode 11.10 `general_tree.cpp`: pohon umum dalam bentuk anak-pertama/saudara-berikutnya, konversinya menjadi pohon biner, dan pohon-pohon yang dibangun kembali dari daftar *preorder* dan derajat. 985
Kode 11.11 `std_map.cpp`: katalog Pustaka Ceria dalam `std::map`, dengan daftar terurut, pencarian, kueri rentang, *successor* dan *predecessor*, minimum dan maksimum, serta penghapusan; dan `std::set` yang menolak duplikat 990
Kode 11.12 `CatalogTreeMap.java`: katalog yang sama dalam `java.util.TreeMap`, dengan `subMap`, `higherKey`, `lowerKey`, `firstKey`, dan `lastKey`; serta sebuah `TreeSet` 992
Kode 11.13 `library_trees.py`: katalog yang dijaga terurut dengan list terurut dan modul `bisect`, dengan `dict` untuk kodenya 994
Kode 11.14 `sorted_catalog.py`: katalog dalam `SortedDict` dari paket `sortedcontainers` 996
Kode 11.15 `laundry_tree.cpp`: papan pengambilan Profitina Laundry, sebuah *binary search tree* berisi pesanan berkunci waktu siap, dengan kueri rentang yang dipangkas, *ceiling*, *floor*, dan penghapusan. 997
Kode 11.16 `exercises.cpp`: penyelesaian Soal 11.6 serta 11.8 sampai 11.11. 1007
Kode 12.1 `tree_kit.hpp`: *toolkit* C++ yang dipakai bersama oleh program-program pohon bab ini. 1020
Kode 12.2 `TreeKit.java`: *toolkit* Java. 1023
Kode 12.3 `tree_kit.py`: *toolkit* Python. 1024
Kode 12.4 `p1200_sample_tree.cpp`: pohon bab ini, dibangun dan diamati. 1026

Kode 12.5 `P1200SampleTree.java`: pohon diamati dalam Java. 1027
Kode 12.6 `p1200_sample_tree.py`: pohon diamati dalam Python. 1028
Kode 12.7 `p1201_depth_first.cpp`: Soal 12.1, secara rekursif dan dengan stack eksplisit. 1030
Kode 12.8 `P1201DepthFirst.java`: Soal 12.1 dalam Java. 1031
Kode 12.9 `p1201_depth_first.py`: Soal 12.1 dalam Python. 1031
Kode 12.10 `p1202_kth_level.cpp`: Soal 12.2, mencetak setiap tingkat dari 0 sampai satu tingkat melewati tingginya. 1034
Kode 12.11 `P1202KthLevel.java`: Soal 12.2 dalam Java. 1035
Kode 12.12 `p1202_kth_level.py`: Soal 12.2 dalam Python. 1036
Kode 12.13 `p1203_sum.cpp`: Soal 12.3, satu baris rekursi. 1038
Kode 12.14 `P1203Sum.java`: Soal 12.3 dalam Java. 1039
Kode 12.15 `p1203_sum.py`: Soal 12.3 dalam Python. 1039
Kode 12.16 `p1204_triple.cpp`: Soal 12.4, mengubah simpul melalui pointer. 1042
Kode 12.17 `P1204Triple.java`: Soal 12.4 dalam Java. 1043
Kode 12.18 `p1204_triple.py`: Soal 12.4 dalam Python. 1043
Kode 12.19 `p1205_mirror.cpp`: Soal 12.5, penukaran dengan pointer yang disimpan di setiap simpul. 1045
Kode 12.20 `P1205Mirror.java`: Soal 12.5 dalam Java. 1047
Kode 12.21 `p1205_mirror.py`: Soal 12.5 dalam Python, dengan penukaran ditulis sebagai satu penugasan *tuple*. 1047
Kode 12.22 `p1206_balanced.cpp`: Soal 12.6, fungsi bantu yang memakai `height`, dan versi satu-lintasan. 1050
Kode 12.23 `P1206Balanced.java`: Soal 12.6 dalam Java. 1052
Kode 12.24 `p1206_balanced.py`: Soal 12.6 dalam Python. 1053
Kode 12.25 `p1207_delete.cpp`: Soal 12.7, penghapusan pada ketiga kasus, dengan setiap hasil dipasang kembali oleh induknya. 1057
Kode 12.26 `P1207Delete.java`: Soal 12.7 dalam Java. 1059
Kode 12.27 `p1207_delete.py`: Soal 12.7 dalam Python. 1060
Kode 12.28 `p1208_find_parent.cpp`: Soal 12.8, versi pohon pencarian dan versi pohon apa pun. 1064
Kode 12.29 `P1208FindParent.java`: Soal 12.8 dalam Java. 1065
Kode 12.30 `p1208_find_parent.py`: Soal 12.8 dalam Python. 1065
Kode 12.31 `p1209_find_child.cpp`: Soal 12.9, satu pemanggilan `search` dan dua pengamatan yang cermat. 1067
Kode 12.32 `P1209FindChild.java`: Soal 12.9 dalam Java. 1068
Kode 12.33 `p1209_find_child.py`: Soal 12.9 dalam Python. 1069
Kode 12.34 `p1210_to_dll.cpp`: Soal 12.10, penelusuran *inorder* yang menambahkan setiap simpul ke list. 1073
Kode 12.35 `P1210ToDll.java`: Soal 12.10 dalam Java, dengan `head` dan `prev` disimpan di satu objek bersama. 1076
Kode 12.36 `p1210_to_dll.py`: Soal 12.10 dalam Python. 1077
Kode 12.37 `p1211_reverse_dll.cpp`: Soal 12.11, menukar kedua panah setiap simpul. 1080
Kode 12.38 `P1211ReverseDll.java`: Soal 12.11 dalam Java. 1082
Kode 12.39 `p1211_reverse_dll.py`: Soal 12.11 dalam Python. 1084
Kode 12.40 `p1212_josephus.cpp`: Soal 12.12, list sirkular dan rumus rekursif, dibandingkan. 1088
Kode 12.41 `P1212Josephus.java`: Soal 12.12 dalam Java. 1089
Kode 12.42 `p1212_josephus.py`: Soal 12.12 dalam Python. 1090
Kode 12.43 `p1213_choose_books.cpp`: Soal 12.13, setiap pilihan tiga buku dengan rekursi ambil-atau-lewati dan *backtracking* 1093
Kode 12.44 `P1213ChooseBooks.java`: Soal 12.13 dalam Java. 1095
Kode 12.45 `p1213_choose_books.py`: Soal 12.13 dalam Python. 1095
Kode 12.46 `p1214_is_bst.cpp`: Soal 12.14, setiap simpul diperiksa terhadap interval yang diwarisi dari leluhurnya. 1097

Kode 12.47 `P1214IsBst.java`: Soal 12.14 dalam Java. 1098
Kode 12.48 `p1214_is_bst.py`: Soal 12.14 dalam Python. 1099
Kode 12.49 `p1215_lca.cpp`: Soal 12.15, satu penelusuran turun sampai kedua nilai berpisah. 1101
Kode 12.50 `P1215Lca.java`: Soal 12.15 dalam Java. 1102
Kode 12.51 `p1215_lca.py`: Soal 12.15 dalam Python. 1103
Kode 12.52 `p1216_levels.cpp`: Soal 12.16, penelusuran *level order* yang mencatat lebar setiap tingkat. 1104
Kode 12.53 `P1216Levels.java`: Soal 12.16 dalam Java. 1105
Kode 12.54 `p1216_levels.py`: Soal 12.16 dalam Python. 1106
Kode 12.55 `ex12_solutions.cpp`: soal latihan pemrograman Bab 12 dalam C++. 1112
Kode 12.56 `Ex12Solutions.java`: soal latihan pemrograman Bab 12 dalam Java. 1113
Kode 12.57 `ex12_solutions.py`: soal latihan pemrograman Bab 12 dalam Python. 1115
Kode 13.1 `graph_repr.cpp`: peta kampus sebagai matriks ketetanggaan, list ketetanggaan, dan list sisi. 1130
Kode 13.2 `GraphRepr.java`: ketiga representasi dalam Java. 1133
Kode 13.3 `graph_repr.py`: ketiga representasi dalam Python. 1135
Kode 13.4 `bfs.cpp`: penelusuran melebar pada peta kampus, dengan jejak langkah, jarak langkah, orang tua, dan rute dengan langkah paling sedikit. 1142
Kode 13.5 `Bfs.java`: penelusuran melebar dalam Java, dengan `ArrayDeque` sebagai *queue*. 1145
Kode 13.6 `bfs.py`: penelusuran melebar dalam Python, dengan `collections.deque` sebagai *queue*. 1147
Kode 13.7 `dfs.cpp`: penelusuran mendalam rekursif dan iteratif dengan waktu penemuan dan waktu selesai, serta pendeteksian siklus dalam graf tak berarah. 1154
Kode 13.8 `Dfs.java`: penelusuran mendalam, rekursif dan iteratif, dalam Java. 1158
Kode 13.9 `dfs.py`: penelusuran mendalam, rekursif dan iteratif, dalam Python. 1160
Kode 13.10 `components.cpp`: komponen terhubung peta kampus, sebelum dan sesudah jalan *shuttle*. 1164
Kode 13.11 `Components.java`: komponen terhubung dalam Java. 1166
Kode 13.12 `components.py`: komponen terhubung dalam Python. 1168
Kode 13.13 `topo.cpp`: pengurutan topologis dengan algoritma Kahn dan dengan penelusuran mendalam, beserta pendeteksian siklus. 1174
Kode 13.14 `Topo.java`: pengurutan topologis dalam Java. 1177
Kode 13.15 `topo.py`: pengurutan topologis dalam Python. 1179
Kode 13.16 `dijkstra.cpp`: algoritma Dijkstra pada peta kampus dengan `std::priority_queue` sebagai *min-heap*. 1185
Kode 13.17 `Dijkstra.java`: algoritma Dijkstra dengan `java.util.PriorityQueue`. 1188
Kode 13.18 `dijkstra.py`: algoritma Dijkstra dengan modul `heapq`. 1189
Kode 13.19 `laundry_site.cpp`: kedalaman klik, tautan masuk, dan halaman yatim situs web Profitina Laundry, dengan penelusuran melebar. 1194
Kode 13.20 `LaundrySite.java`: audit situs web dalam Java. 1196
Kode 13.21 `laundry_site.py`: audit situs web dalam Python. 1197
Kode 13.22 `exercises.cpp`: penyelesaian Soal Latihan 13.10 sampai 13.12. 1203
Kode 13.23 `Exercises.java`: penyelesaian Soal Latihan 13.10 sampai 13.12 dalam Java. 1207
Kode 13.24 `exercises.py`: penyelesaian Soal Latihan 13.10 sampai 13.12 dalam Python. 1209
Kode 14.1 `heap_index.cpp`: induk dan anak-anak setiap simpul pohon *complete* bersepuluh simpul, dalam penomoran berbasis 0 dan berbasis 1. 1220
Kode 14.2 `build_count.cpp`: perbandingan kunci yang dilakukan n penyisipan dan satu pembangunan dari bawah ke atas, pada kunci naik, acak semu, dan turun. 1236
Kode 14.3 `max_heap.cpp`: kelas `MaxHeap` yang lengkap dan demonstrasi setiap operasinya. 1239
Kode 14.4 `MaxHeap.java`: kelas `MaxHeap` dalam Java, dengan larik yang digandakan ketika penuh. 1245
Kode 14.5 `max_heap.py`: kelas `MaxHeap` dalam Python. 1249

Kode 14.6 `heap_sort_full.cpp`: *heap sort* dengan jejak setiap keadaannya, dan pengurutan yang sama yang diterapkan pada rekaman buku untuk memperlihatkan bahwa ia tidak stabil. 1254
Kode 14.7 `library_pq.cpp`: `std::priority_queue` dengan perbandingan buatan sendiri untuk melayani permintaan pemesanan, serta antrean maks bawaan dan antrean min `std::greater` berisi bilangan bulat. 1259
Kode 14.8 `LibraryPQ.java`: `java.util.PriorityQueue` dengan pembanding yang dirakit dari `comparingInt`, `reversed`, dan `thenComparingInt`. 1261
Kode 14.9 `library_pq.py`: `heapq` dengan *tuple* sebagai rekaman, dan fungsi *max-heap* Python 3.14 dengan cadangan untuk versi yang lebih lama. 1262
Kode 14.10 `topk_merge.cpp`: tiga buku yang paling sering dipinjam dari sebuah aliran data dengan *min-heap* berukuran 3, dan penggabungan tiga katalog cabang yang terurut. 1265
Kode 14.11 `dijkstra_pq.cpp`: algoritma Dijkstra pada jalan-jalan pengantaran dengan `std::priority_queue` sebagai *min-heap* dan penghapusan malas. 1268
Kode 14.12 `exercises.cpp`: penyelesaian Soal 14.9 sampai 14.12. 1276
Kode 15.1 `hash_functions.cpp`: metode pembagian dan perkalian, hash *string* penjumlahan dan polinomial, pembaruan bergulir, serta pengukuran *avalanche*. 1290
Kode 15.2 `birthday.cpp`: peluang tabrakan eksak dan hampiran, banyaknya kunci untuk peluang seimbang dan hampir pasti, serta sebuah simulasi. 1296
Kode 15.3 `chaining.cpp`: tabel hash dengan *separate chaining* yang dibangun dari simpul-simpul *single linked list*, dengan pencarian, pembaruan, penghapusan, dan *rehashing*. 1301
Kode 15.4 `Chaining.java`: tabel hash berantai dalam Java. 1306
Kode 15.5 `chaining.py`: tabel hash berantai dalam Python. 1309
Kode 15.6 `open_addressing.cpp`: *linear probing*, *quadratic probing*, dan *double hashing* dalam tabel berisi 16 slot, dengan penghapusan melalui nisan. 1317
Kode 15.7 `OpenAddressing.java`: ketiga barisan *probe* dan nisan dalam Java. 1322
Kode 15.8 `open_addressing.py`: ketiga barisan *probe* dan nisan dalam Python; `None` tidak diperlukan karena keadaan slot disimpan dalam list tersendiri. 1325
Kode 15.9 `load_factor.cpp`: *probe* per pencarian terhadap faktor muat, terukur dan teramalkan; sebaran panjang rantai; dan ongkos *rehashing* dengan penggandaan. 1330
Kode 15.10 `flooding.cpp`: 2048 kunci yang semuanya bertabrakan di bawah hash polinomial publik berbasis 31, dan hash polinomial berkunci dengan basis rahasia yang kembali menyebarkannya. 1339
Kode 15.11 `maps.cpp`: `std::map` versus `std::unordered_map` pada data yang sama, antarmuka *bucket*, kueri rentang, fungsi hash untuk kunci dua medan, dan pengukuran waktu untuk satu juta kunci. 1344
Kode 15.12 `LibraryMaps.java`: `HashMap`, `TreeMap`, dan `LinkedHashMap` pada data yang sama, cara memilih *bin*, *record* sebagai kunci, 65.536 kunci yang bertabrakan, dan pengukuran waktu. 1349
Kode 15.13 `library_maps.py`: urutan penyisipan, nilai hash bilangan bulat dan *string*, pengacakan hash antarjalannya *interpreter*, pertumbuhan `dict`, dan pengukuran waktu terhadap pencarian biner. 1354
Kode 15.14 `applications.cpp`: menghitung kata dalam katalog, mendeteksi pengembalian ganda, *two-sum* pada harga buku, dan *cache* LRU berisi rekaman katalog. 1356
Kode 15.15 `exercises.cpp`: pemeriksaan Soal Latihan 15.1 sampai 15.4 dan penyelesaian Soal Latihan 15.6 sampai 15.9. 1366
Kode 16.1 `p1601_max_heap.cpp`: kelas *max-heap* Soal 16.1 dengan `extractMax`, `increaseKey`, `getMin`, `buildMaxHeap`, `isMaxHeap`, dan `clear`. 1383
Kode 16.2 `P1601MaxHeap.java`: Soal 16.1 dalam Java. 1390
Kode 16.3 `p1601_max_heap.py`: Soal 16.1 dalam Python. 1394
Kode 16.4 `p1602_map_vs_hash.cpp`: empat kunci yang sama di `std::map`, sebuah pohon pencarian, dan di `std::unordered_map`, sebuah tabel hash. 1401
Kode 16.5 `p1602_itinerary.cpp`: Soal 16.2 dengan dua *hash map*, dan laporan untuk setiap cara sebuah tumpukan tiket gagal membentuk satu rantai. 1403
Kode 16.6 `P1602Itinerary.java`: Soal 16.2 dalam Java dengan dua `HashMap`. 1406
Kode 16.7 `p1602_itinerary.py`: Soal 16.2 dalam Python dengan dua `dict` 1408

Kode 16.8 `p1603_bfs_hops.cpp`: penelusuran melebar dari Pusat, dengan lompatan paling sedikit, induk setiap cabang, dan tiga rute. 1412
Kode 16.9 `P1603BfsHops.java`: Soal 16.3 dalam Java. 1415
Kode 16.10 `p1603_bfs_hops.py`: Soal 16.3 dalam Python. 1417
Kode 16.11 `p1604_reading_order.cpp`: algoritma Kahn pada kesebelas topik buku, dengan dan tanpa busur yang menciptakan siklus. 1421
Kode 16.12 `P1604ReadingOrder.java`: Soal 16.4 dalam Java. 1424
Kode 16.13 `p1604_reading_order.py`: Soal 16.4 dalam Python. 1425
Kode 16.14 `p1605_dijkstra.cpp`: algoritma Dijkstra dengan *binary min-heap* dan penghapusan malas, menjejak setiap langkah. 1429
Kode 16.15 `P1605Dijkstra.java`: Soal 16.5 dalam Java. 1431
Kode 16.16 `p1605_dijkstra.py`: Soal 16.5 dalam Python. 1433
Kode 16.17 `p1606_top_k.cpp`: k nilai terbesar dengan *min-heap* berukuran k, dijejak pada delapan hitungan peminjaman dan dijalankan pada seluruh katalog dalam tiga urutan. 1437
Kode 16.18 `P1606TopK.java`: Soal 16.6 dalam Java. 1441
Kode 16.19 `p1606_top_k.py`: Soal 16.6 dalam Python. 1443
Kode 16.20 `p1607_running_median.cpp`: median berjalan dengan *max-heap* untuk separuh bawah dan *min-heap* untuk separuh atas, diperiksa terhadap pengurutan setelah setiap nilai. 1447
Kode 16.21 `P1607RunningMedian.java`: Soal 16.7 dalam Java. 1449
Kode 16.22 `p1607_running_median.py`: Soal 16.7 dalam Python, dengan separuh bawah disimpan dalam bentuk negasi. 1451
Kode 16.23 `p1608_chaining.cpp`: tabel hash dengan *separate chaining*, penyisipan di depan, penghapusan melalui pointer ke tautan, dan *rehash* ketika *load factor* melebihi 1. 1456
Kode 16.24 `P1608Chaining.java`: Soal 16.8 dalam Java. 1459
Kode 16.25 `p1608_chaining.py`: Soal 16.8 dalam Python. 1461
Kode 16.26 `p1609_linear_probing.cpp`: *open addressing* dengan *linear probing* dan *tombstone*, menjejak setiap barisan *probe*. 1466
Kode 16.27 `P1609LinearProbing.java`: Soal 16.9 dalam Java. 1469
Kode 16.28 `p1609_linear_probing.py`: Soal 16.9 dalam Python. 1470
Kode 16.29 `p1610_choose.cpp`: enam struktur diukur pada katalog yang sama (membangun, mencari, kueri rentang, menyisipkan), sepuluh buku dengan penilaian terbanyak dengan *heap* dan dengan pengurutan, serta graf orang-orang yang dicantumkan pada buku yang sama. 1476
Kode 16.30 `P1610Choose.java`: percobaan Subbab 16.8 dalam Java, dengan `TreeMap` sebagai pohon pencarian seimbang dan `HashMap` sebagai tabel hash. 1484
Kode 16.31 `p1610_choose.py`: percobaan Subbab 16.8 dalam Python, dengan `bisect` untuk larik terurut, simpul buatan sendiri untuk *list* dan pohon, dan `dict` sebagai tabel hash. 1494
Kode 16.32 `ex16_solutions.cpp`: penyelesaian soal-soal latihan pemrograman Bab 16. 1509
Kode 16.33 `Ex16Solutions.java`: penyelesaian yang sama dalam Java. 1514
Kode 16.34 `ex16_solutions.py`: penyelesaian yang sama dalam Python. 1519

Daftar Pseudokode

Pseudokode 1.1 `ARRAY-INSERT`: buka celah di indeks `pos` dengan menggeser elemen-elemen sesudahnya satu tempat ke kanan, lalu tulis `x` ke celah itu. 56
Pseudokode 1.2 `ARRAY-DELETE`: tutup celah yang ditinggalkan elemen di indeks `pos` dengan menggeser elemen-elemen sesudahnya satu tempat ke kiri. 56
Pseudokode 2.1 `SEQUENTIAL-SEARCH`: periksa elemen dari depan dan berhenti pada kecocokan pertama. 86
Pseudokode 2.2 `SENTINEL-SEARCH`: tanam salinan kunci sesudah elemen terakhir agar gelung hanya memerlukan satu uji per langkah. 97

Pseudokode 2.3 `BINARY-SEARCH`, iteratif, pada interval tertutup $[low, high]$. 102
Pseudokode 2.4 `BINARY-SEARCH-REC`: bentuk rekursif. Panggilan pertamanya adalah `BINARY-SEARCH-REC(A, x, 0, n - 1)`. 104
Pseudokode 2.5 `LOWER-BOUND`: posisi pertama yang elemennya tidak lebih kecil daripada x. Mengganti < dengan ≤ di baris 5 menghasilkan `UPPER-BOUND`. 112
Pseudokode 2.6 `INTERPOLATION-SEARCH`: pencarian biner dengan titik tengah diganti taksiran, dijaga terhadap kunci di luar rentang dan terhadap penyebut nol. 121
Pseudokode 2.7 `JUMP-SEARCH`: lompati blok-blok utuh, lalu telusuri satu-satunya blok yang dapat memuat kunci. 126
Pseudokode 2.8 `STAIRCASE-SEARCH`: pada kisi yang baris dan kolomnya sama-sama menaik, setiap pemeriksaan menyingkirkan satu baris utuh atau satu kolom utuh. 132
Pseudokode 3.1 *Bubble sort* dengan keluar dini: setiap lintasan membawa elemen terbesar yang tersisa ke ujung, dan lintasan tanpa pertukaran mengakhiri pengurutan. 204
Pseudokode 3.2 *Exchange sort*: posisi i dibandingkan dengan setiap posisi sesudahnya dan mengambil nilai lebih kecil yang dijumpainya. 210
Pseudokode 3.3 *Selection sort*: temukan elemen terkecil yang tersisa, lalu pindahkan ke tempatnya dengan satu pertukaran. 214
Pseudokode 3.4 *Insertion sort*: tumbuhkan awalan terurut dengan menyisipkan satu kunci setiap kali, menggeser elemen yang lebih besar ke kanan. 218
Pseudokode 3.5 *Shell sort*: *insertion sort* pada elemen yang berjarak h, untuk barisan celah menurun yang berakhir di 1. 223
Pseudokode 3.6 `MERGE` dan `MERGE-SORT`. Penyangga T sama panjang dengan A dan dialokasikan sekali saja, sebelum pemanggilan pertama. 231
Pseudokode 3.7 `QUICKSORT` dengan partisi yang mengembalikan indeks akhir pivot. 236
Pseudokode 3.8 Partisi Lomuto: satu penyusuran kiri-ke-kanan yang menumbuhkan wilayah elemen yang tidak lebih besar daripada pivot. 237
Pseudokode 3.9 Partisi Hoare: dua indeks bergerak saling mendekat dan menukar pasangan yang berada di sisi yang salah. 238
Pseudokode 3.10 `SIFT-DOWN` dan `HEAP-SORT`: bangun *max-heap*, lalu berulang kali pindahkan akar ke ujung *heap* yang menyusut dan perbaiki akarnya. 251
Pseudokode 3.11 *Counting sort* untuk kunci bilangan bulat di $0..k-1$. Tidak ada dua kunci yang pernah dibandingkan. 259
Pseudokode 4.1 Insertion sort untuk kata; perbandingan COMES-BEFORE mewujudkan urutan kamus. 319
Pseudokode 4.2 Karakter pertama yang berulang dalam dua lintasan, dengan satu pencacah untuk setiap karakter yang mungkin. 327
Pseudokode 4.3 Pencarian sekuensial pada larik rekaman; hasilnya sebuah indeks, atau −1 bila nama itu tidak ada. 333
Pseudokode 4.4 Aturan perbandingan dua kunci dan insertion sort stabil yang memakainya. 339
Pseudokode 4.5 Alamat elemen larik dalam satu dan dua dimensi (urutan row-major). 345
Pseudokode 4.6 Pencarian biner untuk kedua batas sebuah blok kunci yang sama dalam larik terurut. 350
Pseudokode 4.7 Pencarian interpolasi; uji rentang pada baris 2 menjaga setiap pemeriksaan tetap di dalam larik, dan baris 3 mencegah pembagian dengan nol. 356
Pseudokode 4.8 Bubble sort, selection sort, dan insertion sort, sebagaimana ditelusuri pada Soal 4.8. 364
Pseudokode 4.9 Menggabungkan dua larik terurut ke larik ketiga; paling banyak satu dari kedua gelung penyalin yang bekerja. 373
Pseudokode 4.10 Menghapus setiap kemunculan x dalam satu lintasan dengan indeks baca r dan indeks tulis w. 379
Pseudokode 4.11 Jalan dua indeks yang menemukan setiap pasangan berjumlah tertentu dalam larik terurut. 385
Pseudokode 5.1 Operasi stack pada larik $S[0..N-1]$ dengan indeks top; $top = -1$ berarti kosong. 406
Pseudokode 5.2 Memeriksa bahwa tanda kurung bersarang dengan benar menggunakan satu stack dan satu pemindaian kiri ke kanan. 420

Pseudokode 5.3 Algoritma *shunting-yard* Dijkstra: infiks ke postfiks dengan satu stack operator. 427
Pseudokode 5.4 Mengevaluasi ekspresi postfiks dengan satu stack operan. 428
Pseudokode 5.5 `ENQUEUE` dan `DEQUEUE` pada larik sirkular berisi *N* slot dengan indeks `front` dan pencacah `count`. 444
Pseudokode 6.1 `DYNAMIC-ARRAY-PUSH`: menambahkan `x`, setelah lebih dahulu menggandakan kapasitas bila larik penuh. 496
Pseudokode 7.1 `FRONT-INSERTION`: sisipkan simpul pertama yang baru; dua penugasan pointer, dengan urutan ini. 559
Pseudokode 7.2 `BACK-INSERTION` pada list yang hanya memiliki pointer *head*: berjalan ke simpul terakhir, lalu tempelkan. 560
Pseudokode 7.3 `TRAVERSE`: kunjungi setiap simpul dari yang pertama sampai yang terakhir. 562
Pseudokode 7.4 `FRONT-DELETION`: ingat simpul pertama, pindahkan `head` melewatinya, lalu lepaskan 563
Pseudokode 7.5 `BACK-DELETION` dengan pointer *head* saja: berhenti di simpul kedua dari belakang, putuskan tautannya, lepaskan simpul terakhir. 564
Pseudokode 7.6 `SEARCH`: kembalikan simpul pertama yang datanya sama dengan `key`, atau NIL. 575
Pseudokode 7.7 `INSERT-AFTER`: sisipkan simpul baru tepat sesudah simpul `p` dalam waktu $O(1)$. 576
Pseudokode 7.8 `INSERT-SORTED`: sisipkan `x` sehingga list tetap terurut naik. 577
Pseudokode 7.9 `DELETE-KEY`: buang simpul pertama yang datanya sama dengan `key`, dengan pointer pengekor `prev`. 578
Pseudokode 7.10 `REVERSE-ITERATIVE`: balik list dalam satu lintasan dengan membalik setiap panah `next`. 579
Pseudokode 7.11 `REVERSE-RECURSIVE`: balikkan sisa list, lalu gantungkan simpul pertama pada ujungnya. 581
Pseudokode 9.1 `FRONT-INSERTION` pada *double linked list* yang dicapai lewat `head`: tiga penugasan pointer, dan yang tengah hanya bila list tidak kosong. 755
Pseudokode 9.2 `BACK-INSERTION` dengan pointer *head* saja: berjalan ke simpul terakhir, lalu menautkan ke dua arah. 756
Pseudokode 9.3 `SHOW-BACKWARD` dengan pointer *head* saja: berjalan ke simpul terakhir, lalu mengunjungi setiap simpul dari yang terakhir sampai yang pertama. 757
Pseudokode 9.4 `FRONT-DELETION`: ingat simpul pertama, pindahkan `head`, kosongkan `prev` milik simpul pertama baru, lepaskan simpul lama. 758
Pseudokode 9.5 `BACK-DELETION` dengan pointer *head* saja: berjalan ke simpul terakhir; pendahulunya adalah `del.prev` 759
Pseudokode 9.6 `BACK-DELETION` dengan *head* dan *tail*: simpul sebelum yang terakhir adalah `tail.prev`, sehingga tidak perlu berjalan. 772
Pseudokode 9.7 `INSERT-AFTER`: menautkan simpul baru sesudah simpul `p` dengan empat penugasan pointer, dalam waktu $O(1)$; langkah 4 harus sesudah langkah 2 dan 3 773
Pseudokode 9.8 `INSERT-SORTED`: temukan simpul pertama yang tidak lebih kecil dari `x`, lalu sisipkan sebelumnya 774
Pseudokode 9.9 `DELETE-NODE`: melepas simpul `p` dalam waktu $O(1)$; tidak perlu pencarian maupun pointer pengekor. 775
Pseudokode 9.10 `LIST-INSERT-AFTER` pada list sirkular dengan *sentinel*: empat langkah **Pseudokode 9.7** tanpa ujinya. 790
Pseudokode 9.11 `LIST-DELETE` pada list sirkular dengan *sentinel*: dua penugasan, tanpa uji. 791
Pseudokode 9.12 `LIST-SEARCH-SENTINEL`: simpan kunci di dalam *sentinel*, sehingga perulangan cukup memuat satu uji per langkah. 792
Pseudokode 9.13 `BORROW` pada panel "baru dipinjam": satu pencarian *hash map* ditambah sejumlah tetap penugasan pointer. 800
Pseudokode 10.1 Faktorial dengan gelung dan dengan rekursi. 848
Pseudokode 10.2 Bilangan Fibonacci dengan rekursi bermemo. 852
Pseudokode 10.3 Algoritma Euklides, rekursif dan iteratif. 862
Pseudokode 10.4 Pemangkatan dengan pengkuadratan berulang. 866
Pseudokode 10.5 Menara Hanoi secara rekursif. 870

Pseudokode 11.1 Tiga penelusuran telusur-dalam; ketiganya hanya berbeda pada baris tempat simpul itu sendiri dikunjungi. 929
Pseudokode 11.2 `LEVEL-ORDER`: penelusuran telusur-lebar; anak-anak masuk di belakang *queue*, sehingga satu tingkat utuh dikunjungi sebelum simpul mana pun dari tingkat berikutnya. 931
Pseudokode 11.3 Penelusuran *inorder* dengan *stack* eksplisit: turun ke kiri sambil mendorong setiap simpul, ambil dan kunjungi, lalu lanjutkan dengan subpohon kanan. 932
Pseudokode 11.4 Membangun pohon ekspresi dari notasi postfiks dengan *stack* berisi pohon. 938
Pseudokode 11.5 `TREE-INSERT`: turun seperti pada pencarian dan gantungkan simpul baru pada subpohon kosong tempat perjalanan berakhir. 942
Pseudokode 11.6 Pencarian pada *binary search tree*, secara rekursif dan iteratif; keduanya mengembalikan simpul yang memuat k, atau NIL. 944
Pseudokode 11.7 Minimum adalah simpul paling kiri dan maksimum adalah simpul paling kanan dari pohon yang tidak kosong. 945
Pseudokode 11.8 Mencetak pohon secara menyamping: *inorder* terbalik, menjorok sesuai kedalaman. 946
Pseudokode 11.9 `TREE-SUCCESSOR` tanpa pointer induk: cari k sambil mengingat belokan kiri terakhir. 961
Pseudokode 11.10 `TREE-DELETE`, varian *successor*: daun dibuang, simpul beranak satu digantikan anaknya, dan simpul beranak dua mengambil kunci *successor*-nya. 963
Pseudokode 12.1 Templat fungsi pohon rekursif. Mengisi bagian kosongnya dengan 0 dan $T.key + a + b$ menghasilkan jumlah pada Soal 12.3; dengan -1 dan $1 + \max(a, b)$ menghasilkan tinggi. 1016
Pseudokode 13.1 `BFS`: penelusuran melebar dari sumber s, menghitung jarak langkah d dan orang tua π. 1137
Pseudokode 13.2 `DFS` dan `DFS-VISIT`: penelusuran mendalam dengan waktu penemuan dan waktu selesai. 1150
Pseudokode 13.3 `DFS-ITERATIVE`: penelusuran yang sama dengan *stack* eksplisit berisi pasangan (simpul, indeks tetangga berikutnya), menghasilkan waktu penemuan dan waktu selesai yang sama dengan versi rekursif. 1153
Pseudokode 13.4 `CONNECTED-COMPONENTS`: melabeli setiap simpul dengan nomor komponennya, satu penelusuran melebar untuk setiap komponen. 1163
Pseudokode 13.5 `KAHN`: pengurutan topologis dengan berulang-ulang membuang simpul berderajat-masuk 0. 1171
Pseudokode 13.6 `TOPOLOGICAL-SORT`: urutan topologis menurut waktu selesai DFS yang menurun. 1173
Pseudokode 13.7 `DIJKSTRA`: lintasan terpendek dari satu sumber dengan bobot tak negatif, memakai *min-priority queue* dengan penghapusan malas atas entri basi. 1183
Pseudokode 14.1 `SIFT-UP`: biarkan kunci di indeks i naik sampai induknya paling sedikit sama besar atau sampai ia mencapai akar. 1224
Pseudokode 14.2 `SIFT-DOWN`: biarkan kunci di indeks i turun, menukarnya dengan anak yang lebih besar, sampai tidak ada anak yang lebih besar. 1226
Pseudokode 14.3 `INSERT`: tambahkan kunci sebagai daun terakhir, lalu biarkan ia naik. 1228
Pseudokode 14.4 `EXTRACT-MAX`: simpan akar, pindahkan daun terakhir ke akar, kecilkan *heap*, dan biarkan akar yang baru turun. 1229
Pseudokode 14.5 `GET-MIN` untuk *max-heap*: minimum berada di daun, jadi pindailah daun-daun, yaitu paruh kedua larik. 1230
Pseudokode 14.6 `INCREASE-KEY` dan `DECREASE-KEY` pada *max-heap*: ubah kuncinya, lalu biarkan ia naik atau turun. 1231
Pseudokode 14.7 `BUILD-MAX-HEAP`: lakukan *sift-down* pada setiap simpul internal, dari yang terakhir mundur sampai akar. 1233
Pseudokode 14.8 `IS-MAX-HEAP`: periksa setiap anak terhadap induknya; larik kosong atau berisi satu elemen adalah *heap*. 1238
Pseudokode 14.9 `HEAP-SORT`: bangun *max-heap*, lalu berulang kali pindahkan akar ke ujung *heap* yang menyusut dan perbaiki akarnya. 1254
Pseudokode 14.10 `TOP-K`: simpan k butir terbaik di dalam *min-heap* yang akarnya adalah yang terlemah di antara mereka. 1264

Pseudokode 14.11 `MERGE-K`: gabungkan *k* senarai terurut dengan *min-heap* yang memuat satu elemen terdepan dari setiap senarai. 1264
Pseudokode 14.12 `SIFT-UP-HOLE`: kunci naik melalui lubang yang berpindah dan ditulis sekali di akhir. 1279
Pseudokode 15.1 Hash polinomial sebuah *string* dengan aturan Horner; M adalah 264 bila luapan tak bertanda mesin mengerjakan reduksinya secara cuma-cuma. 1288
Pseudokode 15.2 Pencarian, penyisipan, dan penghapusan pada tabel hash dengan *separate chaining* 1300
Pseudokode 15.3 Pencarian, penyisipan, dan penghapusan dengan pengalamatan terbuka dan nisan. 1317

Daftar Persamaan

(1.1) 8
(1.2) 27
(1.3) 40
(1.4) 42
(1.5) 49
(1.6) 56
(1.7) 57
(1.8) 61
(1.9) 74
(2.1) 83
(2.2) 96
(2.3) 96
(2.4) 104
(2.5) 104
(2.6) 113
(2.7) 120
(2.8) 127
(2.9) 127
(2.10) 131
(2.11) 174
(3.1) 197
(3.2) 203
(3.3) 209
(3.4) 209
(3.5) 209
(3.6) 223
(3.7) 224
(3.8) 236
(3.9) 236
(3.10) 249
(3.11) 249
(3.12) 250
(3.13) 258
(3.14) 259
(3.15) 260
(4.1) 317

(4.2) 317
(4.3) 345
(4.4) 345
(4.5) 350
(4.6) 356
(5.1) 401
(5.2) 402
(5.3) 402
(5.4) 406
(5.5) 417
(5.6) 443
(5.7) 443
(5.8) 450
(5.9) 451
(5.10) 453
(6.1) 481
(6.2) 482
(6.3) 485
(6.4) 490
(6.5) 496
(6.6) 513
(7.1) 552
(7.2) 553
(7.3) 553
(7.4) 553
(7.5) 554
(7.6) 576
(7.7) 581
(8.1) 640
(8.2) 651
(8.3) 661
(8.4) 674
(8.5) 688
(8.6) 717
(9.1) 751
(9.2) 754
(9.3) 791
(9.4) 792
(9.5) 800
(9.6) 804
(9.7) 804
(10.1) 837
(10.2) 838
(10.3) 842
(10.4) 849
(10.5) 850
(10.6) 851
(10.7) 851
(10.8) 851
(10.9) 853
(10.10) 854

(10.11) 861
(10.12) 865
(10.13) 865
(10.14) 866
(10.15) 866
(10.16) 876
(10.17) 883
(10.18) 891
(11.1) 918
(11.2) 922
(11.3) 925
(11.4) 928
(11.5) 928
(11.6) 928
(11.7) 928
(11.8) 937
(11.9) 946
(11.10) 971
(11.11) 972
(11.12) 973
(11.13) 978
(11.14) 982
(12.1) 1017
(12.2) 1019
(12.3) 1020
(12.4) 1020
(12.5) 1037
(12.6) 1041
(12.7) 1052
(12.8) 1088
(12.9) 1092
(13.1) 1123
(13.2) 1124
(13.3) 1125
(13.4) 1126
(13.5) 1126
(13.6) 1130
(13.7) 1141
(13.8) 1141
(13.9) 1153
(13.10) 1182
(13.11) 1185
(14.1) 1218
(14.2) 1218
(14.3) 1219
(14.4) 1219
(14.5) 1220
(14.6) 1222
(14.7) 1222
(14.8) 1227
(14.9) 1235

(14.10) 1235
(14.11) 1235
(14.12) 1235
(14.13) 1257
(14.14) 1264
(14.15) 1265
(14.16) 1270
(15.1) 1286
(15.2) 1286
(15.3) 1287
(15.4) 1287
(15.5) 1288
(15.6) 1289
(15.7) 1295
(15.8) 1295
(15.9) 1295
(15.10) 1296
(15.11) 1313
(15.12) 1314
(15.13) 1314
(15.14) 1314
(15.15) 1315
(15.16) 1315
(15.17) 1329
(15.18) 1329
(15.19) 1329
(15.20) 1330
(15.21) 1330
(15.22) 1336
(15.23) 1337
(15.24) 1342
(15.25) 1342
(15.26) 1353
(16.1) 1379
(16.2) 1381
(16.3) 1382
(16.4) 1383
(16.5) 1412
(16.6) 1429
(16.7) 1437
(16.8) 1437
(16.9) 1453
(16.10) 1454
(16.11) 1466
(16.12) 1504

Kata Pengantar

Setiap program yang pernah Anda pakai menyimpan sesuatu: daftar lagu, antrean pesan, peta jalan, katalog buku. Cara ia menyimpannya — berjajar dalam satu baris, bertaut satu ke berikutnya, tersusun sebagai pohon, tersebar ke dalam ember-ember bernomor — menentukan apakah program yang sama menjawab dalam sekejap atau membuat Anda menunggu. Pilihan itulah yang disebut struktur data, dan itulah seluruh pokok bahasan buku ini. Ingatlah kalimat itu, dan tidak ada satu pun isi halaman-halaman ini yang akan terasa sewenang-wenang.

Buku ini tumbuh dari mata kuliah Struktur Data (EF234201) di Departemen Teknik Informatika, Institut Teknologi Sepuluh Nopember (ITS), Surabaya, dan telah disusun ulang sehingga dapat berdiri sendiri sepenuhnya. Keenam belas babnya mengikuti irama satu semester yang berlangsung enam belas minggu, tetapi Anda tidak memerlukan kelas, dosen, atau dokumen lain untuk belajar darinya.

Untuk siapa buku ini ditulis

Anda dianggap belum tahu apa-apa. Bukan "belum tahu apa-apa tentang struktur data" — benar-benar belum tahu apa-apa. Jika Anda belum pernah menulis sebaris kode, belum pernah mengompilasi program, belum pernah bertanya-tanya di mana sebuah variabel sebenarnya tinggal, Andalah pembaca yang dibayangkan saat buku ini ditulis. Setiap istilah diperkenalkan dengan kata-kata sederhana sebelum dipakai, setiap struktur digambar keadaan demi keadaan sebelum diprogram, dan setiap program dalam buku ini sudah dijalankan sebelum keluarannya dicetak di sini.

Prinsip itu membuat buku ini tebal. Buku yang lebih tipis akan menganggap Anda sudah menguasai bagian-bagian yang sulit, dan buku semacam itu tidak berguna bagi orang yang belum menguasainya.

Satu gagasan, tiga bahasa

Struktur data adalah sebuah gagasan, bukan fitur milik satu bahasa, dan buku ini memperlakukannya demikian. Bahasa yang diajarkan adalah **C++**, karena C++ memperlihatkan memori, alamat, dan *pointer* yang menjadi bahan setiap struktur. Kemudian setiap program diberikan sekali lagi, lengkap dan siap dijalankan, dalam **Java** dan dalam **Python**, mengerjakan hal yang sama dan mencetak keluaran yang sama, baris demi baris. Melihat satu gagasan ditulis dengan tiga cara adalah jalan tercepat untuk memahami gagasan itu sendiri, bukan sekadar kebiasaan satu perkakas; dan itu memperlihatkan apa yang diam-diam dikerjakan pustaka Java dan Python untuk Anda — sehingga bahasa apa pun yang dipakai di tempat kerja Anda kelak, Anda sudah tahu apa yang ada di baliknya.

Isi buku ini

Bab 1 sampai 3 membangun fondasi: cara memori menyimpan data, penyegaran C++ dari nol, *array* secara mendalam beserta cicipan pertama kompleksitas, dan penyiapan lengkap VS Code untuk C++, Java, dan Python di Windows, macOS, dan Linux; lalu pencarian, dari pencarian sekuensial dan *sentinel*-nya sampai *binary search*, *interpolation search*, dan *jump search*; lalu

pengurutan, dari *bubble sort*, *selection sort*, dan *insertion sort*, melalui *Shell sort*, *merge sort*, dan *quicksort*, sampai *counting sort* dan *radix sort*, semuanya diukur pada data nyata. Bab 5 sampai 7 menambahkan struktur yang dapat tumbuh dan menyusut: *stack* dan *queue* beserta aplikasi klasiknya — mencocokkan tanda kurung, *undo* dan *redo*, infiks ke postfiks, *call stack* fungsi, *queue* sirkular, dan *deque*; *pointer*, referensi, dan fungsi, dengan memori dinamis, `nullptr`, `const`, dan *smart pointer*, serta apa yang dikerjakan referensi Java dan nama Python sebagai gantinya; dan *single linked list*, dengan *pointer head* dan *tail*, *dummy head node*, *list* sirkular, serta bukti bahwa tidak ada memori yang bocor. Bab 9 sampai 11 melangkah lebih jauh di sepanjang tautan: *double linked list* dan *linked list* sirkular dengan *sentinel*, sebuah *deque*, dan daftar "baru saja dipakai"; rekursi, yang diikuti bingkai demi bingkai pada *call stack* melalui faktorial, Fibonacci, algoritma Euclid, perpangkatan cepat, dan Menara Hanoi, sampai batas kedalaman di setiap bahasa; serta *tree* dan *binary search tree*, dengan setiap cara penelusuran, bentuk seimbang dan merosot, dan pandangan pertama pada *tree* yang menyeimbangkan diri. Bab 13 sampai 15 menuntaskan perjalanan: graf, dengan cara-cara merepresentasikannya, *breadth-first search* dan *depth-first search*, komponen terhubung, *topological sort*, dan Dijkstra yang ramah pemula; *heap* dan *priority queue*, dari *sift-up* dan *sift-down* sampai membangun *heap* dalam waktu linear, *heap sort*, dan *top-k*; serta *hashing*, dengan fungsi *hash*, tabrakan dan batas ulang tahun, *separate chaining*, *open addressing*, *load factor* dan *rehashing* yang diukur, dan tinjauan ke dalam `std::unordered_map`, `HashMap` Java, dan `dict` Python.

Bab 4, 8, 12, dan 16 adalah bab **Evaluasi dan Pembahasan**. Masing-masing menutup seperempat bagian buku dengan menyelesaikan persoalan utuh secara tuntas — setiap persoalan dinyatakan ulang, dirancang di kertas, diprogram, diuji, dijalankan, dan dijelaskan — lengkap dengan penyelesaian keliru yang menggoda beserta pembongkarannya, dan varian "uji diri" di akhir setiap persoalan. Bab terakhir di antaranya ditutup dengan memilih struktur data yang tepat untuk satu katalog dan mengukur setiap struktur dalam buku ini terhadapnya.

Di sepanjang buku, satu perpustakaan kecil tumbuh bab demi bab: **Pustaka Ceria**, perpustakaan kampus yang dibuka dengan delapan buku bertumpuk di lantai dan setahun kemudian memiliki delapan puluh ribu buku. Katalognya disimpan dalam *array*, dicari, dan diurutkan; kereta pengembalian bukunya adalah *stack* dan antrean katalogisasinya adalah *queue*; data bukunya ditautkan, disusun menjadi *search tree*, dan di-*hash* menurut judul; peta kampusnya adalah graf dan permintaan pemesanan bukunya adalah *heap*. Di sampingnya berjalan **Profitina Laundry**, usaha binatu kecil yang menjadi studi kasus praktik dalam buku-buku Profitina: pesanan hariannya diurutkan dari yang terberat, antrean *regular* dan *express*-nya, total mingguannya, menu layanannya sebagai *tree*, situs webnya sebagai graf, dan meja layanannya yang dibangun dari *hash map* dan *priority queue*. Setiap bab juga memiliki subbab *Profitina dalam Praktik* yang menunjukkan di mana gagasan yang sama muncul dalam produk nyata.

Setiap program dalam buku ini sudah dijalankan sebelum keluarannya dicetak. Program-program itu dikumpulkan, bab demi bab, dalam folder `Code` yang menyertai buku ini, dengan versi `cpp`, `java`, dan `python` berdampingan di setiap bab, dan README yang menjelaskan cara membangun serta menjalankan setiap program di terminal dan di VS Code, pada Windows, macOS, dan Linux. Setiap kode dalam buku ini adalah berkas di folder itu, karakter demi karakter.

Cara membacanya

Dengan editor terbuka dan kompilator siap. Ketikkan contohnya, jangan hanya menyalinnya; saat mengetik itulah kesalahan terjadi, dan dari kesalahan itulah pelajaran didapat. Sebelum

menjalankan sebuah program, gambarlah strukturnya di kertas — kotak-kotaknya, panah-panahnya, indeks-indeksnya — dan tebak apa yang akan dicetaknya; lalu jalankan dan bandingkan. Setelah sebuah contoh berjalan, rusakkan dengan sengaja lalu perhatikan seperti apa kegagalannya. Semua soal latihan di akhir tiap bab dilengkapi penyelesaian lengkap — gunakan sesudah usaha Anda sendiri, bukan sebagai penggantinya.

Gambar, tabel, kode, pseudokode, dan persamaan yang bernomor semuanya adalah tautan hidup: pada edisi elektronik, klik "Kode 5.3" atau "[4]" di bagian mana pun dan Anda langsung dibawa ke sana. Daftar Isi serta daftar-daftar di bagian depan buku bekerja dengan cara yang sama.

Tentang penulis

Irfan Subakti menggambarkan dirinya dalam satu baris — *Father | Educator | Entrepreneur | Bridging Tech & Nature: CS Lecturer × Crypto/Digital Business × AI/ML Specialist × Fruit Farmer | Storyteller* — ayah, pendidik, wirausahawan, penjembatan teknologi dan alam: dosen ilmu komputer, pelaku bisnis kripto dan digital, spesialis AI/ML, petani buah, dan pencerita. Setiap bagian dari baris itu ikut masuk ke dalam buku ini. Ia mengajar di Departemen Teknik Informatika ITS sejak 1998. Ia membimbing dan menguji tugas akhir mahasiswa sarjana, mengajar MLOps dalam program IBM SkillsBuild, dan membangun produk: ia mendirikan Profitina, platform visibilitas AI yang rekayasanya muncul di subbab "Profitina dalam Praktik" pada setiap bab. Jauh dari papan ketik, ia merawat kebun buah dan durian keluarga — di sanalah ia belajar bahwa perangkat lunak yang baik, seperti kebun yang baik, ditumbuhkan dengan sabar, satu bagian yang terawat demi satu bagian. Jiwa penceritanya adalah alasan buku ini memilih kisah ketimbang daftar, setiap kali sebuah kisah bisa mengantarkan maknanya.

Ucapan terima kasih

Kepada para mahasiswa yang mempelajari versi-versi awal materi ini dan mengajukan pertanyaan yang menyingkapkan di mana sebuah penjelasan masih tipis, serta kepada rekan-rekan di ITS yang bersama-sama mengajarkan dan menyempurnakan mata kuliah ini dari tahun ke tahun. Standar buku ini juga berutang pada buku-buku teks klasik algoritma dan struktur data serta dokumentasi resmi C++, Java, dan Python, yang dikutip di setiap tempat penggunaannya.

Surabaya, September 2026



Irfan Subakti · yifana@gmail.com



profitina.com

Bab 1

Fondasi: Penyegaran C++, Memori, dan Array

Pada akhir bab ini Anda akan sanggup menjelaskan apa itu struktur data dan mengapa pilihan atasnya dapat membuat program yang sama seribu kali lebih cepat atau lebih lambat; menguraikan bagaimana memori komputer menyimpan data sebagai byte-byte bernomor, bagaimana sebuah tipe memberi makna pada byte-byte itu, dan mengapa sebagian variabel tinggal di *stack* sementara yang lain tinggal di *heap*; membaca dan menulis C++ yang dipakai di sepanjang buku ini, termasuk tipe, variabel, masukan dan keluaran, percabangan, perulangan, fungsi, *struct*, dan operator alamat `&`; mendeklarasikan, mengisi, menelusuri, dan menjaga batas *array* (larik) satu dan dua dimensi, menghitung alamat persis setiap elemen dengan rumus *row-major*, serta memilih antara array biasa, `std::array`, dan `std::vector`, antara array Java dan `ArrayList`, dan antara `list` Python dan modul `array`; menerangkan, dengan menghitung langkah, mengapa membaca sebuah elemen array berbiaya $O(1)$ sedangkan menyisipkannya berbiaya $O(n)$; serta memasang, di Windows, macOS, atau Linux, sebuah meja kerja lengkap berupa Visual Studio Code dengan C++, Java, dan Python, lalu membuktikan bahwa semuanya bekerja. Setiap program dalam bab ini disajikan dalam ketiga bahasa itu, dan setiap keluaran yang tercetak di sini adalah keluaran yang benar-benar dihasilkan program.

1.1 Titik Berangkat: Perpustakaan Tanpa Rak

1.1.1 Pustaka Ceria

Bayangkan sebuah perpustakaan kecil yang riang di sebuah kampus. Namanya **Pustaka Ceria**, dan pada pagi pertamanya ia memiliki delapan buku, sebuah meja, seorang pustakawan, dan tidak ada yang lain. Belum ada rak. Buku-buku itu datang dalam sebuah kardus, dan sang pustakawan, yang masih baru, begitu saja menumpuknya di lantai menurut urutan ia mengeluarkannya.

Pukul delapan pagi seorang mahasiswa menanyakan *Filosofi Kopi*. Pustakawan memandang tumpukan itu, mengangkat buku pertama, membaca sampulnya, menyisihkannya, mengangkat buku kedua, dan menemukan buku yang dicari pada percobaan ketiga. Mudah. Dengan delapan buku, cara apa pun berhasil.

Setahun kemudian Pustaka Ceria memiliki delapan puluh ribu buku, dan cara menumpuk itu telah menjadi bencana. Seorang mahasiswa yang menanyakan satu judul mungkin harus menunggu sementara pustakawan mengangkat dan membaca rata-rata empat puluh ribu sampul. Tidak ada yang menjadi lebih lambat dalam membaca sampul; perpustakaannya saja yang tumbuh, sedangkan *cara menyimpan bukunya* tidak ikut tumbuh. Maka perpustakaan itu membeli rak, memberinya nomor, mengurutkan buku menurut kode, dan menyimpan katalog kartu. Kini pustakawan dapat berjalan langsung ke rak yang tepat, melirik sepanjang rak itu, dan kembali dalam semenit.

Tidak ada yang berubah pada buku-bukunya. Delapan puluh ribu judul yang sama, pengarang yang sama, halaman yang sama. Yang berubah adalah **penataannya**: di mana setiap buku disimpan, apa yang dicatat tentang letaknya, dan apa yang boleh diandaikan pustakawan ketika ia pergi mencari. Penataan itulah yang oleh buku ini disebut *struktur data*, dan Pustaka Ceria akan menemani kita di sepanjang keenam belas bab. Dalam bab ini ia mendapat rak sungguhan yang pertama, yaitu sebuah array. Dalam bab-bab berikutnya ia mendapat tumpukan buku kembalian yang menunggu dikembalikan ke rak, antrean pemesanan, pohon kategori, graf buku-buku yang sering dipinjam bersama, dan tabel hash yang menemukan buku mana pun dari kodenya dalam satu langkah.

1.1.2 Apa itu struktur data

Sebuah program adalah sekumpulan instruksi yang bekerja pada data: bilangan, nama, harga, tanggal, gambar. Selama program hanya menangani satu atau dua nilai, ia cukup memberi masing-masing sebuah variabel. Begitu ia menangani *banyak* nilai sejenis, seperti buku-buku sebuah perpustakaan, mahasiswa sebuah kelas, atau pesanan sebuah toko, ia memerlukan cara untuk menyimpannya bersama, menemukan salah satunya, menambah satu, membuang satu, dan mengunjungi semuanya menurut suatu urutan. Memutuskan *bagaimana* nilai-nilai itu disimpan adalah merancang sebuah struktur data.

> **DEFINISI.** *Struktur data* adalah cara menyimpan dan menata sekumpulan data di dalam memori komputer, bersama operasi-operasi yang dapat dilakukan terhadapnya, sehingga operasi-operasi itu dapat dijalankan dengan benar dan efisien. Lima operasi yang menjadi ukuran hampir setiap struktur data adalah **penyisipan** (*insert*, menambah butir), **penghapusan** (*delete*, membuang butir), **pencarian** (*search*, menemukan butir, atau mengetahui bahwa butir itu tidak ada), **penelusuran** (*traverse*, mengunjungi setiap butir tepat sekali), dan **pengurutan** (*sort*, menyusun butir-butir menurut urutan) [1].

Di dalam definisi ini tersembunyi dua gagasan, dan ada baiknya keduanya segera dipisahkan.

Yang pertama adalah *antarmuka* (*interface*): daftar operasi, apa yang dijanjikan tiap operasi, dan sama sekali tidak menyebut bagaimana janji itu ditepati. “Saya dapat memberi Anda buku berkode B1005, atau mengatakan bahwa buku itu tidak ada di sini” adalah sebuah antarmuka. Ilmuwan komputer menyebut tipe data yang diuraikan hanya lewat antarmukanya sebagai **tipe data abstrak** (*abstract data type*, ADT).

Yang kedua adalah *implementasi*: susunan konkret byte di dalam memori dan langkah-langkah konkret yang menjalankan setiap operasi. “Buku-buku berjajar menurut urutan kode, dan saya menemukan satu buku dengan membuka deretan itu di tengah lalu memutuskan separuh mana yang dipertahankan” adalah satu implementasi antarmuka tadi. Begitu pula “buku-buku ditaruh dalam delapan kotak yang dipilih menurut angka terakhir kodenya, dan saya hanya mencari di kotak yang tepat”. Keduanya menepati janji yang sama, dengan biaya yang sangat berbeda.

Satu uji sederhana dalam bahasa sehari-hari: antarmuka adalah apa yang perlu diketahui *pengunjung* perpustakaan; implementasi adalah apa yang perlu diketahui *pustakawan*. Struktur data yang baik membuat pengunjung dapat melupakan implementasinya sama sekali, dan membuat pustakawan dapat mengubah implementasi itu tanpa disadari pengunjung.

1.1.3 Mengapa pilihan itu penting

“Tetapi komputer itu cepat,” mungkin seorang pembaca menyanggah. “Mengapa repot soal cara, kalau mesin dapat membaca sejuta sampul dalam sekejap mata?” Jawabannya: banyaknya

langkah sebuah cara tidak tumbuh dengan laju yang sama untuk setiap cara, dan ketika koleksinya besar, perbedaan *laju* itu menenggelamkan perbedaan kecepatan mesin apa pun.

Andaikan Pustaka Ceria menyimpan n buku. Tiga cara menemukan satu buku dibandingkan dalam **Tabel 1.1**. Cara pertama membaca setiap sampul, dalam kasus terburuk semua n sampul. Cara kedua menjaga buku tetap terurut dan berulang kali membelah dua bagian yang masih layak dicari, sehingga memerlukan sekitar $\log_2 n$ kali lihat: banyaknya kali n dapat dibelah dua sampai tersisa satu buku. Cara ketiga adalah tabel hash, yang menghitung letak buku dari kodenya dan rata-rata hanya melihat sekali. Bab 2 membangun dua cara pertama, dan Bab 15 membangun yang ketiga.

Tabel 1.1 Banyaknya kali lihat dalam kasus terburuk untuk menemukan satu buku di antara n buku, menurut tiga cara menyimpan buku.

Buku n	Baca setiap sampul (n kali lihat)	Belah dua tiap kali ($\approx\log_2 n$ kali lihat)	Hitung letaknya (sekitar 1 kali lihat)
8	8	3	1
1.000	1.000	10	1
80.000	80.000	17	1
1.000.000	1.000.000	20	1
1.000.000.000	1.000.000.000	30	1

Bacalah baris terakhir perlahan-lahan. Semiliar buku, dan cara kedua memerlukan tiga puluh kali lihat, sedangkan cara pertama semiliar. Jika setiap kali lihat memakan satu mikrodetik, cara pertama akan memakan sekitar tujuh belas menit dan cara kedua tiga puluh mikrodetik. Tidak ada komputer yang lebih cepat yang mampu menutup jurang seperti itu: mesin yang seratus kali lebih cepat mengubah tujuh belas menit menjadi sepuluh detik, yang masih tiga ratus ribu kali lebih lambat daripada cara yang lebih baik pada mesin lama. *Memilih struktur data yang tepat biasanya lebih berharga daripada membeli komputer tercepat.* Kalimat itulah alasan buku ini ada.

1.1.4 Struktur data di balik setiap halaman web

Mudah untuk mengira bahwa struktur data hanya hidup di buku teks. Kenyataannya, Anda memakai ratusan struktur data setiap kali membuka sebuah halaman web, dan ada gunanya berhenti sejenak untuk melihat di mana saja, karena contoh ini membuat kata yang abstrak itu menjadi nyata.

Pertama, dua istilah yang sering tertukar. **Internet** adalah jaringan dari jaringan-jaringan yang sangat besar: jutaan jaringan komputer yang lebih kecil, milik universitas, perusahaan, penyedia telekomunikasi, dan rumah tangga, yang saling terhubung dan berbicara dengan satu keluarga aturan bersama untuk memindahkan data yang disebut **TCP/IP** [2]. **World Wide Web** hanyalah salah satu layanan yang berjalan di atas Internet. Web adalah layanan tempat sebuah program bernama *peramban web* (*web browser*) meminta dokumen kepada program bernama *server web*, dengan buku aturan bernama **HTTP**, *HyperText Transfer Protocol*, lalu menampilkan dokumen itu kepada manusia. Surel, transfer berkas, obrolan, panggilan video, gim daring, musik *streaming*, dan aplikasi di ponsel adalah layanan lain yang memakai Internet yang sama tanpa menjadi web. Jadi jawaban atas pertanyaan lama "apakah Internet dan web itu sama?" adalah tidak: web hanyalah salah satu penumpang di jalan-jalan Internet.

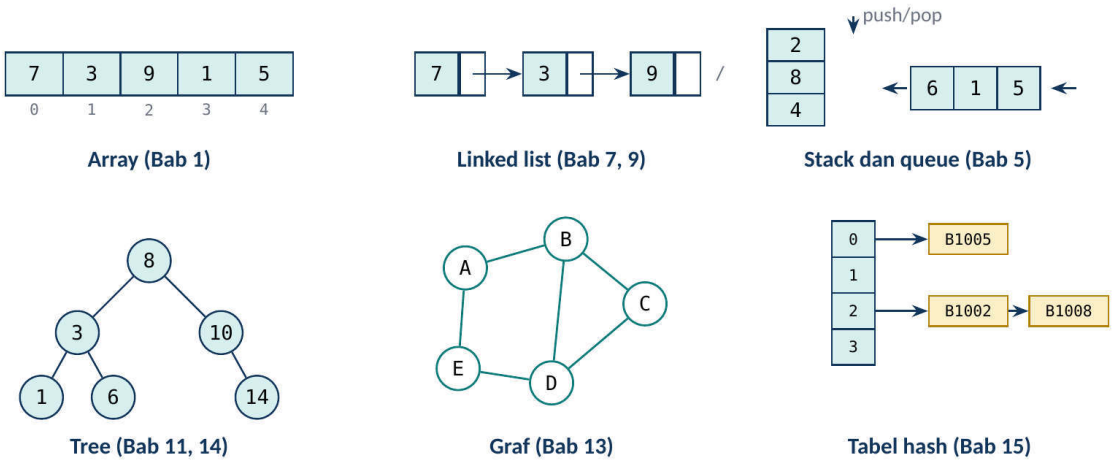
Sekarang ikutilah satu permintaan. Anda mengetikkan alamat sebuah halaman ke peramban.

1. Peramban mencari nama situs itu di **cache**-nya, sebuah tabel berisi nama-nama yang baru saja dilihat beserta alamat numeriknya. Tabel itu adalah *tabel hash*, struktur Bab 15, karena ia harus menjawab "pernahkah aku melihat nama ini?" dalam satu langkah.
2. Jika nama itu baru, pertanyaannya berjalan ke serangkaian *name server*, yang wilayah tanggung jawabnya membentuk sebuah *pohon* (*tree*), struktur Bab 11.
3. Permintaan itu menyeberangi Internet melalui *router*, yang masing-masing menyimpan permintaan-permintaan yang menunggu jalur keluar yang sama dalam sebuah *antrean* (*queue*), struktur Bab 5, sehingga yang pertama tiba adalah yang pertama keluar. Router memilih jalurnya dengan menghitung pada sebuah *graf* sambungan, struktur Bab 13.
4. Server menerima permintaan itu dan membaca data halaman dari berkas-berkas yang blok-bloknya didaftar dalam *array* dan diindeks oleh *pohon*.
5. Peramban menerima halaman itu dan membangun dari teksnya sebuah *pohon* elemen, lalu menelusurinya dengan bantuan *stack* (tumpukan) untuk menata dan menggambar setiap elemen di layar.

Lima langkah dan paling sedikit enam struktur data, masing-masing dipilih karena membuat operasi yang diperlukan langkah itu menjadi murah. Semuanya ada di buku ini.

1.1.5 Enam keluarga, dan peta jalan buku ini

Seberagam apa pun, struktur data yang dipakai sehari-hari termasuk dalam segelintir keluarga. **Gambar 1.1** memperlihatkan keenam keluarga yang dipelajari buku ini, masing-masing digambar dalam bentuk yang akan dipakai bab-bab berikutnya.



Gambar 1.1 Enam keluarga struktur data dalam buku ini: array, *linked list*, *stack* dan *queue*, *tree* (bersama *heap*), graf, dan tabel hash, masing-masing diberi label bab yang membangunnya.

- **Array** menyimpan butir-butirnya berdampingan dalam satu blok memori, masing-masing dapat dicapai lewat nomor posisinya. Array adalah pokok bahasan bab ini dan fondasi bagi hampir semua yang lain.
- **Linked list** (senarai berantai) menyimpan setiap butir dalam kotak kecilnya sendiri yang juga mencatat di mana kotak berikutnya berada, sehingga butir dapat ditambah dan dibuang tanpa memindahkan butir lain.
- **Stack** (tumpukan) hanya mengizinkan penambahan dan pengambilan di satu ujung, seperti setumpuk piring; **queue** (antrean) menambah di satu ujung dan mengambil di ujung lain, seperti barisan di depan loket.

- **Tree** (pohon) menata butir-butir bertingkat di bawah satu akar, seperti silsilah keluarga atau folder di sebuah diska; **heap** adalah pohon yang dijaga dalam urutan khusus sehingga butir terbesar atau terkecil selalu berada di puncak.
- **Graf** mencatat butir-butir dan hubungan di antara mereka, seperti kota-kota dan jalan-jalan, atau halaman-halaman web dan tautan-tautannya.
- **Tabel hash** menghitung di mana sebuah butir seharusnya berada dari kunci butir itu sendiri, sehingga sering kali dapat menemukannya dalam satu langkah.

Tabel 1.2 menjabarkan keenam belas bab. Setiap bab keempat, yang bertanda *Evaluasi dan Pembahasan*, berhenti sejenak untuk mengerjakan serangkaian soal tentang tiga bab sebelumnya, secara terperinci.

Tabel 1.2 Keenam belas bab buku ini dan apa yang ditambahkan masing-masing.

Bab	Judul	Yang ditambahkannya
1	Fondasi: Penyegaran C++, Memori, dan Array	Memori, C++, array, cicipan pertama kompleksitas, meja kerja
2	Pencarian	Pencarian sekuensial, sentinel, biner, dan interpolasi; notasi *O* secara lengkap
3	Pengurutan	Pengurutan-pengurutan klasik, dari *bubble sort* sampai *quicksort*, *merge sort*, dan *heapsort*
4	Evaluasi dan Pembahasan I	Soal-soal terbahas tentang Bab 1 sampai 3
5	Stack dan Queue	*Last-in first-out* dan *first-in first-out*, di atas array
6	Pointer, Referensi, dan Fungsi	Kisah pointer yang dimulai dalam bab ini, dituntaskan
7	Single Linked List	Simpul dan tautan; penyisipan tanpa menggeser
8	Evaluasi dan Pembahasan II	Soal-soal terbahas tentang Bab 5 sampai 7
9	Double Linked List dan Linked List Sirkular	Tautan dua arah dan cincin
10	Rekursi	Fungsi yang memanggil dirinya sendiri
11	Tree dan Binary Search Tree	Hierarki, penelusuran, pencarian terurut
12	Evaluasi dan Pembahasan III	Soal-soal terbahas tentang Bab 9 sampai 11
13	Graf	Matriks dan list ketetanggaan, penelusuran melebar dan mendalam
14	Heap dan Priority Queue	Sifat heap dan operasi-operasinya
15	Hashing dan Tabel Hash	Fungsi hash, tabrakan, *chaining*, dan *open addressing*
16	Evaluasi dan Pembahasan IV	Soal-soal terbahas tentang Bab 13 sampai 15 dan seluruh buku

1.2 Tiga Bahasa, Satu Gagasan

1.2.1 Mengapa C++ lebih dahulu, dan mengapa juga Java dan Python

Buku ini mengajarkan struktur data dalam **C++**, dan menyajikan setiap program sekali lagi dalam **Java** dan **Python**, lengkap dan siap dijalankan, mencetak hasil yang sama sejauh bahasa-bahasa itu memungkinkan. Masing-masing dari ketiganya punya alasan yang baik.

C++ menjadi bahasa teks utama karena ia tidak menyembunyikan apa pun. Ketika sebuah program C++ mendeklarasikan array berisi delapan buku, kedelapan buku itu benar-benar berjajar di memori, dan program dapat menanyakan di mana masing-masing bermula. Ketika program C++ meminta memori selagi berjalan, ia juga wajib mengembalikannya. Pada dasarnya, mata pelajaran struktur data adalah pelajaran tentang bagaimana data duduk di memori, dan C++

membiarkan kita memandang memori itu secara langsung. Sifat yang sama membuat C++ menjadi bahasa sistem operasi, peramban, basis data, mesin gim, dan bagian-bagian dalam yang cepat dari banyak sistem lain.

Java dan Python adalah bahasa tempat sebagian besar perangkat lunak dunia yang lain ditulis, dan masing-masing mengambil keputusan berbeda tentang memori. Java menaruh setiap objek di *heap* dan membebaskannya secara otomatis, tetapi masih membedakan bilangan biasa dari objek. Python menjadikan *segala sesuatu* objek dan setiap variabel sebuah nama yang merujuk ke objek. Melihat struktur data yang sama dalam ketiga bahasa memperlihatkan bagian mana yang merupakan gagasannya sendiri dan bagian mana yang sekadar kebiasaan satu bahasa.

Ketiganya termasuk bahasa yang paling banyak dipakai di dunia. **Tabel 1.3** memperlihatkan puncak Indeks Komunitas Pemrograman TIOBE edisi September 2026, yang memperkirakan kepopuleran sebuah bahasa dari banyaknya insinyur, kursus, dan vendor yang ditemukan mesin pencari untuknya [3], [4].

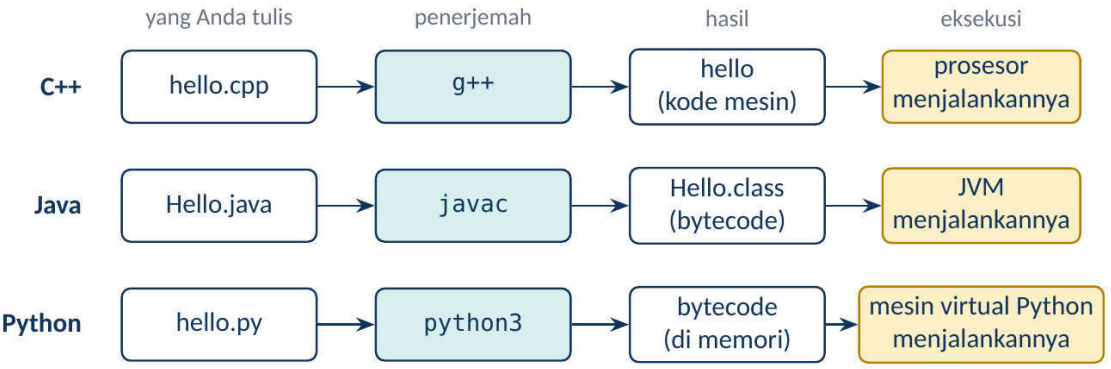
Tabel 1.3 Sepuluh bahasa pemrograman terpopuler menurut Indeks TIOBE September 2026.

Peringkat	Bahasa	Nilai
1	Python	17,76%
2	C	10,28%
3	C++	8,67%
4	Java	7,54%
5	C#	4,22%
6	JavaScript	2,76%
7	Visual Basic	2,55%
8	SQL	2,16%
9	R	1,69%
10	Rust	1,34%

Indeks kepopuleran bukanlah peringkat mutu, dan TIOBE sendiri menegaskan bahwa indeksnya tidak mengukur bahasa "terbaik" maupun bahasa yang paling banyak dipakai untuk menulis kode [3]. Indeks itu memberi tahu kita bahwa ketiga bahasa buku ini, bersama C, bahasa asal C++, berada di puncak daftar, dan sudah bertahun-tahun berada di sana.

1.2.2 Bagaimana program menjadi sesuatu yang berjalan

Prosesor komputer hanya memahami *kode mesin*-nya sendiri: deretan panjang bilangan, masing-masing sebuah instruksi seperti "jumlahkan dua register ini" atau "muat empat byte dari alamat ini". Tidak ada lagi orang yang menulis kode mesin dengan tangan. Kita menulis *kode sumber* dalam bahasa yang dapat dibaca manusia, lalu sebuah program menerjemahkannya. Ketiga bahasa buku ini menerjemahkan dengan tiga cara berbeda, yang digambar berdampingan dalam **Gambar 1.2**.



Gambar 1.2 Dari berkas sumber ke program yang berjalan: C++ dikompilasi menjadi kode mesin, Java menjadi *bytecode* untuk Java Virtual Machine, dan Python menjadi *bytecode* yang langsung dijalankan mesin virtualnya sendiri.

- **C++** *dikompilasi*. Kompilatornya, `g++` atau `clang++`, membaca seluruh `hello.cpp`, memeriksanya, lalu menulis berkas baru, yaitu *berkas eksekusi* (*executable*), yang penuh kode mesin. Yang Anda jalankan adalah berkas eksekusi itu; kode sumbernya tidak diperlukan lagi. Kompilasi memakan sedikit waktu, tetapi hasilnya berjalan dengan kecepatan penuh prosesor.
- **Java** juga dikompilasi, tetapi tidak menjadi kode mesin. Kompilator `javac` mengubah `Hello.java` menjadi `Hello.class`, yang berisi *bytecode*, kode mesin untuk sebuah prosesor khayalan bernama **Java Virtual Machine** (JVM). Perintah `java` menyalakan sebuah JVM, yang menjalankan *bytecode* itu dan, untuk bagian yang sering dijalankan, mengompilasinya lebih lanjut menjadi kode mesin sungguhan selagi program berjalan.
- **Python** biasanya disebut *diinterpretasikan*. Perintah `python3` membaca `hello.py`, menerjemahkannya menjadi *bytecode* di memori, dan seketika menjalankan *bytecode* itu pada mesin virtual Python. Tidak ada langkah terpisah dan tidak ada berkas terpisah untuk dijalankan, sehingga Python sangat nyaman untuk mencoba-coba, dan biasanya lebih lambat daripada C++ atau Java untuk komputasi berat.

1.2.3 Bagaimana kode disajikan

Setiap program dalam bab ini berada di folder `Chapter01` kode pendamping buku, dalam tiga subfolder, `cpp`, `java`, dan `python`, disertai sebuah `README.md` yang menjelaskan cara membangun dan menjalankan setiap program di Windows, macOS, dan Linux, dari terminal maupun dari VS Code. Subbab 1.7 memasang semua yang Anda perlukan. Jika Anda belum pernah memasang kompilator, Anda mungkin ingin membaca Subbab 1.7 lebih dahulu lalu kembali ke sini; semua yang ada di antara sini dan sana ditulis agar dapat dibaca, dipahami, dan dicocokkan dengan keluaran tercetak bahkan sebelum Anda menjalankannya.

Program C++ dikompilasi dengan `g++ -std=c++17 -O2 -Wall -Wextra` dan tidak menghasilkan satu peringatan pun; program Java dengan `javac` dari JDK 21, dan juga berjalan di JDK 25 yang terkini; program Python dengan Python 3.11, dan juga berjalan di Python 3.14 yang terkini. Bila ketiga bahasa mencetak hal yang persis sama, keluarannya dicetak sekali. Bila berbeda, karena bahasanya sendiri memang berbeda, setiap keluaran dicetak dan perbedaannya dijelaskan.

1.3 Bagaimana Memori Menyimpan Data

Sebelum dapat berbicara dengan masuk akal tentang array, atau tentang struktur data apa pun, kita memerlukan gambaran yang jelas tentang tempat struktur-struktur itu dibangun: memori

komputer. Gambarannya sederhana, dan begitu Anda memilikinya, banyak isi buku ini menjadi akal sehat.

1.3.1 Bit dan byte

Pada tingkat paling dasar, komputer menyimpan segala sesuatu sebagai **bit**. Sebuah bit adalah sakelar mungil yang mati atau menyala, ditulis 0 atau 1. Satu bit saja hanya dapat membedakan dua hal: ya atau tidak, lampu mati atau lampu menyala. Dua bit bersama dapat berada dalam empat keadaan berbeda, 00, 01, 10, dan 11; tiga bit dalam delapan keadaan. Setiap bit tambahan menggandakan banyaknya keadaan, karena setiap keadaan lama kini dapat diikuti 0 atau 1. Jadi sekelompok k bit dapat berada dalam

$$2^k \text{ keadaan berbeda} \tag{1.1}$$

dan Persamaan (1.1) adalah alasan bilangan 2 beserta pangkat-pangkatnya muncul di mana-mana dalam komputasi.

Bit dikelompokkan per delapan. Sekelompok delapan bit adalah sebuah **byte**, dan menurut Persamaan (1.1) sebuah byte mempunyai $2^8 = 256$ keadaan. Kita dapat memakai 256 keadaan itu untuk mewakili bilangan bulat 0 sampai 255: bit-bitnya dibaca sebagai bilangan berbasis 2, sama seperti angka-angka 407 dibaca dalam basis 10 sebagai $4 \cdot 100 + 0 \cdot 10 + 7 \cdot 1$. Misalnya byte 00000111 bernilai $4 + 2 + 1 = 7$, dan 11101010 bernilai $128 + 64 + 32 + 8 + 2 = 234$.

Menulis delapan angka biner itu melelahkan, sehingga pemrogram menulis byte dalam **heksadesimal**, basis 16. Keenam belas angkanya adalah 0 sampai 9 lalu `a` sampai `f` untuk sepuluh sampai lima belas. Empat bit membentuk tepat satu angka heksadesimal, sehingga sebuah byte selalu tepat dua angka: 00000111 adalah `07`, dan 11101010, yang terbelah menjadi $1110 = 14$ dan $1010 = 10$, adalah `ea`. Di C++, Java, dan Python bilangan heksadesimal ditulis dengan awalan `0x`, seperti `0xEA`.

1.3.2 Alamat

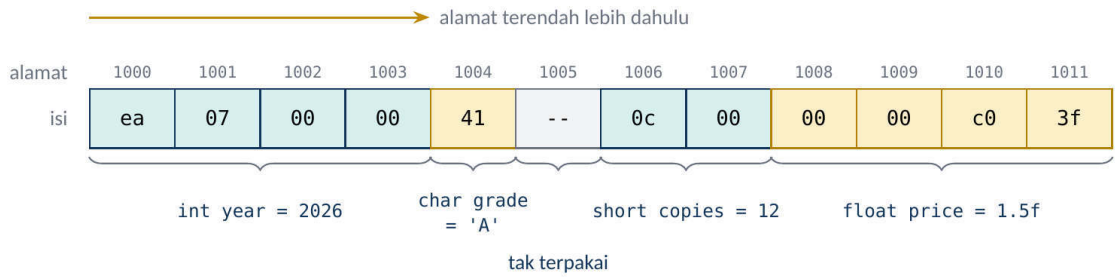
Memori komputer yang sedang berjalan adalah satu deretan byte yang sangat panjang, bermiliar-miliar jumlahnya. Setiap byte mempunyai nomor, yaitu **alamat**-nya, yang dihitung naik dari nol, persis seperti rumah-rumah di sepanjang jalan yang dinomori dari satu ujung. Ketika program berkata "bacalah byte di alamat 1004", perangkat keras memori langsung menuju rumah nomor 1004; ia tidak perlu melewati rumah-rumah sebelumnya. Fakta tunggal ini, bahwa alamat mana pun dapat dicapai dalam waktu singkat yang sama, disebut **akses acak** (*random access*), dan inilah yang membuat array, struktur bab ini, begitu cepat.

Nilai yang memerlukan lebih dari satu byte, seperti bilangan bulat yang dapat lebih besar dari 255, disimpan dalam beberapa byte yang *berurutan*, dan alamat nilai itu adalah alamat byte pertamanya. Dalam gambar-gambar buku ini kita memakai alamat kecil yang ramah seperti 1000; alamat sungguhan adalah bilangan yang jauh lebih besar, biasanya dicetak dalam heksadesimal, seperti yang akan diperlihatkan program-program di bawah.

1.3.3 Tipe memberi makna pada byte

Byte sendiri tidak bermakna apa-apa. Keempat byte `ea 07 00 00` dapat berupa sebuah bilangan, empat huruf, sebagian gambar, atau sebagian instruksi mesin. Yang menentukan adalah **tipe** yang dinyatakan program untuknya. Tipe menetapkan dua hal: *berapa banyak* byte yang ditempati nilai itu, dan *bagaimana* byte-byte itu dibaca.

Gambar 1.3 memperlihatkan dua belas byte berurutan yang menyimpan empat variabel bertipe berbeda.



Gambar 1.3 Dua belas byte memori yang menyimpan sebuah `int`, sebuah `char`, sebuah `short`, dan sebuah `float`; setiap tipe menentukan berapa byte yang dipakai sebuah nilai dan bagaimana byte itu dibaca, dan byte beralamat terendah datang lebih dahulu.

- `int year = 2026` menempati empat byte, di alamat 1000 sampai 1003. Ditulis dalam heksadesimal, 2026 adalah `0x000007EA`, yaitu keempat byte `00 00 07 ea` dari yang paling bermakna sampai yang paling tidak bermakna. Kebanyakan komputer masa kini, termasuk setiap prosesor Intel, AMD, dan Apple, menyimpan **byte yang paling tidak bermakna lebih dahulu**, di alamat terendah; urutan ini disebut **little-endian**, dan itulah sebabnya gambar memperlihatkan `ea 07 00 00`. Urutan kebalikannya, byte paling bermakna lebih dahulu, adalah **big-endian**; urutan ini dipakai dalam protokol jaringan dan dalam format berkas milik Java sendiri [5].
- `char grade = 'A'` menempati satu byte. Huruf A disimpan sebagai kode karakternya, 65, yaitu `0x41`.
- Byte di alamat 1005 tidak terpakai. Pada mesin ini sebuah `short` harus bermula di alamat genap, sehingga kompilator melewatkan satu byte. Celah ini disebut **padding** (pengisi), dan kita akan bertemu lagi dengannya di Subbab 1.5.3.
- `short copies = 12` menempati dua byte, `0c 00`.
- `float price = 1.5f` menempati empat byte, tetapi `float` dibaca dengan cara yang sama sekali lain daripada `int`: sebagian bitnya menyatakan angka-angkanya dan sebagian lagi menyatakan letak titik desimalnya. Keempat byte yang sama, bila dibaca sebagai `int`, akan bermakna 1.069.547.520.

Butir terakhir itu layak diulang dengan kata-kata sederhana: *memori tidak tahu apa yang disimpannya; hanya program yang tahu, melalui tipe-tipe yang dipakainya.*

1.3.4 Melihatnya sendiri

Kode 1.1 menanyakan kepada mesin berapa byte yang ditempati setiap tipe dasar C++, memeriksa keempat byte `int` 2026 satu per satu, dan mencetak alamat dua variabel.

Kode 1.1 `memory_basics.cpp`: ukuran tipe-tipe dasar, byte-byte sebuah `int`, dan sebuah alamat di *stack* serta di *heap*.

```cpp
// memory_basics.cpp - ukuran, byte, dan tempat variabel berada
#include <iomanip>
#include <iostream>
using namespace std;

int main() {
    cout << "Sizes on this machine, in bytes:\n";
    cout << "  char      " << sizeof(char) << '\n';
    cout << "  bool      " << sizeof(bool) << '\n';
    cout << "  short     " << sizeof(short) << '\n';
    cout << "  int       " << sizeof(int) << '\n';
    cout << "  long long " << sizeof(long long) << '\n';
    cout << "  float     " << sizeof(float) << '\n';
    cout << "  double    " << sizeof(double) << '\n';

    int year = 2026;                        // 2026 = 0x000007EA
    const unsigned char* b = reinterpret_cast<const unsigned char*>(&year);
    cout << "\nThe int " << year << " occupies these bytes, lowest address first:";
    for (size_t i = 0; i < sizeof year; ++i) {
        cout << ' ' << hex << setw(2) << setfill('0') << int(b[i]);
    }
    cout << dec << '\n';
    cout << "So this machine is " << (b[0] == 0xEA ? "little" : "big")
         << "-endian.\n";

    int onStack = 7;                        // variabel lokal: di stack
    int* onHeap = new int(8);               // diminta saat program berjalan: di heap
    cout << "\nThe local variable lives at " << &onStack << " (stack)\n";
    cout << "The new int lives at        " << onHeap << " (heap)\n";
    delete onHeap;                          // kembalikan memori heap
    return 0;
}
```

```
Sizes on this machine, in bytes:
  char      1
  bool      1
  short     2
  int       4
  long long 8
  float     4
  double    8

The int 2026 occupies these bytes, lowest address first: ea 07 00 00
So this machine is little-endian.

The local variable lives at 0x7ffd287f05e4 (stack)
The new int lives at        0x55a6fbee52c0 (heap)
```

Program ini memakai empat gagasan yang dijelaskan tuntas dalam penyegaran Subbab 1.4, jadi di sini cukup tur singkat. Operator `sizeof` memberikan banyaknya byte yang ditempati sebuah tipe atau variabel. Ekspresi `&year` adalah *alamat* `year`, dan `reinterpret_cast<const unsigned char*>` menyuruh kompilator membiarkan kita melihat byte-byte di sana satu per satu. Manipulator `hex`, `setw(2)`, dan `setfill('0')` mencetak setiap byte sebagai dua angka heksadesimal. Paragraf terakhir menaruh satu `int` di *stack* dan satu di *heap*, yang dijelaskan Subbab 1.3.5, lalu mencetak kedua alamatnya. Alamat Anda akan berbeda, dan berubah dari satu kali jalan ke kali berikutnya, karena sistem operasi modern sengaja menempatkan memori program di posisi acak setiap kali program dimulai, sehingga beberapa jenis serangan menjadi jauh lebih sulit.

Ukuran yang tercetak adalah ukuran pada mesin Linux, Windows, atau macOS 64-bit. Standar C++ hanya menetapkan bahwa `char` berukuran satu byte dan bahwa setiap tipe dalam daftar `short`, `int`, `long long` paling sedikit sama besar dengan tipe sebelumnya; ukuran persisnya dipilih

kompilator untuk tiap mesin. Itulah sebabnya buku ini selalu bertanya kepada `sizeof` alih-alih mengandaikan.

Java menetapkan ukuran tipe-tipe dasarnya di dalam bahasa itu sendiri, sama pada setiap mesin, dan sama sekali tidak membiarkan program melihat alamat. **Kode 1.2** mencetak apa yang diizinkan Java untuk kita lihat.

Kode 1.2 `MemoryBasics.java`: ukuran tipe Java yang tetap dan byte-byte sebuah `int` dalam urutan milik mesin itu sendiri.

```java
// MemoryBasics.java - ukuran, byte, dan tempat nilai berada
import java.nio.ByteBuffer;
import java.nio.ByteOrder;

public class MemoryBasics {
    public static void main(String[] args) {
        System.out.println("Sizes fixed by the Java language, in bytes:");
        System.out.println("  byte      " + Byte.BYTES);
        System.out.println("  char      " + Character.BYTES);
        System.out.println("  short     " + Short.BYTES);
        System.out.println("  int       " + Integer.BYTES);
        System.out.println("  long      " + Long.BYTES);
        System.out.println("  float     " + Float.BYTES);
        System.out.println("  double    " + Double.BYTES);

        int year = 2026;                      // 2026 = 0x000007EA
        byte[] b = ByteBuffer.allocate(Integer.BYTES)
                .order(ByteOrder.nativeOrder()).putInt(year).array();
        StringBuilder sb = new StringBuilder();
        for (byte x : b) {
            sb.append(String.format(" %02x", x & 0xFF));
        }
        System.out.println("\nThe int " + year + " in this machine's byte order:"
                + sb);
        boolean little = ByteOrder.nativeOrder() == ByteOrder.LITTLE_ENDIAN;
        System.out.println("So this machine is " + (little ? "little" : "big")
                + "-endian.");

        int onStack = 7;                      // variabel lokal: di frame stack
        int[] onHeap = new int[] {8};         // array adalah objek heap
        System.out.println("\nThe local variable holds " + onStack + " (stack)");
        System.out.println("The array object holds   " + onHeap[0] + " (heap)");
        System.out.println("Java never shows addresses;"
                + " the garbage collector frees the heap.");
    }
}
```

```
Sizes fixed by the Java language, in bytes:
  byte      1
  char      2
  short     2
  int       4
  long      8
  float     4
  double    8

The int 2026 in this machine's byte order: ea 07 00 00
So this machine is little-endian.

The local variable holds 7 (stack)
The array object holds   8 (heap)
Java never shows addresses; the garbage collector frees the heap.
```

Perhatikan satu perbedaan dalam tabelnya: `char` Java menempati *dua* byte, karena Java menyimpan karakter dalam pengodean UTF-16, yang cukup lapang untuk huruf-huruf dari

kebanyakan aksara dunia. Perhatikan pula apa yang tidak ada: tidak ada alamat yang dicetak, karena Java sengaja menyembunyikannya.

Python melangkah lebih jauh lagi. Modul `struct`-nya dapat memberi tahu kita ukuran tipe-tipe C yang menjadi bahan interpreter Python, yang pada mesin ini sama dengan ukuran di C++, dan `sys.getsizeof` memberi tahu berapa byte yang sungguh-sungguh ditempati sebuah objek Python. **Kode 1.3** melakukan keduanya.

Kode 1.3 `memory_basics.py`: ukuran tipe C dilihat lewat `struct`, byte-byte 2026, dan ukuran sebenarnya objek-objek Python.

```
# memory_basics.py - ukuran, byte, dan tempat nilai berada
import struct
import sys

print("Sizes of the C types Python's struct module uses, in bytes:")
for label, code in [("char", "c"), ("bool", "?"), ("short", "h"), ("int", "i"),
                    ("long long", "q"), ("float", "f"), ("double", "d")]:
    print(f"  {label:<9} {struct.calcsize(code)}")

year = 2026                                # 2026 = 0x000007EA
raw = year.to_bytes(4, sys.byteorder)
print(f"\nThe int {year} as 4 bytes in this machine's order: {raw.hex(' ')}")
print(f"So this machine is {sys.byteorder}-endian.")

on_stack = 7
on_heap = [8]
print(f"\nsys.getsizeof(7)   = {sys.getsizeof(on_stack)} bytes:"
      " even 7 is a whole object")
print(f"sys.getsizeof([8]) = {sys.getsizeof(on_heap)} bytes:"
      " a list object on the heap")
print("Every Python value is an object on the heap; names only refer to it.")
```

```
Sizes of the C types Python's struct module uses, in bytes:
  char      1
  bool      1
  short     2
  int       4
  long long 8
  float     4
  double    8

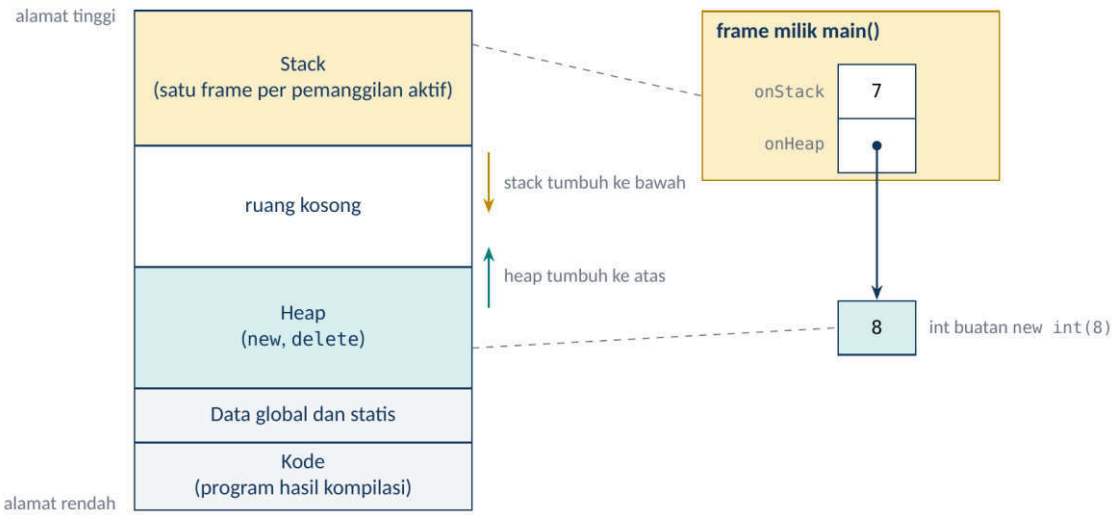
The int 2026 as 4 bytes in this machine's order: ea 07 00 00
So this machine is little-endian.

sys.getsizeof(7)   = 28 bytes: even 7 is a whole object
sys.getsizeof([8]) = 64 bytes: a list object on the heap
Every Python value is an object on the heap; names only refer to it.
```

Bilangan 7 menempati *28 byte* di Python, tujuh kali empat byte `int` C++. Alasannya, bilangan bulat Python adalah objek lengkap di *heap*: selain nilainya, ia mencatat tipenya, hitungan berapa banyak nama yang merujuk kepadanya, dan panjangnya, karena bilangan bulat Python boleh tumbuh sebesar apa pun. Harga itu membeli kenyamanan, dan di Subbab 1.5.10 kita akan melihat berapa biayanya bila ada jutaan bilangan.

1.3.5 Stack dan heap

Ketika sebuah program berjalan, sistem operasi memberinya rentang alamat yang besar dan membaginya menjadi wilayah-wilayah dengan tugas berbeda. **Gambar 1.4** memperlihatkan wilayah-wilayah program C++ yang lazim.



Gambar 1.4 Memori program C++ yang sedang berjalan: kode dan data global di bawah, *heap* tumbuh ke atas, *stack* tumbuh ke bawah, dan, di kanan, *frame* milik `main` yang pointer `onHeap`-nya merujuk ke sebuah `int` di *heap*.

- Wilayah **kode** menyimpan instruksi mesin program itu sendiri.
- Wilayah **data global dan statis** menyimpan variabel yang dideklarasikan di luar semua fungsi. Variabel-variabel ini ada sepanjang program berjalan.
- **Stack** menyimpan variabel-variabel milik fungsi yang sedang berjalan saat itu. Setiap kali sebuah fungsi dipanggil, sebuah blok memori yang disebut **stack frame** disisihkan di puncak *stack* untuk parameter dan variabel lokalnya; ketika fungsi itu kembali, *frame*-nya seketika dilepaskan, cukup dengan menggeser puncak *stack* ke bawah lagi. Karena *frame* ditambahkan dan dibuang hanya di puncak, dengan urutan masuk-terakhir keluar-pertama, wilayah ini berperilaku persis seperti struktur data *stack* di Bab 5, dan dari situlah namanya berasal. *Stack* sangat cepat dan tidak menuntut pembukuan apa pun dari pemrogram, tetapi ukurannya kecil, biasanya satu sampai delapan megabyte, dan variabel di dalamnya hanya hidup selama fungsi yang membuatnya masih berjalan.
- **Heap** menyimpan memori yang diminta program secara eksplisit selagi berjalan, dengan `new` di C++, dan yang harus dikembalikan secara eksplisit dengan `delete`. Variabel di *heap* tidak terikat pada fungsi mana pun: ia hidup sampai program menghapusnya. *Heap* itu besar, sering kali bergigabyte, dan lebih lambat dipakai daripada *stack* karena sistem harus mencari potongan bebas berukuran tepat setiap kali. (Wilayah ini berbagi nama dengan struktur data *heap* di Bab 14, tetapi selain nama keduanya tidak punya kesamaan apa pun.)

Dalam **Kode 1.1**, `onStack` adalah variabel lokal `main`, jadi ia berada di *frame* `main` di *stack*. Ekspresi `new int(8)` meminta ruang untuk satu `int` kepada *heap*, menyimpan 8 di sana, dan memberikan alamatnya, yang disimpan program dalam `onHeap`. Alamat-alamat yang tercetak memperlihatkan kedua wilayah itu dengan jelas: alamat *stack* diawali `0x7ff...`, dekat puncak rentang alamat program, sedangkan alamat *heap* diawali `0x55...`, jauh lebih rendah. Akhirnya `delete onHeap;` mengembalikan memori *heap* itu. Lupa menulis baris itu tidak akan merusak program kecil ini, tetapi dalam program yang berjalan berhari-hari, setiap `delete` yang terlupa adalah kehilangan memori kecil yang permanen, yang disebut **kebocoran memori** (*memory leak*).

Java dan Python menyederhanakan gambaran ini bagi pemrogram. Di Java, variabel lokal bertipe dasar (`int`, `double`, dan sejenisnya) serta *referensi* tinggal di *stack frame*, sedangkan setiap objek dan setiap array tinggal di *heap*. Di Python, setiap nilai adalah objek di *heap*, dan variabel hanyalah nama yang merujuk ke salah satunya. Di kedua bahasa, sebagian dari sistem waktu-

jalannya yang disebut **garbage collector** (pengumpul sampah) menemukan objek-objek *heap* yang tidak lagi dirujuk siapa pun lalu membebaskannya secara otomatis, sehingga tidak ada `delete`. Ini kemudahan besar; harganya, pemrogram tidak lagi menentukan kapan persisnya memori dilepaskan.

Bab ini menaruh hampir segala sesuatu di *stack*, agar dapat memusatkan perhatian pada array. Bab 6 kembali ke *heap*, ke `new` dan `delete`, dan ke pointer, secara tuntas.

1.4 Penyegaran C++ bagi Pemula

Subbab ini mengumpulkan, di satu tempat, seluruh C++ yang diandaikan oleh bagian buku selebihnya. Jika Anda sudah memprogram dalam C++, bacalah sekilas untuk menangkap kosakata dan kebiasaan-kebiasaannya. Jika C++ baru bagi Anda, atau sudah lama tidak Anda pakai, bacalah perlahan, ketikkan program-programnya, dan ubahlah untuk melihat apa yang terjadi: tidak ada yang lebih sulit daripada semestinya dalam struktur data begitu bahasanya sendiri tidak lagi menghalangi. *A Tour of C++* karya Stroustrup [6] adalah pendamping yang baik untuk semua isi subbab ini.

1.4.1 Bentuk sebuah program C++

Setiap program C++ dalam buku ini memiliki kerangka yang sama:

```
#include <iostream>          // masukkan pustaka yang kita perlukan
using namespace std;         // agar kita boleh menulis cout, bukan std::cout

int main() {                 // program mulai berjalan di sini
    cout << "Hello\n";       // pernyataan, masing-masing diakhiri titik koma
    return 0;                // 0 memberi tahu sistem operasi "semua beres"
}
```

- Baris yang diawali `#include` memasukkan bagian-bagian **pustaka standar**: `<iostream>` untuk masukan dan keluaran, `<string>` untuk teks, `<vector>` untuk array yang dapat tumbuh, dan seterusnya.
- Setiap nama dalam pustaka standar tinggal di sebuah *namespace* bernama `std`, sehingga nama lengkapnya adalah `std::cout`, `std::string`, `std::vector`. Baris `using namespace std;` membuat kita boleh menanggalkan awalan itu. Ini praktis untuk program pendek seperti milik kita; dalam program besar, tempat dua pustaka mungkin memakai nama yang sama, kode profesional biasanya menulis `std::` secara lengkap.
- `int main()` adalah fungsi tempat setiap program C++ dimulai. `int` di depan namanya menyatakan bahwa fungsi ini mengembalikan bilangan bulat kepada sistem operasi; menurut kebiasaan, `0` berarti berhasil.
- Kurung kurawal `{ }` mengelompokkan pernyataan menjadi sebuah **blok**. Setiap pernyataan sederhana diakhiri titik koma.
- `//` memulai komentar yang berlanjut sampai akhir baris; `/* ... */` mengapit komentar sepanjang apa pun. Kompilator mengabaikan komentar; manusia tidak.

1.4.2 Tipe dan variabel

Variabel adalah sepotong memori bernama yang menyimpan satu nilai bertipe tertentu. Mendeklarasikannya berarti memesan byte-byte itu dan memberinya nama:

```
int age = 20;                // empat byte, bernama age, berisi 20
double price = 24500.0;      // delapan byte berisi bilangan real
char grade = 'A';            // satu byte berisi kode karakter
bool available = true;       // satu byte berisi true atau false
string title = "Laskar Pelangi";   // objek teks dari <string>
const int LOAN_DAYS = 7;     // konstanta: kompilator menolak mengubahnya
auto count = 8;              // auto: kompilator menyimpulkan tipenya, di sini int
```

Tabel 1.4 mendaftar tipe-tipe dasar yang dipakai buku ini, beserta ukuran yang dicetak **Kode 1.1** dan rentang nilai yang dapat ditampung masing-masing.

Tabel 1.4 Tipe-tipe dasar C++ yang dipakai buku ini, beserta ukurannya pada mesin 64-bit.

Tipe	Byte (di sini)	Menyimpan	Rentang atau ketelitian
`char`	1	satu kode karakter, atau bilangan kecil	-128 sampai 127
`bool`	1	`true` atau `false`	dua nilai
`short`	2	bilangan bulat kecil	$-32{,}768$ sampai $32{,}767$
`int`	4	bilangan bulat	-2^{31} sampai $2^{31}-1$, sekitar ± 2.1 miliar
`long long`	8	bilangan bulat besar	-2^{63} sampai $2^{63}-1$, sekitar $\pm 9.2 \times 10^{18}$
`float`	4	bilangan real	sekitar 7 angka bermakna
`double`	8	bilangan real	sekitar 15 sampai 16 angka bermakna
`size_t`	8	ukuran atau indeks, tidak pernah negatif	0 sampai $2^{64}-1$

Tiga kebiasaan mencegah sebagian besar kesalahan pemula dalam memakai variabel.

Selalu beri nilai pada variabel sebelum membacanya. Variabel lokal yang dideklarasikan sebagai `int x;` berisi byte apa saja yang kebetulan ada di potongan *stack* itu. Membacanya adalah *bug* yang mungkin tampak berfungsi selama berbulan-bulan. Tulislah `int x = 0;`, atau beri nilai pertamanya seketika.

Pilih tipe yang cukup besar. Sebuah `int` tidak dapat menghitung melewati sekitar 2,1 miliar. Total biaya keterlambatan Pustaka Ceria dalam rupiah selama beberapa tahun dapat melampaui angka itu, sehingga uang dalam buku ini disimpan dalam `long` atau `long long`. Ketika sebuah nilai melampaui tipenya, ia diam-diam *meluap* (*overflow*) dan berputar menjadi bilangan yang salah.

Waspadai pembagian bilangan bulat. Bila kedua sisi `/` bilangan bulat, C++ membagi lalu membuang sisanya: `7 / 2` adalah `3`, bukan `3.5`. Untuk mendapatkan hasil bagi yang sebenarnya, ubahlah salah satu sisi menjadi bilangan real lebih dahulu, seperti `static_cast<double>(7) / 2`, yang bernilai `3.5`. Operator `%` memberikan sisa bagi: `7 % 2` adalah `1`.

1.4.3 Masukan dan keluaran

C++ menulis ke layar melalui objek `cout` (*character output*) dengan operator `<<`, dan membaca dari papan ketik melalui `cin` dengan `>>`. Arah panahnya menunjukkan arah aliran data: dari nilai *masuk ke* `cout`, dan dari `cin` *masuk ke* variabel. Beberapa nilai dapat dirangkai: `cout << "Total: " << sum << '\n';`. Karakter `'\n'` memulai baris baru.

Kode 1.4 membaca berapa hari data peminjaman yang menyusul, lalu bilangan bulat sebanyak itu, dan mencetak total, rata-rata, nilai terkecil, dan nilai terbesarnya.

Kode 1.4 `io_sum.cpp`: membaca sebuah cacah dan bilangan sebanyak itu dengan `cin`, lalu meringkasnya.

```cpp
// io_sum.cpp - baca sebuah cacah dan bilangan sebanyak itu, lalu ringkas
#include <iomanip>
#include <iostream>
using namespace std;

int main() {
    int n;
    if (!(cin >> n) || n <= 0) {
        cout << "Expected a positive count first.\n";
        return 1;
    }
    long long sum = 0;
    int smallest = 0, largest = 0;
    for (int i = 0; i < n; ++i) {
        int x;
        cin >> x;
        sum += x;
        if (i == 0 || x < smallest) smallest = x;
        if (i == 0 || x > largest) largest = x;
    }
    double average = static_cast<double>(sum) / n;
    cout << "Days read : " << n << '\n';
    cout << "Total     : " << sum << '\n';
    cout << "Average   : " << fixed << setprecision(2) << average << '\n';
    cout << "Smallest  : " << smallest << '\n';
    cout << "Largest   : " << largest << '\n';
    return 0;
}
```

Berkas `week_loans.txt` di folder bab ini menyimpan jumlah peminjaman Pustaka Ceria selama satu minggu:

```
7
12 9 15 4 20 11 6
```

Alih-alih mengetik bilangan-bilangan itu, kita dapat membiarkan program membaca berkas seolah-olah berkas itu papan ketik, dengan **mengalihkan** (*redirect*) masukan standar memakai `<` ketika menjalankannya: `./io_sum < week_loans.txt` di macOS dan Linux, `io_sum.exe < week_loans.txt` di Command Prompt Windows, atau `Get-Content week_loans.txt | .\io_sum.exe` di PowerShell. Program itu mencetak:

```
Days read : 7
Total     : 77
Average   : 11.00
Smallest  : 4
Largest   : 20
```

Beberapa rincian layak diberi sepatah kata. Uji `if (!(cin >> n) || n <= 0)` membaca cacahnya *sekaligus* memeriksa bahwa pembacaan itu berhasil: `cin >> n` gagal, dan ujinya menjadi benar, bila masukan tidak diawali sebuah bilangan. Variabel `smallest` dan `largest` diisi dari bilangan pertama, `i == 0`, karena sebelum itu belum ada pembanding. Rata-rata dihitung sebagai `static_cast<double>(sum) / n` untuk menghindari pembagian bilangan bulat, dan `fixed << setprecision(2)` mencetaknya dengan tepat dua angka desimal.

Java membaca masukan melalui `Scanner` yang membungkus `System.in`, dan Python melalui `input()` atau, untuk satu berkas penuh token, `sys.stdin.read().split()`. **Kode 1.5** dan **Kode 1.6** adalah kedua versinya; keduanya membaca berkas yang sama dan mencetak kelima baris yang persis sama.

Kode 1.5 `IoSum.java`: ringkasan yang sama dengan `Scanner`.

```java
// IoSum.java - baca sebuah cacah dan bilangan sebanyak itu, lalu ringkas
import java.util.Scanner;

public class IoSum {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        if (!in.hasNextInt()) {
            System.out.println("Expected a positive count first.");
            return;
        }
        int n = in.nextInt();
        if (n <= 0) {
            System.out.println("Expected a positive count first.");
            return;
        }
        long sum = 0;
        int smallest = 0, largest = 0;
        for (int i = 0; i < n; i++) {
            int x = in.nextInt();
            sum += x;
            if (i == 0 || x < smallest) smallest = x;
            if (i == 0 || x > largest) largest = x;
        }
        double average = (double) sum / n;
        System.out.println("Days read : " + n);
        System.out.println("Total     : " + sum);
        System.out.printf("Average   : %.2f%n", average);
        System.out.println("Smallest  : " + smallest);
        System.out.println("Largest   : " + largest);
    }
}
```

Kode 1.6 `io_sum.py`: ringkasan yang sama, dengan membaca semua token dari masukan standar sekaligus.

```python
# io_sum.py - baca sebuah cacah dan bilangan sebanyak itu, lalu ringkas
import sys

tokens = sys.stdin.read().split()
if not tokens or not tokens[0].lstrip("-").isdigit() or int(tokens[0]) <= 0:
    print("Expected a positive count first.")
    sys.exit(1)
n = int(tokens[0])
values = [int(t) for t in tokens[1:1 + n]]

total = sum(values)
average = total / n
print(f"Days read : {n}")
print(f"Total     : {total}")
print(f"Average   : {average:.2f}")
print(f"Smallest  : {min(values)}")
print(f"Largest   : {max(values)}")
```

Versi Python memperlihatkan gaya lain yang dianjurkan Python: alih-alih memperbarui total berjalan di dalam perulangan, ia membangun list `values` dalam satu baris lalu menyerahkannya kepada fungsi bawaan `sum`, `min`, dan `max`. Pekerjaan yang dilakukan sama saja.

1.4.4 Mengambil keputusan dan mengulang

Program memilih dan mengulang. C++ memiliki tiga pernyataan untuk memilih dan tiga untuk mengulang, dan Java memiliki pernyataan yang persis sama dengan ejaan yang sama.

`if` dan `else` menjalankan satu blok atau blok lain bergantung pada sebuah kondisi:

```
if (daysLate <= 0) {
    fee = 0;
} else if (daysLate <= 7) {
    fee = daysLate * 1000;
} else {
    fee = 7000 + (daysLate - 7) * 2000;
}
```

Kondisi dibangun dari perbandingan, `==` (sama dengan; perhatikan *dua* tanda sama dengan), `!=`, `<`, `<=`, `>`, dan `>=`, yang digabungkan dengan `&&` (dan), `||` (atau), dan `!` (bukan).

`switch` memilih di antara banyak kasus menurut nilai sebuah bilangan bulat atau karakter. Setiap `case` sebaiknya diakhiri `break` atau `return`; kalau tidak, eksekusi "jatuh" (*fall through*) ke kasus berikutnya, yang sesekali berguna dan biasanya merupakan *bug*. Kasus `default` menangkap setiap nilai yang tidak terdaftar.

`while` mengulang sebuah blok selama sebuah kondisi terpenuhi, dengan menguji sebelum setiap putaran; `do ... while` menguji sesudah setiap putaran, sehingga bloknya berjalan paling sedikit sekali; dan `for` mengemas ketiga bagian perulangan berhitung dalam satu baris:

```
for (int i = 0; i < n; i++) {    // mulai; teruskan selama; langkah
    // berjalan untuk i = 0, 1, ..., n-1
}
```

Bacalah `for (int i = 0; i < n; i++)` sebagai "mulailah `i` dari nol; selama `i` kurang dari `n`, jalankan bloknya lalu tambahkan satu pada `i`". Perulangan ini berjalan tepat n kali, dengan `i` mengambil nilai 0 sampai $n - 1$: persis posisi-posisi sah sebuah array berisi n elemen, dan itulah sebabnya baris ini muncul di hampir setiap program dalam buku ini. **For berbasis rentang** (*range-based for*), `for (const Loan& loan : loans)`, mengunjungi setiap elemen sebuah array atau wadah tanpa indeks sama sekali.

Di dalam perulangan apa pun, `break` meninggalkan perulangan seketika, dan `continue` melewatkan sisa putaran saat itu lalu melanjutkan ke putaran berikutnya.

1.4.5 Fungsi

Fungsi adalah potongan program bernama yang menerima beberapa nilai, yaitu *parameter*-nya, melakukan suatu pekerjaan, dan boleh mengembalikan satu nilai, yaitu *nilai kembalian*-nya. Fungsi memungkinkan kita menulis sebuah gagasan sekali, memberinya nama, dan memakainya berkali-kali; fungsi juga memungkinkan kita membaca program panjang sebagai beberapa langkah bermakna, bukan seribu baris.

```
long lateFee(int daysKept, char memberType) {   // tipe kembalian, nama, parameter
    ...
    return base / 2;                             // kembalikan sebuah nilai
}
```

Baris pertama adalah **signature** (tanda tangan) fungsi: ia mengembalikan `long`, bernama `lateFee`, dan menerima sebuah `int` dan sebuah `char`. Fungsi yang tidak mengembalikan apa-apa bertipe kembalian `void`. Fungsi harus *dideklarasikan* sebelum dipanggil; dalam program-program pendek kita, setiap fungsi cukup didefinisikan di atas `main`.

Secara bawaan C++ mengoper parameter **menurut nilai** (*by value*): fungsi menerima *salinan* setiap argumen, dan mengubah salinan itu tidak mengubah variabel milik pemanggil. Subbab 1.4.7 memperlihatkan cara membiarkan fungsi mengubah variabel pemanggil, dan Bab 6 membahas persoalan ini secara tuntas.

1.4.6 Struct: banyak nilai di bawah satu nama

Sebuah buku di Pustaka Ceria mempunyai kode, judul, pengarang, dan kategori; sebuah peminjaman mempunyai judul, banyaknya hari, dan jenis anggota. Menyimpan masing-masing dalam variabel terpisah segera menjadi mustahil dikelola. **Struct** menghimpun nilai-nilai yang berkaitan menjadi satu tipe baru:

```
struct Loan {
    string title;
    int daysKept;
    char memberType;
};

Loan first = {"Bumi Manusia", 10, 'S'};   // buat satu, beri nilai setiap field
cout << first.title;                       // baca sebuah field dengan titik
first.daysKept = 12;                       // ubah sebuah field
```

Nilai-nilai di dalam struct adalah **field**-nya atau *anggota*-nya, dan tanda titik memilih salah satunya. Struct adalah cara paling sederhana untuk menguraikan sebuah rekaman, dan array berisi struct adalah tabel rekaman yang paling sederhana, salah satu struktur data yang paling umum dari semuanya.

Kode 1.7 menyatukan semua gagasan subbab ini. Pustaka Ceria meminjamkan buku selama tujuh hari. Buku yang terlambat dikembalikan dikenai Rp1.000 per hari untuk minggu pertama keterlambatan dan Rp2.000 per hari sesudahnya; mahasiswa membayar separuh, dosen membayar penuh, dan tamu membayar dua kali lipat. Program ini menyimpan enam peminjaman dalam array berisi struct `Loan` dan menghitung setiap biaya keterlambatan dengan satu fungsi.

Kode 1.7 `late_fee.cpp`: konstanta, sebuah struct, fungsi dengan `if` dan `switch`, array berisi struct, dan perulangan `for` berbasis rentang dengan `continue`.

```cpp
// late_fee.cpp - alur kendali, fungsi, struct: biaya terlambat Pustaka Ceria
#include <iostream>
#include <string>
using namespace std;

const int LOAN_DAYS = 7;                  // buku boleh dipinjam selama 7 hari

struct Loan {
    string title;
    int daysKept;
    char memberType;                      // 'S' mahasiswa, 'L' dosen, 'G' tamu
};

// Biaya dalam rupiah, atau -1 bila jenis anggota tak dikenal.
long lateFee(int daysKept, char memberType) {
    int daysLate = daysKept - LOAN_DAYS;
    if (daysLate <= 0) {
        return 0;
    }
    long base;
    if (daysLate <= 7) {
        base = daysLate * 1000L;          // Rp1.000/hari pada minggu pertama
    } else {
        base = 7 * 1000L + (daysLate - 7) * 2000L;   // lalu Rp2.000/hari
    }
    switch (memberType) {
        case 'S': return base / 2;        // mahasiswa membayar separuh
        case 'L': return base;
        case 'G': return base * 2;        // tamu membayar dua kali lipat
        default:  return -1;
    }
}

int main() {
    Loan loans[] = {
        {"Laskar Pelangi", 5, 'S'},
        {"Bumi Manusia", 10, 'S'},
        {"Filosofi Kopi", 10, 'L'},
        {"Fisika Dasar", 16, 'G'},
        {"Negeri 5 Menara", 20, 'S'},
        {"Sejarah Nusantara", 9, 'X'},
    };
    long total = 0;
    for (const Loan& loan : loans) {
        long fee = lateFee(loan.daysKept, loan.memberType);
        if (fee < 0) {
```

Kode 1.7 (Lanjutan) *late_fee.cpp: konstanta, sebuah struct, fungsi dengan if dan switch, array berisi struct, dan perulangan for berbasis rentang dengan continue.*

```cpp
            cout << loan.title << ": unknown member type '" << loan.memberType
                 << "'\n";
            continue;                     // lewati, lanjut ke berikutnya
        }
        cout << loan.title << " (" << loan.memberType << ", " << loan.daysKept
             << " days): Rp" << fee << '\n';
        total += fee;
    }
    cout << "Total fees: Rp" << total << '\n';
    return 0;
}
```

```
Laskar Pelangi (S, 5 days): Rp0
Bumi Manusia (S, 10 days): Rp1500
Filosofi Kopi (L, 10 days): Rp3000
Fisika Dasar (G, 16 days): Rp22000
Negeri 5 Menara (S, 20 days): Rp9500
Sejarah Nusantara: unknown member type 'X'
Total fees: Rp36000
```

Periksalah dua baris dengan tangan, karena mengerjakan program dengan tangan adalah kebiasaan paling berguna yang dapat diajarkan buku ini. *Fisika Dasar* dipinjam 16 hari oleh seorang tamu: terlambat $16 - 7 = 9$ hari, sehingga biaya dasarnya $7 \times 1000 + (9 - 7) \times 2000 = 7000 + 4000 = 11{,}000$, dan tamu membayar dua kali lipat, 22,000. *Negeri 5 Menara* dipinjam 20 hari oleh seorang mahasiswa: terlambat 13 hari, dasar $7000 + 6 \times 2000 = 19{,}000$, separuhnya 9,500. Jenis anggota tak dikenal `'X'` membuat `lateFee` mengembalikan -1; perulangan melaporkannya lalu `continue` ke peminjaman berikutnya, sehingga satu rekaman yang buruk tidak merusak totalnya.

Di Java, rumah yang wajar untuk sekelompok kecil nilai adalah **record**, sebuah kelas yang field-field-nya ditetapkan saat ia dibuat dan yang secara otomatis dilengkapi Java dengan konstruktor dan metode pengakses seperti `loan.title()`. Di Python rumahnya adalah **dataclass**, dan Python 3.10 ke atas dapat mengganti rangkaian pernyataan `if` pada satu nilai dengan pernyataan `match`. **Kode 1.8** dan **Kode 1.9** mencetak ketujuh baris yang persis sama dengan versi C++.

Kode 1.8 `LateFee.java`: biaya keterlambatan dengan `record` Java dan `switch`.

```java
// LateFee.java - alur kendali, metode, record: biaya terlambat Pustaka Ceria
public class LateFee {
    static final int LOAN_DAYS = 7;           // buku boleh dipinjam selama 7 hari

    record Loan(String title, int daysKept, char memberType) { }

    // Biaya dalam rupiah, atau -1 bila jenis anggota tak dikenal.
    static long lateFee(int daysKept, char memberType) {
        int daysLate = daysKept - LOAN_DAYS;
        if (daysLate <= 0) {
            return 0;
        }
        long base;
        if (daysLate <= 7) {
            base = daysLate * 1000L;          // Rp1.000/hari pada minggu pertama
        } else {
            base = 7 * 1000L + (daysLate - 7) * 2000L;   // lalu Rp2.000/hari
        }
        switch (memberType) {
            case 'S': return base / 2;        // mahasiswa membayar separuh
            case 'L': return base;
            case 'G': return base * 2;        // tamu membayar dua kali lipat
            default:  return -1;
        }
    }

    public static void main(String[] args) {
        Loan[] loans = {
            new Loan("Laskar Pelangi", 5, 'S'),
            new Loan("Bumi Manusia", 10, 'S'),
            new Loan("Filosofi Kopi", 10, 'L'),
            new Loan("Fisika Dasar", 16, 'G'),
            new Loan("Negeri 5 Menara", 20, 'S'),
            new Loan("Sejarah Nusantara", 9, 'X'),
        };
        long total = 0;
        for (Loan loan : loans) {
            long fee = lateFee(loan.daysKept(), loan.memberType());
            if (fee < 0) {
                System.out.println(loan.title() + ": unknown member type '"
                        + loan.memberType() + "'");
                continue;                     // lewati, lanjut ke berikutnya
            }
            System.out.println(loan.title() + " (" + loan.memberType() + ", "
                    + loan.daysKept() + " days): Rp" + fee);
            total += fee;
```

Kode 1.8 (Lanjutan) *LateFee.java: biaya keterlambatan dengan record Java dan switch.*

```java
        }
        System.out.println("Total fees: Rp" + total);
    }
}
```

Kode 1.9 `late_fee.py`: biaya keterlambatan dengan `dataclass` dan pernyataan `match`.

```python
# late_fee.py - alur kendali, fungsi, dataclass: biaya terlambat Pustaka Ceria
from dataclasses import dataclass

LOAN_DAYS = 7                               # buku boleh dipinjam selama 7 hari


@dataclass
class Loan:
    title: str
    days_kept: int
    member_type: str                        # 'S' mahasiswa, 'L' dosen, 'G' tamu


def late_fee(days_kept: int, member_type: str) -> int:
    """Biaya dalam rupiah, atau -1 bila jenis anggota tak dikenal."""
    days_late = days_kept - LOAN_DAYS
    if days_late <= 0:
        return 0
    if days_late <= 7:
        base = days_late * 1000             # Rp1.000/hari pada minggu pertama
    else:
        base = 7 * 1000 + (days_late - 7) * 2000   # lalu Rp2.000/hari
    match member_type:
        case "S":
            return base // 2                # mahasiswa membayar separuh
        case "L":
            return base
        case "G":
            return base * 2                 # tamu membayar dua kali lipat
        case _:
            return -1


loans = [
    Loan("Laskar Pelangi", 5, "S"),
    Loan("Bumi Manusia", 10, "S"),
    Loan("Filosofi Kopi", 10, "L"),
    Loan("Fisika Dasar", 16, "G"),
    Loan("Negeri 5 Menara", 20, "S"),
    Loan("Sejarah Nusantara", 9, "X"),
]
total = 0
for loan in loans:
    fee = late_fee(loan.days_kept, loan.member_type)
    if fee < 0:
        print(f"{loan.title}: unknown member type '{loan.member_type}'")
```

Kode 1.9 (Lanjutan) *late_fee.py: biaya keterlambatan dengan dataclass dan pernyataan match.*

```python
        continue                            # lewati, lanjut ke berikutnya
    print(f"{loan.title} ({loan.member_type}, {loan.days_kept} days): Rp{fee}")
    total += fee
print(f"Total fees: Rp{total}")
```

Satu perbedaan kecil mudah terlewat: C++ dan Java menghitung separuh biaya dengan `/`, yang untuk bilangan bulat sudah membuang sisanya, tetapi `/` di Python selalu menghasilkan bilangan real, sehingga versi Python memakai `//`, pembagian bilangan bulat milik Python.

1.4.7 Operator alamat, dan pandangan pertama pada pointer

Kita sudah sekali memakai operator `&`, dalam **Kode 1.1**. Kini saatnya memandangnya dengan saksama, karena operator inilah pintu menuju separuh isi buku ini.

Setiap variabel mempunyai dua hal yang dapat kita tanyakan: **nilai**-nya, yaitu data yang tersimpan di dalamnya, dan **alamat**-nya, yaitu di mana di memori ia disimpan. Menulis nama variabel memberikan nilainya. Menulis `&` di depan nama itu memberikan alamatnya. Variabel yang nilainya berupa alamat disebut **pointer** (penunjuk). Pointer dideklarasikan dengan `*` sesudah tipe benda yang ditunjuknya:

```
int age = 20;
int* p = &age;      // p adalah "pointer ke int", berisi alamat age
cout << *p;         // * di depan pointer "mengikutinya": mencetak 20
*p = 21;            // menulis melalui pointer: age kini 21
```

Simbol `*` yang sama memainkan dua peran, dan itu membingungkan semua orang pada awalnya. Dalam sebuah *deklarasi*, `int* p`, ia berarti "p adalah pointer". Dalam sebuah *ekspresi*, `*p`, ia berarti "benda yang ditunjuk p", dan disebut **dereferensi**. **Kode 1.10** memperlihatkan semua ini, sekaligus alasan praktis mengapa kita peduli: fungsi yang menerima sebuah *alamat* dapat mengubah variabel pemanggil, sedangkan fungsi yang menerima sebuah *salinan* tidak dapat.

Kode 1.10 `address_of.cpp`: `&` memberikan alamat, `*` mengikutinya, dan fungsi yang menerima alamat dapat mengubah variabel pemanggil.

```
// address_of.cpp - operator alamat, pointer pertama, dan dua cara mengoper
#include <iostream>
using namespace std;

void birthdayByValue(int a) {             // a adalah salinan
    a = a + 1;
}

void birthdayByAddress(int* p) {          // p menyimpan alamat milik pemanggil
    *p = *p + 1;
}

int main() {
    int age = 20;
    int* p = &age;                        // p kini menyimpan alamat age

    cout << "age          = " << age << '\n';
    cout << "&age         = " << &age << "  (an address)\n";
    cout << "p            = " << p << "  (the same address)\n";
    cout << "*p           = " << *p << "  (the value found there)\n";
    cout << "sizeof(age)  = " << sizeof(age) << " bytes\n";
    cout << "sizeof(p)    = " << sizeof(p) << " bytes\n";

    *p = 21;                              // menulis melalui pointer
    cout << "after *p = 21, age = " << age << '\n';

    birthdayByValue(age);
    cout << "after birthdayByValue(age), age = " << age << '\n';
    birthdayByAddress(&age);
    cout << "after birthdayByAddress(&age), age = " << age << '\n';
    return 0;
}
```

```
age          = 20
&age         = 0x7ffd1c876cd4  (an address)
p            = 0x7ffd1c876cd4  (the same address)
*p           = 20  (the value found there)
sizeof(age)  = 4 bytes
sizeof(p)    = 8 bytes
after *p = 21, age = 21
after birthdayByValue(age), age = 21
after birthdayByAddress(&age), age = 22
```

`birthdayByValue(age)` menerima salinan 21 dalam parameternya `a`, menambahkan satu pada salinan itu, dan salinan itu lenyap ketika fungsi kembali; `age` tetap 21. `birthdayByAddress(&age)` menerima *alamat* `age`, dan `*p = *p + 1` mengubah `int` yang tersimpan di alamat itu, yaitu `age` sendiri; sesudahnya `age` bernilai 22. Perhatikan pula `sizeof(p) = 8`: pada mesin 64-bit setiap alamat, dan karena itu setiap pointer, menempati delapan byte, apa pun yang ditunjuknya.

Java dan Python tidak memiliki operator `&` dan tidak pernah memperlihatkan alamat. Meski begitu, gagasan dasarnya tetap ada: di kedua bahasa, variabel bertipe objek menyimpan sebuah **referensi**, yang berperilaku seperti pointer yang dikelola bahasa itu untuk kita. Menugaskan satu referensi ke referensi lain menghasilkan dua nama untuk objek yang *sama*, dan metode yang menerima referensi ke sebuah array atau list dapat mengubah isinya. **Kode 1.11** dan **Kode 1.12** memperlihatkan percobaan yang sama, dengan array atau list satu slot yang memerankan alamat.

Kode 1.11 `AddressOf.java`: Java menyalin nilai dasar ke dalam metode, tetapi berbagi objek lewat referensi.

```java
// AddressOf.java - Java tanpa &: nilai disalin, objek dibagi bersama
public class AddressOf {
    static void birthdayByValue(int a) {      // a adalah salinan
        a = a + 1;
    }

    static void birthdayInBox(int[] box) {    // box merujuk ke array pemanggil
        box[0] = box[0] + 1;
    }

    public static void main(String[] args) {
        int age = 20;
        int[] box = {20};                     // array satu slot di heap
        int[] sameBox = box;                  // referensi kedua, array yang sama

        System.out.println("age          = " + age);
        System.out.println("box[0]       = " + box[0]);
        sameBox[0] = 21;                       // menulis melalui referensi kedua
        System.out.println("after sameBox[0] = 21, box[0] = " + box[0]);

        birthdayByValue(age);
        System.out.println("after birthdayByValue(age), age = " + age);
        birthdayInBox(box);
        System.out.println("after birthdayInBox(box), box[0] = " + box[0]);
    }
}
```

```
age          = 20
box[0]       = 20
after sameBox[0] = 21, box[0] = 21
after birthdayByValue(age), age = 20
after birthdayInBox(box), box[0] = 22
```

Kode 1.12 `address_of.py`: nama-nama Python merujuk ke objek; `id()` memperlihatkan bahwa dua nama merujuk ke list yang sama.

```python
# address_of.py - Python tidak punya &: nama merujuk ke objek, dan id() mengenalinya
def birthday_by_value(a):
    a = a + 1                               # hanya mengikat ulang nama lokal


def birthday_in_box(box):
    box[0] = box[0] + 1                     # mengubah list milik pemanggil


age = 20
box = [20]
same_box = box                              # nama kedua untuk list yang sama

print(f"age          = {age}")
print(f"box[0]       = {box[0]}")
same_box[0] = 21
print(f"after same_box[0] = 21, box[0] = {box[0]}")
print(f"id(box) == id(same_box) ? {id(box) == id(same_box)}")

birthday_by_value(age)
print(f"after birthday_by_value(age), age = {age}")
birthday_in_box(box)
print(f"after birthday_in_box(box), box[0] = {box[0]}")
```

```
age          = 20
box[0]       = 20
after same_box[0] = 21, box[0] = 21
id(box) == id(same_box) ? True
after birthday_by_value(age), age = 20
after birthday_in_box(box), box[0] = 22
```

Sejauh inilah pointer dibahas dalam bab ini: cukup jauh untuk membaca sebuah alamat dan untuk memahami mengapa `catalog[i]` di subbab berikutnya sama dengan `*(catalog + i)`. Bab 6 menuntaskan kisahnya, dan sejak Bab 7 pointer memikul beban setiap struktur berantai dalam buku ini.

1.5 Array Secara Mendalam

1.5.1 Satu deret slot bernomor

Rak pertama Pustaka Ceria adalah sebilah papan dengan tempat-tempat bernomor, dan struktur data pertamanya persis seperti itu.

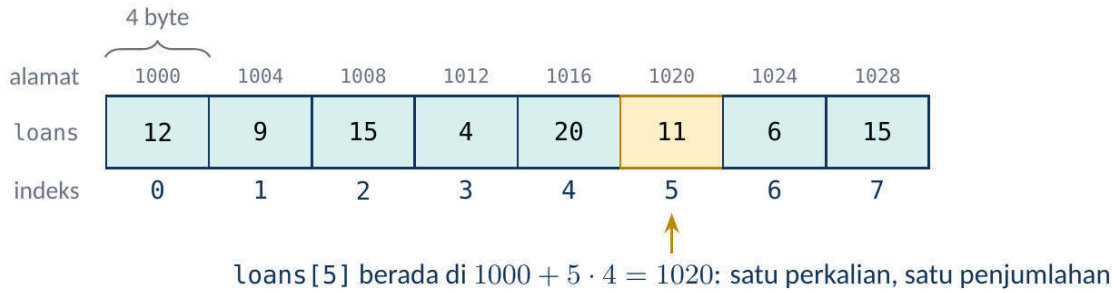
> **DEFINISI.** *Array* (larik) adalah sejumlah tetap elemen bertipe sama yang disimpan berurutan dalam satu blok memori yang bersinambung, masing-masing dikenali dengan sebuah *indeks* bilangan bulat. Di C++, Java, dan Python indeks pertama adalah 0, sehingga array berisi n elemen mempunyai indeks $0, 1, \ldots, n-1$ [1], [7].

Di C++ sebuah array dideklarasikan dengan menyebut tipe elemen, nama, dan, di dalam kurung siku, banyaknya elemen:

```
int loans[8];                                   // 8 int, nilainya belum diisi
int loans[8] = {12, 9, 15, 4, 20, 11, 6, 15};   // 8 int, semuanya diisi
int loans[] = {12, 9, 15, 4, 20, 11, 6, 15};    // kompilator yang menghitung: 8
int zeros[5] = {0};                             // yang pertama 0, sisanya juga
```

Banyaknya elemen harus sudah diketahui ketika program dikompilasi, dan tidak pernah berubah. Elemen berindeks `i` ditulis `loans[i]`, dan dapat dibaca atau diubah seperti variabel mana pun: `loans[3] = 5;` menaruh 5 di tempat keempat.

Gambar 1.5 memperlihatkan array `loans` di memori. Kedelapan `int` itu menempati $8 \times 4 = 32$ byte, dari alamat 1000 sampai 1031, tanpa apa pun di antaranya.



Gambar 1.5 Array `loans` berisi delapan `int` di memori: setiap elemen menempati 4 byte, sehingga elemen i bermula di $1000 + 4i$.

Karena elemen-elemennya berukuran sama dan duduk berdampingan, alamat elemen mana pun dapat *dihitung* alih-alih dicari. Jika array bermula di **alamat dasar** b dan setiap elemen menempati s byte, elemen berindeks i bermula di

$$\text{alamat}(A[i]) = b + i \cdot s \tag{1.2}$$

Dalam **Gambar 1.5**, $b = 1000$ dan $s = 4$, sehingga `loans[5]` bermula di $1000 + 5 \cdot 4 = 1020$, dan `loans[0]` di 1000 itu sendiri. Kompilator C++ menyisipkan tepat perhitungan ini setiap kali ia bertemu `loans[i]` dalam program: satu perkalian, satu penjumlahan, dan lompatan ke alamat itu. Tidak jadi soal apakah array itu berisi delapan elemen atau delapan juta; mencapai elemen i berbiaya sama. Persamaan (1.2) adalah seluruh rahasia array, dan Subbab 1.6 mengubahnya menjadi pernyataan pertama kita tentang kompleksitas.

1.5.2 Mengapa indeks dimulai dari 0

Pemula sering merasa janggal bahwa elemen pertama bernomor 0 dan yang terakhir bernomor $n - 1$. Persamaan (1.2) menjelaskannya: indeks bukanlah "elemen ke berapa dalam hitungan", melainkan "berapa elemen melewati titik awal". Elemen pertama berjarak nol elemen dari titik awal, sehingga alamatnya $b + 0 \cdot s = b$. Seandainya indeks dimulai dari 1, setiap akses harus menghitung $b + (i - 1) \cdot s$, satu pengurangan lebih banyak, selamanya.

C++ membuat hubungan antara array dan alamat menjadi gamblang. Dalam kebanyakan ekspresi, nama sebuah array mewakili alamat elemen pertamanya, dan menambahkan bilangan bulat i pada alamat bertipe "pointer ke `T`" menggesernya maju sejauh i *elemen*, yaitu $i \cdot$ sizeof(T) byte. Jadi

```
loans[i]    sama persis dengan    *(loans + i)
&loans[i]   sama persis dengan    loans + i
```

Kurung siku hanyalah ejaan yang lebih ramah untuk "mulai dari dasar, geser i elemen, ikuti alamatnya". Program-program di subbab berikutnya memeriksa kesamaan ini, elemen demi elemen.

Ingatlah satu kalimat dari subbab ini: **array berisi n elemen tidak mempunyai elemen berindeks n.** Yang terakhir adalah $n - 1$.

1.5.3 Array berisi rekaman: katalog Pustaka Ceria

Array boleh berisi elemen bertipe apa pun, termasuk struct. Pustaka Ceria menyimpan empat fakta tentang setiap buku: kode lima karakter seperti `B1001`, judul, nama pengarangnya, dan

kategori. Berkas header `book.h` dalam **Kode 1.13** menguraikan rekaman itu beserta tiga fungsi pembantu kecil; kedua program C++ berikutnya menyertakannya.

Kode 1.13 `book.h`: rekaman `Book` milik Pustaka Ceria, dengan array karakter berukuran tetap, serta fungsi pembantu untuk mengisi dan mencetaknya.

```
// book.h - rekaman Pustaka Ceria, dipakai catalog_1d.cpp dan shelves_2d.cpp
#ifndef BOOK_H
#define BOOK_H

#include <cstdio>
#include <cstring>

struct Book {
    char id[6];                          // "B1001" ditambah penutup '\0'
    char title[100];
    char author[60];
    char category[30];
};

// Menyalin src ke dst, memotongnya bila perlu; dst selalu diakhiri '\0'.
inline void safeCopy(char* dst, std::size_t dstSize, const char* src) {
    std::strncpy(dst, src, dstSize - 1);
    dst[dstSize - 1] = '\0';
}

inline void populateBook(Book& b, const char* id, const char* title,
                         const char* author, const char* category) {
    safeCopy(b.id, sizeof(b.id), id);
    safeCopy(b.title, sizeof(b.title), title);
    safeCopy(b.author, sizeof(b.author), author);
    safeCopy(b.category, sizeof(b.category), category);
}

inline void printCatalogHeader() {
    std::printf("%-6s %-26s %-20s %s\n", "ID", "TITLE", "AUTHOR", "CATEGORY");
}

inline void printBook(const Book& b) {
    std::printf("%-6s %-26s %-20s %s\n", b.id, b.title, b.author, b.category);
}

#endif
```

Setiap field berupa array `char` berukuran tetap, bukan `std::string`, dan itu disengaja. `std::string` menyimpan karakter-karakternya di tempat lain, di *heap*, dan hanya memegang catatan kecil tentang letaknya; struct berisi array `char` menyimpan semua karakternya *di dalam* dirinya sendiri, sehingga seluruh `Book` menjadi satu blok padat berukuran $6 + 100 + 60 + 30 = 196$ byte. Itu menjadikannya contoh paling bersih tentang tata letak array berisi rekaman. Harganya adalah sebuah batas: judul paling panjang 99 karakter, karena satu `char` diperlukan untuk `'\0'` yang menandai akhir string gaya C. Fungsi pembantu `safeCopy` menghormati batas itu, dengan memotong teks yang lebih panjang alih-alih menulis melewati ujung field.

Kode 1.14 mendeklarasikan array berisi delapan `Book`, mengisi dan mencetaknya, lalu mengukur di mana setiap elemen bermula, relatif terhadap elemen pertama.

Kode 1.14 `catalog_1d.cpp`: katalog Pustaka Ceria sebagai array berisi delapan `Book`, beserta offset byte setiap elemen.

```
// catalog_1d.cpp - array satu dimensi berisi Book, dan letak setiap elemennya
#include <cstddef>
#include <cstdio>
#include "book.h"

const int N = 8;

int main() {
    Book catalog[N];
    populateBook(catalog[0], "B1001", "Laskar Pelangi",
                 "Andrea Hirata", "Fiksi");
    populateBook(catalog[1], "B1002", "Bumi Manusia",
                 "Pramoedya A. Toer", "Fiksi");
    populateBook(catalog[2], "B1003", "Filosofi Kopi",
                 "Dee Lestari", "Fiksi");
    populateBook(catalog[3], "B1004", "Sejarah Nusantara",
                 "Slamet Muljana", "Sejarah");
    populateBook(catalog[4], "B1005", "Algoritma & Struktur Data",
                 "Rinaldi Munir", "Teknologi");
    populateBook(catalog[5], "B1006", "Cerita Rakyat Nusantara",
                 "Tim Pustaka Ceria", "Anak-anak");
    populateBook(catalog[6], "B1007", "Fisika Dasar",
                 "Halliday & Resnick", "Sains");
    populateBook(catalog[7], "B1008", "Negeri 5 Menara",
                 "Ahmad Fuadi", "Fiksi");

    std::printf("Pustaka Ceria catalog, %d books\n", N);
    printCatalogHeader();
    for (int i = 0; i < N; i++) {
        printBook(catalog[i]);
    }

    std::printf("\nsizeof(Book) = %zu bytes\n", sizeof(Book));
    std::printf("field offsets: id %zu, title %zu, author %zu, category %zu\n",
                offsetof(Book, id), offsetof(Book, title),
                offsetof(Book, author), offsetof(Book, category));
```

Kode 1.14 (Lanjutan) *catalog_1d.cpp: katalog Pustaka Ceria sebagai array berisi delapan Book, beserta offset byte setiap elemen.*

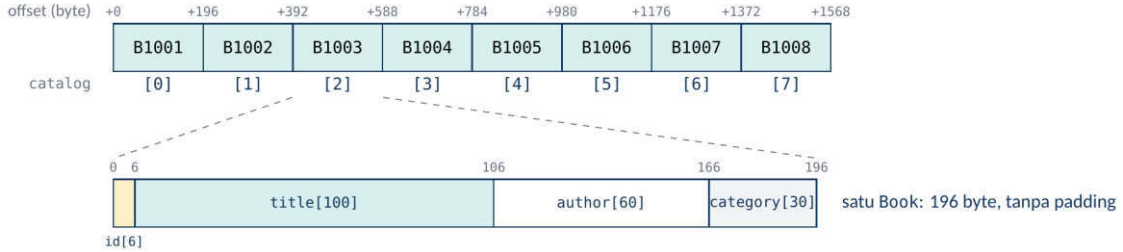
```
    const char* base = reinterpret_cast<const char*>(&catalog[0]);
    std::printf("\n%-3s %-26s %s\n",
                "i", "&catalog[i] - base (bytes)", "== catalog + i ?");
    for (int i = 0; i < N; i++) {
        const char* here = reinterpret_cast<const char*>(&catalog[i]);
        long offset = static_cast<long>(here - base);
        bool same = (&catalog[i] == catalog + i);
        std::printf("%-3d %-26ld %s\n", i, offset, same ? "yes" : "NO");
    }
    return 0;
}
```

```
Pustaka Ceria catalog, 8 books
ID     TITLE                      AUTHOR               CATEGORY
B1001  Laskar Pelangi             Andrea Hirata        Fiksi
B1002  Bumi Manusia               Pramoedya A. Toer    Fiksi
B1003  Filosofi Kopi              Dee Lestari          Fiksi
B1004  Sejarah Nusantara          Slamet Muljana       Sejarah
B1005  Algoritma & Struktur Data  Rinaldi Munir        Teknologi
B1006  Cerita Rakyat Nusantara    Tim Pustaka Ceria    Anak-anak
B1007  Fisika Dasar               Halliday & Resnick   Sains
B1008  Negeri 5 Menara            Ahmad Fuadi          Fiksi

sizeof(Book) = 196 bytes
field offsets: id 0, title 6, author 106, category 166

i   &catalog[i] - base (bytes) == catalog + i ?
0   0                          yes
1   196                        yes
2   392                        yes
3   588                        yes
4   784                        yes
5   980                        yes
6   1176                       yes
7   1372                       yes
```

Keluaran ini menegaskan, byte demi byte, gambar dalam **Gambar 1.6**. Struct itu menempati tepat 196 byte, field-field-nya bermula di offset 0, 6, 106, dan 166 (makro `offsetof` menanyakannya kepada kompilator), dan elemen-elemen array yang berurutan bermula tepat 196 byte terpisah, seperti diramalkan Persamaan (1.2) dengan $s = 196$. Kolom terakhir menegaskan bahwa `&catalog[i]` dan `catalog + i` adalah alamat yang sama untuk setiap `i`.



Gambar 1.6 Katalog sebagai delapan rekaman `Book` berukuran 196 byte yang disambung ujung ke ujung, dan isi satu rekaman: keempat field-nya bermula di offset 0, 6, 106, dan 166.

Mengapa tepat 196, tanpa *padding* seperti byte yang kita lihat dalam **Gambar 1.3**? Setiap field `Book` adalah array `char`, dan sebuah `char` boleh bermula di alamat mana pun, sehingga kompilator tidak punya apa-apa untuk diselaraskan. Soal Latihan 1.14 menambahkan sebuah field `int` dan mengamati *padding* itu muncul.

Java dan Python menyimpan katalog dengan cara lain, dan penting untuk memahami bagaimana. Di Java, `new Book[8]` membuat array berisi delapan *referensi*, semuanya `null` pada awalnya; setiap `new Book(...)` membuat objek terpisah di suatu tempat di *heap*, dan slot array menerima referensi ke objek itu. List Python pun demikian: sederet referensi ke objek-objek terpisah. **Kode 1.16**, bersama kelas dalam **Kode 1.15**, dan **Kode 1.18**, bersama modul dalam **Kode 1.17**, mencetak katalog yang sama; alih-alih offset, keduanya memperlihatkan bahwa menyalin sebuah elemen hanya menyalin referensinya.

Kode 1.15 `Book.java`: rekaman `Book` di Java, dengan field `String`.

```java
// Book.java - rekaman Pustaka Ceria, dipakai Catalog1D dan Shelves2D
public class Book {
    final String id;
    final String title;
    final String author;
    final String category;

    Book(String id, String title, String author, String category) {
        this.id = id;
        this.title = title;
        this.author = author;
        this.category = category;
    }

    static void printCatalogHeader() {
        System.out.printf("%-6s %-26s %-20s %s%n",
                "ID", "TITLE", "AUTHOR", "CATEGORY");
    }

    void print() {
        System.out.printf("%-6s %-26s %-20s %s%n", id, title, author, category);
    }
}
```

Kode 1.16 `Catalog1D.java`: katalog sebagai array Java berisi referensi `Book`.

```java
// Catalog1D.java - array satu dimensi berisi referensi Book
public class Catalog1D {
    static final int N = 8;

    public static void main(String[] args) {
        Book[] catalog = new Book[N];          // delapan slot, mula-mula null
        catalog[0] = new Book("B1001", "Laskar Pelangi",
                "Andrea Hirata", "Fiksi");
        catalog[1] = new Book("B1002", "Bumi Manusia",
                "Pramoedya A. Toer", "Fiksi");
        catalog[2] = new Book("B1003", "Filosofi Kopi",
                "Dee Lestari", "Fiksi");
        catalog[3] = new Book("B1004", "Sejarah Nusantara",
                "Slamet Muljana", "Sejarah");
        catalog[4] = new Book("B1005", "Algoritma & Struktur Data",
                "Rinaldi Munir", "Teknologi");
        catalog[5] = new Book("B1006", "Cerita Rakyat Nusantara",
                "Tim Pustaka Ceria", "Anak-anak");
        catalog[6] = new Book("B1007", "Fisika Dasar",
                "Halliday & Resnick", "Sains");
        catalog[7] = new Book("B1008", "Negeri 5 Menara",
                "Ahmad Fuadi", "Fiksi");

        System.out.println("Pustaka Ceria catalog, " + N + " books");
        Book.printCatalogHeader();
        for (int i = 0; i < N; i++) {
            catalog[i].print();
        }

        System.out.println("\ncatalog.length = " + catalog.length);
        System.out.println(
                "Each slot holds a reference; each Book is its own object.");
        Book alias = catalog[4];               // menyalin referensi, bukan buku
        System.out.println("alias == catalog[4] ? " + (alias == catalog[4]));
    }
}
```

Kode 1.17 `book.py`: rekaman `Book` di Python, sebagai *dataclass*.

```python
# book.py - rekaman Pustaka Ceria, dipakai catalog_1d.py dan shelves_2d.py
from dataclasses import dataclass


@dataclass
class Book:
    id: str
    title: str
    author: str
    category: str

    def print(self) -> None:
        print(f"{self.id:<6} {self.title:<26} {self.author:<20} {self.category}")


def print_catalog_header() -> None:
    print(f"{'ID':<6} {'TITLE':<26} {'AUTHOR':<20} CATEGORY")
```

Kode 1.18 `catalog_1d.py`: katalog sebagai list Python.

```python
# catalog_1d.py - list Python berisi referensi Book
from book import Book, print_catalog_header

catalog = [
    Book("B1001", "Laskar Pelangi", "Andrea Hirata", "Fiksi"),
    Book("B1002", "Bumi Manusia", "Pramoedya A. Toer", "Fiksi"),
    Book("B1003", "Filosofi Kopi", "Dee Lestari", "Fiksi"),
    Book("B1004", "Sejarah Nusantara", "Slamet Muljana", "Sejarah"),
    Book("B1005", "Algoritma & Struktur Data", "Rinaldi Munir", "Teknologi"),
    Book("B1006", "Cerita Rakyat Nusantara", "Tim Pustaka Ceria", "Anak-anak"),
    Book("B1007", "Fisika Dasar", "Halliday & Resnick", "Sains"),
    Book("B1008", "Negeri 5 Menara", "Ahmad Fuadi", "Fiksi"),
]

print(f"Pustaka Ceria catalog, {len(catalog)} books")
print_catalog_header()
for book in catalog:
    book.print()

print(f"\nlen(catalog) = {len(catalog)}")
print("Each slot holds a reference; each Book is its own object.")
alias = catalog[4]                      # menyalin referensi, bukan buku
print(f"alias is catalog[4] ? {alias is catalog[4]}")
```

Program Java mencetak katalog yang sama dengan program C++, disusul:

```
catalog.length = 8
Each slot holds a reference; each Book is its own object.
alias == catalog[4] ? true
```

dan program Python mencetak katalog yang sama disusul:

```
len(catalog) = 8
Each slot holds a reference; each Book is its own object.
alias is catalog[4] ? True
```

Jadi array buku di Java dan Python adalah deretan *referensi* yang bersinambung, masing-masing delapan byte, sedangkan bukunya sendiri tersebar di seluruh *heap*. Rumus Persamaan (1.2) tetap menemukan *referensi* mana pun dalam satu langkah; satu langkah lagi mengikutinya sampai ke buku.

1.5.4 Tetap di dalam batas

Apa yang terjadi bila program meminta `loans[8]`, atau `loans[-1]`, pada array berisi delapan elemen? Jawabannya adalah perbedaan terpenting antara C++ di satu sisi dan Java serta Python di sisi lain.

C++ tidak memeriksa. Ekspresi `loans[8]` sekadar menghitung $b + 8 \cdot 4$ lalu membaca byte apa pun yang ada di sana: sebagian variabel lain, sebagian *stack frame*, atau memori yang sama sekali bukan milik program. Standar C++ menyebut ini **perilaku tak terdefinisi** (*undefined behavior*): apa pun boleh terjadi, termasuk, yang paling berbahaya, tidak terjadi apa-apa yang tampak. Program mungkin mencetak bilangan aneh, macet jauh kemudian di tempat yang tak berhubungan, atau berjalan benar di mesin Anda dan gagal di mesin lain. Menulis melewati ujung array adalah cara terciptanya beberapa lubang keamanan paling terkenal dalam sejarah, karena penyerang yang dapat memilih apa yang dituliskan dapat menimpa data, atau alamat kembali, sebuah *stack frame*.

C++ memberi kita dua pertahanan. Yang pertama adalah disiplin: setiap perulangan atas array berjalan `for (int i = 0; i < n; i++)`, dengan `<` dan tidak pernah `<=`. Yang kedua adalah pengakses berpemeriksa `at(i)` milik wadah standar `std::array` dan `std::vector` (Subbab 1.5.8), yang memverifikasi indeks dan *melempar eksepsi* bertipe `out_of_range` bila indeks itu tidak sah. **Kode 1.19** memakai keduanya.

Kode 1.19 `bounds.cpp`: pengindeksan di dalam batas, dan pengakses berpemeriksa `at()` yang menolak indeks di luar jangkauan.

```
// bounds.cpp - tetap di dalam array, dan apa yang dilakukan pengakses berpemeriksa
#include <array>
#include <iostream>
#include <stdexcept>
#include <vector>
using namespace std;

int main() {
    array<int, 5> a = {10, 20, 30, 40, 50};
    vector<int> v = {10, 20, 30, 40, 50};

    cout << "a.size() = " << a.size() << ", valid indexes 0.." << a.size() - 1
         << '\n';
    cout << "a[0] = " << a[0] << ", a[4] = " << a[4] << '\n';

    for (size_t i : {3, 5}) {
        try {
            int value = v.at(i);          // melempar eksepsi bila i tidak sah
            cout << "v.at(" << i << ") = " << value << '\n';
        } catch (const out_of_range& e) {
            cout << "v.at(" << i << ") refused: " << e.what() << '\n';
        }
    }
    // v[5] lolos kompilasi, tetapi membacanya perilaku tak terdefinisi.
    return 0;
}
```

```
a.size() = 5, valid indexes 0..4
a[0] = 10, a[4] = 50
v.at(3) = 40
v.at(5) refused: vector::_M_range_check: __n (which is 5) >= this->size() (which is 5)
```

Blok `try` menjalankan kode yang mungkin melempar eksepsi; jika itu terjadi, kendali melompat ke blok `catch` yang cocok, dan `e.what()` menguraikan masalahnya. Pesan itu ditulis oleh pustaka,

dan yang tampak adalah pesan pustaka GCC; kompilator lain menyusun kalimatnya secara berbeda.

Java dan Python selalu memeriksa. Setiap indeks dibandingkan dengan panjang array sebelum dipakai, dan indeks yang tidak sah memunculkan eksepsi, `ArrayIndexOutOfBoundsException` di Java dan `IndexError` di Python, tepat di baris tempat kesalahan dibuat. **Kode 1.20** dan **Kode 1.21** memperlihatkannya.

Kode 1.20 `Bounds.java`: Java memeriksa setiap indeks, untuk array maupun `ArrayList`.

```java
// Bounds.java - Java memeriksa setiap indeks dan melempar eksepsi
import java.util.ArrayList;
import java.util.List;

public class Bounds {
    public static void main(String[] args) {
        int[] a = {10, 20, 30, 40, 50};
        List<Integer> v = new ArrayList<>(List.of(10, 20, 30, 40, 50));

        System.out.println("a.length = " + a.length + ", valid indexes 0.."
                + (a.length - 1));
        System.out.println("a[0] = " + a[0] + ", a[4] = " + a[4]);

        for (int i : new int[] {3, 5}) {
            try {
                int value = a[i];             // melempar eksepsi bila i tidak sah
                System.out.println("a[" + i + "] = " + value);
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("a[" + i + "] refused: " + e.getMessage());
            }
        }
        try {
            System.out.println("v.get(5) = " + v.get(5));
        } catch (IndexOutOfBoundsException e) {
            System.out.println("v.get(5) refused: " + e.getMessage());
        }
    }
}
```

```
a.length = 5, valid indexes 0..4
a[0] = 10, a[4] = 50
a[3] = 40
a[5] refused: Index 5 out of bounds for length 5
v.get(5) refused: Index 5 out of bounds for length 5
```

Kode 1.21 `bounds.py`: Python memeriksa setiap indeks, dan juga menerima indeks negatif.

```python
# bounds.py - Python memeriksa setiap indeks, dan indeks negatif dihitung dari ujung
a = [10, 20, 30, 40, 50]

print(f"len(a) = {len(a)}, valid indexes 0..{len(a) - 1} (and -1..-{len(a)})")
print(f"a[0] = {a[0]}, a[4] = {a[4]}")

for i in (3, 5):
    try:
        value = a[i]                      # memunculkan IndexError bila i tidak sah
        print(f"a[{i}] = {value}")
    except IndexError as e:
        print(f"a[{i}] refused: {e}")
print(f"a[-1] = {a[-1]}, a[-5] = {a[-5]}")
```

```
len(a) = 5, valid indexes 0..4 (and -1..-5)
a[0] = 10, a[4] = 50
a[3] = 40
a[5] refused: list index out of range
a[-1] = 50, a[-5] = 10
```

Python menambahkan satu kekhasan: indeks *negatif* dihitung dari ujung, sehingga `a[-1]` adalah elemen terakhir dan `a[-5]`, dalam list berisi lima, adalah elemen pertama. Indeks sah list sepanjang n karena itu adalah $-n$ sampai $n-1$. Ini sebuah kemudahan di Python, sekaligus jebakan bagi siapa saja yang membawa kebiasaan itu kembali ke C++, tempat `a[-1]` sekadar memori tepat *sebelum* array.

1.5.5 Menelusuri array: penelusuran sehari-hari

Banyak sekali algoritma array adalah perulangan yang sama dengan isi yang berbeda: kunjungi setiap elemen sekali, dari indeks 0 sampai $n-1$, dan lakukan sesuatu terhadapnya. Ini disebut **penelusuran** (*traversal*). **Kode 1.22** menghimpun lima penelusuran yang semestinya dapat ditulis setiap pemrogram tanpa berpikir: jumlah, posisi elemen terbesar, posisi pertama sebuah nilai, berapa kali sebuah nilai muncul, dan membalik array di tempat.

Kode 1.22 `array_ops.cpp`: jumlah, posisi maksimum, pencarian linear, pencacahan, dan pembalikan array di tempat.

```cpp
// array_ops.cpp - penelusuran sehari-hari pada array satu dimensi
#include <iostream>
using namespace std;

long sumOf(const int a[], int n) {
    long s = 0;
    for (int i = 0; i < n; i++) s += a[i];
    return s;
}

int indexOfMax(const int a[], int n) {
    int best = 0;
    for (int i = 1; i < n; i++) {
        if (a[i] > a[best]) best = i;
    }
    return best;
}

int findFirst(const int a[], int n, int key) {       // -1 berarti "tidak ditemukan"
    for (int i = 0; i < n; i++) {
        if (a[i] == key) return i;
    }
    return -1;
}

int countOf(const int a[], int n, int key) {
    int c = 0;
    for (int i = 0; i < n; i++) {
        if (a[i] == key) c++;
    }
    return c;
}

void reverseInPlace(int a[], int n) {
    for (int i = 0, j = n - 1; i < j; i++, j--) {
        int t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
}
```

Kode 1.22 (Lanjutan) *array_ops.cpp: jumlah, posisi maksimum, pencarian linear, pencacahan, dan pembalikan array di tempat.*

```cpp
void print(const int a[], int n) {
    cout << '[';
    for (int i = 0; i < n; i++) cout << (i ? ", " : "") << a[i];
    cout << "]\n";
}

int main() {
    int loans[] = {12, 9, 15, 4, 20, 11, 6, 15};
    int n = sizeof loans / sizeof loans[0];    // berlaku di sini, tidak di fungsi

    cout << "loans      = "; print(loans, n);
    cout << "n          = " << n << '\n';
    cout << "sum        = " << sumOf(loans, n) << '\n';
    int m = indexOfMax(loans, n);
    cout << "max        = " << loans[m] << " at index " << m << '\n';
    cout << "find 15    = " << findFirst(loans, n, 15) << '\n';
    cout << "find 99    = " << findFirst(loans, n, 99) << '\n';
    cout << "count 15   = " << countOf(loans, n, 15) << '\n';
    reverseInPlace(loans, n);
    cout << "reversed   = "; print(loans, n);
    return 0;
}
```

```
loans      = [12, 9, 15, 4, 20, 11, 6, 15]
n          = 8
sum        = 92
max        = 20 at index 4
find 15    = 2
find 99    = -1
count 15   = 2
reversed   = [15, 6, 11, 20, 4, 15, 9, 12]
```

Tiga rincian membuat fungsi-fungsi ini benar, dan masing-masing adalah pelajaran yang diandalkan bab-bab berikutnya.

Array yang dioper ke fungsi tidak membawa panjangnya. Parameter yang ditulis `const int a[]` bagi kompilator hanyalah pointer `const int* a` ke elemen pertama; fungsi tidak dapat mengetahui berapa elemen yang menyusul. Itulah sebabnya setiap fungsi menerima `n` sebagai parameter kedua. Di `main`, idiom `sizeof loans / sizeof loans[0]` menghitung panjangnya, $32/4 = 8$, karena di sana `loans` masih array sungguhan yang ukurannya diketahui kompilator; di dalam fungsi, ekspresi yang sama akan membagi ukuran sebuah *pointer* dengan ukuran sebuah `int`, dan memberikan jawaban yang salah. Kata `const` menjanjikan bahwa fungsi tidak akan mengubah array itu, dan kompilator menagih janji tersebut.

`findFirst` mengembalikan -1 untuk "tidak ditemukan". Karena setiap indeks sah bernilai 0 atau lebih, -1 tidak mungkin tertukar dengan sebuah posisi. Konvensi ini, "sebuah posisi, atau -1", adalah kontrak setiap algoritma pencarian di Bab 2.

`reverseInPlace` tidak memerlukan array kedua. Dua indeks bermula di kedua ujung, `i = 0` dan `j = n - 1`, menukar elemen-elemennya, lalu melangkah saling mendekat sampai bertemu. Setiap penukaran memakai variabel sementara `t`, karena menulis `a[i] = a[j]; a[j] = a[i];` akan menghilangkan `a[i]` yang lama. Array berisi n elemen memerlukan $\lfloor n/2 \rfloor$ penukaran, dengan $\lfloor x \rfloor$, *floor* (lantai) dari x, berarti x yang dibulatkan ke bawah menjadi bilangan bulat.

Array Java mengetahui panjangnya sendiri, `a.length`, sehingga fungsi-fungsi Java dalam **Kode 1.23** tidak memerlukan parameter `n`; list Python mengetahui panjangnya melalui `len(a)`,

dan Python menukar dua elemen dalam satu baris, `a[i], a[j] = a[j], a[i]`, seperti diperlihatkan **Kode 1.24**. Keduanya mencetak keluaran yang persis sama dengan di atas.

Kode 1.23 `ArrayOps.java`: kelima penelusuran di Java.

```java
// ArrayOps.java - penelusuran sehari-hari pada array satu dimensi
import java.util.Arrays;

public class ArrayOps {
    static long sumOf(int[] a) {
        long s = 0;
        for (int x : a) s += x;
        return s;
    }

    static int indexOfMax(int[] a) {
        int best = 0;
        for (int i = 1; i < a.length; i++) {
            if (a[i] > a[best]) best = i;
        }
        return best;
    }

    static int findFirst(int[] a, int key) {  // -1 berarti "tidak ditemukan"
        for (int i = 0; i < a.length; i++) {
            if (a[i] == key) return i;
        }
        return -1;
    }

    static int countOf(int[] a, int key) {
        int c = 0;
        for (int x : a) {
            if (x == key) c++;
        }
        return c;
    }

    static void reverseInPlace(int[] a) {
        for (int i = 0, j = a.length - 1; i < j; i++, j--) {
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }

    public static void main(String[] args) {
        int[] loans = {12, 9, 15, 4, 20, 11, 6, 15};
        int n = loans.length;                // array mengetahui panjangnya sendiri
```

Kode 1.23 (Lanjutan) *ArrayOps.java: kelima penelusuran di Java.*

```java
        System.out.println("loans      = " + Arrays.toString(loans));
        System.out.println("n          = " + n);
        System.out.println("sum        = " + sumOf(loans));
        int m = indexOfMax(loans);
        System.out.println("max        = " + loans[m] + " at index " + m);
        System.out.println("find 15    = " + findFirst(loans, 15));
        System.out.println("find 99    = " + findFirst(loans, 99));
        System.out.println("count 15   = " + countOf(loans, 15));
        reverseInPlace(loans);
        System.out.println("reversed   = " + Arrays.toString(loans));
    }
}
```

Kode 1.24 `array_ops.py`: kelima penelusuran di Python.

```
# array_ops.py - penelusuran sehari-hari pada list satu dimensi
def sum_of(a):
    s = 0
    for x in a:
        s += x
    return s


def index_of_max(a):
    best = 0
    for i in range(1, len(a)):
        if a[i] > a[best]:
            best = i
    return best


def find_first(a, key):                 # -1 berarti "tidak ditemukan"
    for i in range(len(a)):
        if a[i] == key:
            return i
    return -1


def count_of(a, key):
    c = 0
    for x in a:
        if x == key:
            c += 1
    return c


def reverse_in_place(a):
    i, j = 0, len(a) - 1
    while i < j:
        a[i], a[j] = a[j], a[i]
        i += 1
        j -= 1


loans = [12, 9, 15, 4, 20, 11, 6, 15]
n = len(loans)                          # list mengetahui panjangnya sendiri
```

Kode 1.24 (Lanjutan) *array_ops.py: kelima penelusuran di Python.*

```
print(f"loans      = {loans}")
print(f"n          = {n}")
print(f"sum        = {sum_of(loans)}")
m = index_of_max(loans)
print(f"max        = {loans[m]} at index {m}")
print(f"find 15    = {find_first(loans, 15)}")
print(f"find 99    = {find_first(loans, 99)}")
print(f"count 15   = {count_of(loans, 15)}")
reverse_in_place(loans)
print(f"reversed   = {loans}")
```

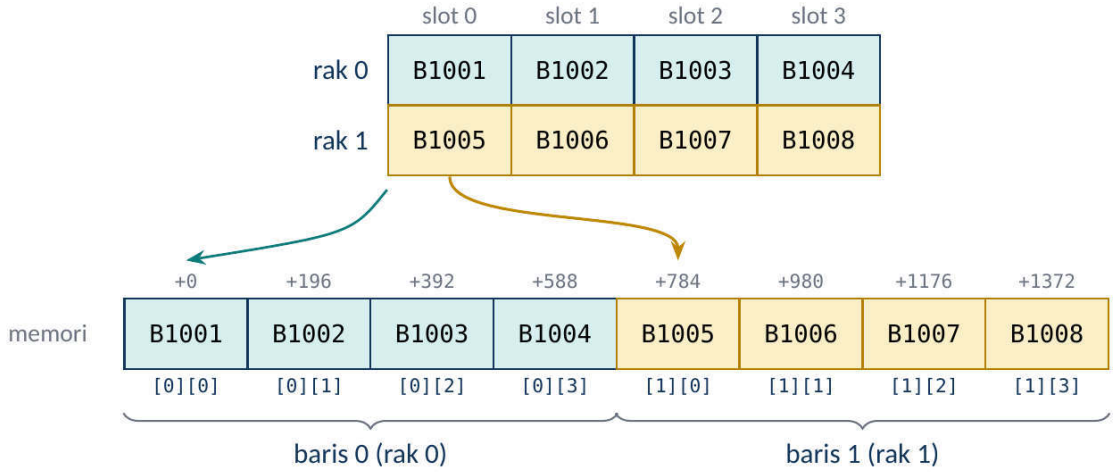
1.5.6 Array dua dimensi dan tata letak row-major

Gedung Pustaka Ceria mempunyai rak-rak sungguhan, dan setiap rak mempunyai sejumlah tetap slot. Katalog satu dimensi tidak menangkap bentuk itu; **array dua dimensi** menangkapnya. Di C++,

```
Book shelf[2][4];      // 2 rak, masing-masing 4 slot
shelf[1][2]            // rak 1, slot 2
```

mendeklarasikan kisi berisi $2 \times 4 = 8$ buku. Indeks pertama memilih **baris** (raknya), indeks kedua memilih **kolom** (slotnya). Baris dan kolom sama-sama dinomori dari 0.

Namun memori adalah satu deretan byte, bukan kisi. Bagaimana kisi dapat muat dalam satu deretan? C++ menata array dua dimensi dalam **urutan row-major** (baris demi baris): seluruh baris 0 lebih dahulu, lalu seluruh baris 1 tepat sesudahnya, dan seterusnya, persis seolah-olah kisi itu digelar menjadi satu array satu dimensi yang panjang, baris demi baris [7], [8]. **Gambar 1.7** memperlihatkannya.



Gambar 1.7 Array `shelf` berukuran 2×4 sebagai kisi, dan sebagaimana ia sungguh-sungguh terbaring di memori: baris 0 lebih dahulu, lalu baris 1, setiap elemen berjarak 196 byte dari elemen sebelumnya.

Dalam deretan yang digelar itu, elemen `shelf[s][p]` didahului s baris utuh yang masing-masing berisi `SLOTS` elemen, ditambah p elemen di barisnya sendiri. Posisinya dalam deretan itu karena itu $s \cdot \text{SLOTS} + p$, dan menurut Persamaan (1.2) alamatnya adalah

$$\text{alamat}(A[r][c]) = b + (r \cdot C + c) \cdot s \tag{1.3}$$

dengan C banyaknya kolom, r dan c indeks baris dan kolom, b alamat dasar, dan s ukuran satu elemen. Perhatikan apa yang *tidak* muncul dalam Persamaan (1.3): banyaknya baris. Untuk menemukan sebuah elemen, kompilator perlu tahu panjang setiap baris, tetapi tidak perlu tahu berapa banyak barisnya. Itulah sebabnya fungsi C++ yang menerima array dua dimensi harus diberi tahu banyaknya kolom di parameternya, `void f(Book a[][4], int rows)`, sedangkan banyaknya baris boleh dioper terpisah.

Kode 1.25 mengisi rak-rak dengan kedelapan buku yang sama dan memeriksa Persamaan (1.3) untuk setiap elemen.

Kode 1.25 `shelves_2d.cpp`: array dua dimensi berisi `Book`, dan rumus row-major yang diperiksa untuk setiap elemen.

```cpp
// shelves_2d.cpp - array dua dimensi berisi Book dan tata letak row-major-nya
#include <cstdio>
#include "book.h"

const int SHELVES = 2;
const int SLOTS = 4;

int main() {
    Book shelf[SHELVES][SLOTS];
    populateBook(shelf[0][0], "B1001", "Laskar Pelangi",
                 "Andrea Hirata", "Fiksi");
    populateBook(shelf[0][1], "B1002", "Bumi Manusia",
                 "Pramoedya A. Toer", "Fiksi");
    populateBook(shelf[0][2], "B1003", "Filosofi Kopi",
                 "Dee Lestari", "Fiksi");
    populateBook(shelf[0][3], "B1004", "Sejarah Nusantara",
                 "Slamet Muljana", "Sejarah");
    populateBook(shelf[1][0], "B1005", "Algoritma & Struktur Data",
                 "Rinaldi Munir", "Teknologi");
    populateBook(shelf[1][1], "B1006", "Cerita Rakyat Nusantara",
                 "Tim Pustaka Ceria", "Anak-anak");
    populateBook(shelf[1][2], "B1007", "Fisika Dasar",
                 "Halliday & Resnick", "Sains");
    populateBook(shelf[1][3], "B1008", "Negeri 5 Menara",
                 "Ahmad Fuadi", "Fiksi");

    for (int s = 0; s < SHELVES; s++) {
        std::printf("shelf %d:", s);
        for (int p = 0; p < SLOTS; p++) std::printf(" %s", shelf[s][p].id);
        std::printf("\n");
    }

    const char* base = reinterpret_cast<const char*>(&shelf[0][0]);
    std::printf("\nsizeof(Book) = %zu, SLOTS = %d\n", sizeof(Book), SLOTS);
    std::printf("%-2s %-2s %-6s %-10s %-24s %s\n",
                "s", "p", "id", "offset", "(s*SLOTS+p)*sizeof(Book)", "same?");
    for (int s = 0; s < SHELVES; s++) {
        for (int p = 0; p < SLOTS; p++) {
            const char* here = reinterpret_cast<const char*>(&shelf[s][p]);
            long actual = static_cast<long>(here - base);
            long formula = static_cast<long>((s * SLOTS + p) * sizeof(Book));
            std::printf("%-2d %-2d %-6s %-10ld %-24ld %s\n", s, p, shelf[s][p].id,
                        actual, formula, actual == formula ? "yes" : "NO");
        }
    }
    return 0;
```

Kode 1.25 (Lanjutan) *shelves_2d.cpp: array dua dimensi berisi Book, dan rumus row-major yang diperiksa untuk setiap elemen.*

```
}
```

```
shelf 0: B1001 B1002 B1003 B1004
shelf 1: B1005 B1006 B1007 B1008

sizeof(Book) = 196, SLOTS = 4
s  p  id     offset     (s*SLOTS+p)*sizeof(Book) same?
0  0  B1001  0          0                        yes
0  1  B1002  196        196                      yes
0  2  B1003  392        392                      yes
0  3  B1004  588        588                      yes
1  0  B1005  784        784                      yes
1  1  B1006  980        980                      yes
1  2  B1007  1176       1176                     yes
1  3  B1008  1372       1372                     yes
```

Ambillah satu baris dan kerjakan hitungannya sendiri: `shelf[1][2]` berisi B1007, dan $(1 \cdot 4 + 2) \cdot 196 = 6 \cdot 196 = 1176$, persis offset yang terukur.

Urutan row-major adalah sebuah pilihan, bukan hukum alam. Bahasa Fortran, beserta pustaka-pustaka numerik turunannya, memakai urutan **column-major** (kolom demi kolom), menyimpan kolom 0 lebih dahulu, lalu kolom 1. Untuk array dengan R baris, alamat column-major-nya adalah

$$\text{alamat}(A[r][c]) = b + (c \cdot R + r) \cdot s \tag{1.4}$$

Kode yang menghitung alamat dengan tangan, atau yang mengoper array antarbahasa, harus tahu urutan mana yang sedang dihadapinya; mencampuradukkannya adalah sumber *bug* yang nyata dan mudah terlewat. Soal Latihan 1.4 melatih kedua rumus itu.

Java dan Python tidak mempunyai array dua dimensi yang sejati. `Book[][]` di Java adalah *array berisi array*: array berisi dua referensi, masing-masing merujuk ke array baris terpisah yang berisi empat referensi. List berisi list di Python adalah gagasan yang sama. Baris-barisnya bahkan tidak harus sama panjang, dan di Java array seperti itu disebut **bergerigi** (*jagged*). Karena itu **Kode 1.26** dan **Kode 1.27** tidak dapat mengukur offset; keduanya mencetak rak-rak dan *posisi* row-major $s \cdot \text{SLOTS} + p$ yang akan dipakai Persamaan (1.3) seandainya kisi itu dipadatkan menjadi satu deretan.

Kode 1.26 `Shelves2D.java`: array Java berisi array baris.

```java
// Shelves2D.java - array dua dimensi: array berisi array baris
public class Shelves2D {
    static final int SHELVES = 2;
    static final int SLOTS = 4;

    public static void main(String[] args) {
        Book[][] shelf = new Book[SHELVES][SLOTS];
        shelf[0][0] = new Book("B1001", "Laskar Pelangi",
                "Andrea Hirata", "Fiksi");
        shelf[0][1] = new Book("B1002", "Bumi Manusia",
                "Pramoedya A. Toer", "Fiksi");
        shelf[0][2] = new Book("B1003", "Filosofi Kopi",
                "Dee Lestari", "Fiksi");
        shelf[0][3] = new Book("B1004", "Sejarah Nusantara",
                "Slamet Muljana", "Sejarah");
        shelf[1][0] = new Book("B1005", "Algoritma & Struktur Data",
                "Rinaldi Munir", "Teknologi");
        shelf[1][1] = new Book("B1006", "Cerita Rakyat Nusantara",
                "Tim Pustaka Ceria", "Anak-anak");
        shelf[1][2] = new Book("B1007", "Fisika Dasar",
                "Halliday & Resnick", "Sains");
        shelf[1][3] = new Book("B1008", "Negeri 5 Menara",
                "Ahmad Fuadi", "Fiksi");

        for (int s = 0; s < SHELVES; s++) {
            System.out.print("shelf " + s + ":");
            for (int p = 0; p < SLOTS; p++) System.out.print(" " + shelf[s][p].id);
            System.out.println();
        }

        System.out.println("\nshelf.length = " + shelf.length
                + " rows, shelf[0].length = " + shelf[0].length + " slots");
        System.out.printf("%-2s %-2s %-6s %s%n",
                "s", "p", "id", "row-major position s*SLOTS+p");
        for (int s = 0; s < SHELVES; s++) {
            for (int p = 0; p < SLOTS; p++) {
                System.out.printf("%-2d %-2d %-6s %d%n",
                        s, p, shelf[s][p].id, s * SLOTS + p);
            }
        }
    }
}
```

```
shelf 0: B1001 B1002 B1003 B1004
shelf 1: B1005 B1006 B1007 B1008

shelf.length = 2 rows, shelf[0].length = 4 slots
s  p  id     row-major position s*SLOTS+p
0  0  B1001  0
0  1  B1002  1
0  2  B1003  2
0  3  B1004  3
1  0  B1005  4
1  1  B1006  5
1  2  B1007  6
1  3  B1008  7
```

Kode 1.27 `shelves_2d.py`: list Python berisi list baris.

```python
# shelves_2d.py - list dua dimensi: list berisi list baris
from book import Book

SHELVES = 2
SLOTS = 4

shelf = [
    [Book("B1001", "Laskar Pelangi", "Andrea Hirata", "Fiksi"),
     Book("B1002", "Bumi Manusia", "Pramoedya A. Toer", "Fiksi"),
     Book("B1003", "Filosofi Kopi", "Dee Lestari", "Fiksi"),
     Book("B1004", "Sejarah Nusantara", "Slamet Muljana", "Sejarah")],
    [Book("B1005", "Algoritma & Struktur Data", "Rinaldi Munir", "Teknologi"),
     Book("B1006", "Cerita Rakyat Nusantara", "Tim Pustaka Ceria", "Anak-anak"),
     Book("B1007", "Fisika Dasar", "Halliday & Resnick", "Sains"),
     Book("B1008", "Negeri 5 Menara", "Ahmad Fuadi", "Fiksi")],
]

for s in range(SHELVES):
    print(f"shelf {s}:", " ".join(book.id for book in shelf[s]))

print(f"\nlen(shelf) = {len(shelf)} rows, len(shelf[0]) = {len(shelf[0])} slots")
print(f"{'s':<2} {'p':<2} {'id':<6} row-major position s*SLOTS+p")
for s in range(SHELVES):
    for p in range(SLOTS):
        print(f"{s:<2} {p:<2} {shelf[s][p].id:<6} {s * SLOTS + p}")
```

Program Python mencetak baris-baris yang sama, dengan `len(shelf)` dan `len(shelf[0])` sebagai ganti `shelf.length` dan `shelf[0].length`.

1.5.7 Tabel bilangan

Array dua dimensi yang paling umum berisi bilangan: tabel hasil pengukuran, lembar kerja, papan permainan, piksel sebuah gambar. Pustaka Ceria mencatat berapa buku yang dipinjam dari masing-masing tiga rak dalam setiap minggu dari empat minggu, dan ingin mengetahui total setiap baris, setiap kolom, dan seluruh tabel. **Kode 1.28** mengerjakannya dengan dua perulangan bersarang: perulangan luar menelusuri baris, perulangan dalam menelusuri kolom-kolom satu baris.

Kode 1.28 `loan_table.cpp`: tabel peminjaman 3×4 dengan total baris, total kolom, dan total keseluruhan.

```cpp
// loan_table.cpp - array 2-D berisi bilangan: total baris dan total kolom
#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

const int ROWS = 3;                       // tiga rak
const int COLS = 4;                       // empat minggu

int main() {
    string shelfName[ROWS] = {"Fiksi", "Sejarah", "Sains"};
    int loans[ROWS][COLS] = {
        {31, 28, 35, 40},
        {12, 15, 9, 14},
        {20, 22, 18, 25},
    };

    cout << left << setw(8) << "shelf";
    for (int c = 0; c < COLS; c++) {
        cout << right << setw(5) << ("W" + to_string(c + 1));
    }
    cout << setw(7) << "total" << '\n';

    int columnTotal[COLS] = {0};          // setiap elemen bermula dari 0
    int grandTotal = 0;
    for (int r = 0; r < ROWS; r++) {
        int rowTotal = 0;
        cout << left << setw(8) << shelfName[r];
        for (int c = 0; c < COLS; c++) {
            cout << right << setw(5) << loans[r][c];
            rowTotal += loans[r][c];
            columnTotal[c] += loans[r][c];
        }
        cout << setw(7) << rowTotal << '\n';
        grandTotal += rowTotal;
    }
    cout << left << setw(8) << "total";
    for (int c = 0; c < COLS; c++) cout << right << setw(5) << columnTotal[c];
    cout << setw(7) << grandTotal << '\n';
    return 0;
}
```

```
shelf      W1   W2   W3   W4  total
Fiksi      31   28   35   40    134
Sejarah    12   15    9   14     50
Sains      20   22   18   25     85
total      63   65   62   79    269
```

Total baris diakumulasikan dalam variabel `rowTotal`, yang dinolkan ulang untuk setiap baris; total kolom memerlukan satu total berjalan per kolom, sehingga disimpan dalam array `columnTotal`, yang dideklarasikan dengan `= {0}` sehingga setiap elemennya bernilai nol. Manipulator `left`, `right`, dan `setw(k)` mencetak setiap nilai dalam kolom selebar *k* karakter.

Perulangan-perulangan itu mengunjungi tabel baris demi baris, dalam urutan yang sama dengan urutan elemen-elemen berbaring di memori. Itu bukan hanya wajar tetapi juga cepat: prosesor modern mengambil memori per blok 64 byte, yang disebut *cache line*, sehingga membaca alamat-alamat berurutan jauh lebih cepat daripada melompat ke sana kemari. Untuk tabel kecil perbedaannya tak terlihat; untuk tabel berisi jutaan bilangan, menukar kedua perulangan sehingga perulangan dalam berjalan menuruni sebuah kolom dapat membuat program yang sama beberapa kali lebih lambat [5].

Versi Java dan Python dalam **Kode 1.29** dan **Kode 1.30** mencetak tabel yang sama.

Kode 1.29 `LoanTable.java`: tabel peminjaman di Java.

```java
// LoanTable.java - array 2-D berisi bilangan: total baris dan total kolom
public class LoanTable {
    static final int ROWS = 3;                // tiga rak
    static final int COLS = 4;                // empat minggu

    public static void main(String[] args) {
        String[] shelfName = {"Fiksi", "Sejarah", "Sains"};
        int[][] loans = {
            {31, 28, 35, 40},
            {12, 15, 9, 14},
            {20, 22, 18, 25},
        };

        System.out.printf("%-8s", "shelf");
        for (int c = 0; c < COLS; c++) System.out.printf("%5s", "W" + (c + 1));
        System.out.printf("%7s%n", "total");

        int[] columnTotal = new int[COLS];    // setiap elemen bermula dari 0
        int grandTotal = 0;
        for (int r = 0; r < ROWS; r++) {
            int rowTotal = 0;
            System.out.printf("%-8s", shelfName[r]);
            for (int c = 0; c < COLS; c++) {
                System.out.printf("%5d", loans[r][c]);
                rowTotal += loans[r][c];
                columnTotal[c] += loans[r][c];
            }
            System.out.printf("%7d%n", rowTotal);
            grandTotal += rowTotal;
        }
        System.out.printf("%-8s", "total");
        for (int c = 0; c < COLS; c++) System.out.printf("%5d", columnTotal[c]);
        System.out.printf("%7d%n", grandTotal);
    }
}
```

Kode 1.30 `loan_table.py`: tabel peminjaman di Python.

```
# loan_table.py - list 2-D berisi bilangan: total baris dan total kolom
ROWS = 3                                    # tiga rak
COLS = 4                                    # empat minggu

shelf_name = ["Fiksi", "Sejarah", "Sains"]
loans = [
    [31, 28, 35, 40],
    [12, 15, 9, 14],
    [20, 22, 18, 25],
]

header = "".join(f"{'W' + str(c + 1):>5}" for c in range(COLS))
print(f"{'shelf':<8}" + header + f"{'total':>7}")

column_total = [0] * COLS                   # setiap elemen bermula dari 0
grand_total = 0
for r in range(ROWS):
    row_total = 0
    line = f"{shelf_name[r]:<8}"
    for c in range(COLS):
        line += f"{loans[r][c]:>5}"
        row_total += loans[r][c]
        column_total[c] += loans[r][c]
    print(line + f"{row_total:>7}")
    grand_total += row_total
totals = "".join(f"{t:>5}" for t in column_total)
print(f"{'total':<8}" + totals + f"{grand_total:>7}")
```

Python mempunyai satu jebakan pada tabel yang pernah menjerat hampir setiap pemrogram Python. Ekspresi `[0] * 4` membuat list berisi empat nol, dan menggoda untuk membuat tabel 3×4 sebagai `[[0] * 4] * 3`. Namun `*` pada list mengulang *referensi*: hasilnya list berisi tiga referensi ke baris yang *sama*. **Kode 1.31** memperlihatkan apa yang terjadi bila satu elemen diubah.

Kode 1.31 `aliasing_rows.py`: `[[0] * 4] * 3` menghasilkan tiga referensi ke satu baris; *list comprehension* menghasilkan tiga baris.

```
# aliasing_rows.py - mengapa [[0] * 4] * 3 bukan tabel 3 x 4
wrong = [[0] * 4] * 3                       # tiga referensi ke SATU baris
right = [[0] * 4 for _ in range(3)]         # tiga baris terpisah

wrong[0][1] = 9
right[0][1] = 9
print("wrong:", wrong)
print("right:", right)
print("wrong rows are one object:", wrong[0] is wrong[1] is wrong[2])
```

```
wrong: [[0, 9, 0, 0], [0, 9, 0, 0], [0, 9, 0, 0]]
right: [[0, 9, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
wrong rows are one object: True
```

Mengubah `wrong[0][1]` tampak mengubah ketiga baris, karena memang hanya ada satu baris. Idiom yang benar, `[[0] * 4 for _ in range(3)]`, menjalankan `[0] * 4` tiga kali dan menghasilkan tiga baris terpisah.

1.5.8 `std::array` dan `std::vector`

Array yang kita pakai sejauh ini adalah array yang diwarisi C++ dari C, yang sering disebut array **mentah** atau **bawaan**. Array ini secepat yang dimungkinkan memori, dan mempunyai tiga kelemahan: ia lupa panjangnya ketika dioper ke fungsi, ia tidak dapat disalin dengan `=`, dan

ukurannya ditetapkan ketika program dikompilasi. Pustaka standar C++ menawarkan dua wadah yang mempertahankan kecepatannya dan menghapus kelemahan-kelemahan itu [9].

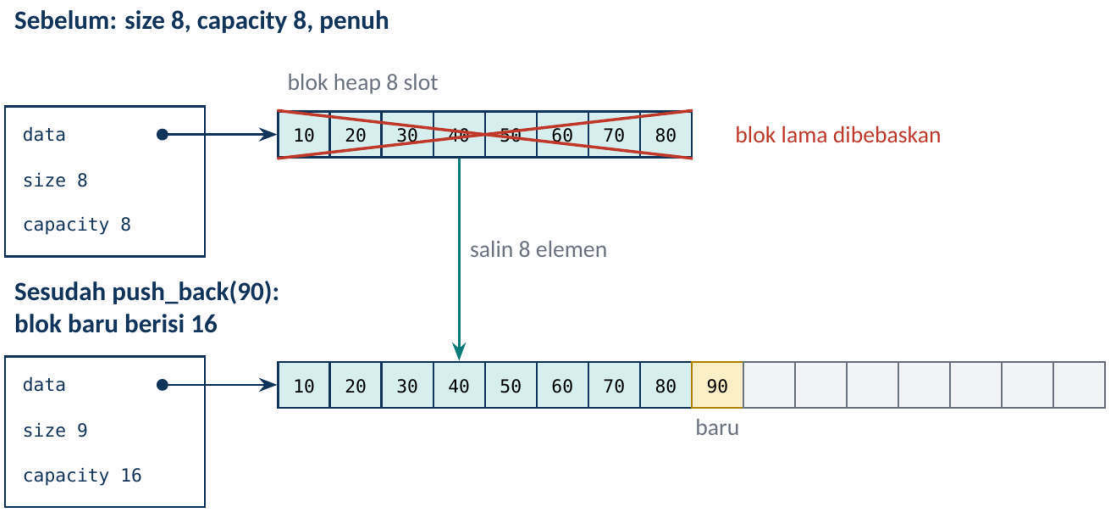
`std::array<T, N>`, dari `<array>`, adalah array berukuran tetap berisi `N` elemen bertipe `T` yang mengetahui ukurannya sendiri (`a.size()`), dapat disalin dan dibandingkan dengan `=` dan `==`, menyediakan pengakses berpemeriksa `a.at(i)`, dan tidak menambah biaya apa pun: elemen-elemennya berbaring di memori persis seperti elemen array mentah. Di mana pun buku ini memerlukan array yang ukurannya diketahui sejak awal, `std::array` adalah pilihan yang lebih baik.

`std::vector<T>`, dari `<vector>`, adalah array yang ukurannya dapat berubah selagi program berjalan. `v.push_back(x)` menambahkan `x` di ujung, `v.pop_back()` membuang elemen terakhir, `v.size()` memberikan banyaknya elemen, dan `v[i]` serta `v.at(i)` membaca dan menulisnya persis seperti pada array. Inilah wadah yang paling banyak dipakai di C++, dan cara ia tumbuh layak dipahami, karena sekilas sebuah array *tidak dapat* tumbuh.

Sebuah vector menyimpan tiga hal: alamat sebuah blok memori di *heap*, **size**-nya (berapa elemen yang terpakai), dan **capacity**-nya (berapa elemen yang muat dalam blok itu). Selama `size < capacity`, `push_back` cukup menulis elemen baru ke slot bebas berikutnya. Ketika blok penuh, `push_back` harus

1. meminta blok baru yang lebih besar kepada *heap*,
2. menyalin (atau memindahkan) setiap elemen yang ada ke dalamnya,
3. membebaskan blok lama, dan
4. baru kemudian menambahkan elemen baru.

Gambar 1.8 memperlihatkan satu kali tumbuh ulang seperti itu, dan **Kode 1.32** mengamati sebuah vector tumbuh dari kosong sampai 100 elemen, mencetak satu baris setiap kali kapasitasnya berubah.



Gambar 1.8 `std::vector` berkapasitas 8 yang sudah penuh menerima elemen kesembilan: blok berisi 16 dipesan, kedelapan elemen disalin, blok lama dibebaskan, dan 90 ditambahkan.

Kode 1.32 `vector_growth.cpp`: `std::array` berukuran tetap, dan `std::vector` yang menggandakan kapasitasnya ketika tumbuh.

```cpp
// vector_growth.cpp - std::array ukurannya tetap; std::vector tumbuh sendiri
#include <array>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    array<int, 5> fixedFive = {1, 2, 3, 4, 5};
    cout << "std::array<int, 5> holds " << fixedFive.size() << " ints, always\n\n";

    vector<int> v;
    size_t lastCapacity = v.capacity();
    cout << "empty vector: size 0, capacity " << lastCapacity << '\n';
    int moves = 0;                          // elemen yang disalin saat tumbuh ulang
    for (int i = 1; i <= 100; i++) {
        size_t sizeBefore = v.size();
        v.push_back(i);
        if (v.capacity() != lastCapacity) {
            moves += static_cast<int>(sizeBefore);
            cout << "push_back #" << i << ": capacity " << lastCapacity
                 << " -> " << v.capacity() << " (moved " << sizeBefore << ")\n";
            lastCapacity = v.capacity();
        }
    }
    cout << "after 100 push_backs: size " << v.size() << ", capacity "
         << v.capacity() << ", elements moved " << moves << '\n';
    return 0;
}
```

```
std::array<int, 5> holds 5 ints, always

empty vector: size 0, capacity 0
push_back #1: capacity 0 -> 1 (moved 0)
push_back #2: capacity 1 -> 2 (moved 1)
push_back #3: capacity 2 -> 4 (moved 2)
push_back #5: capacity 4 -> 8 (moved 4)
push_back #9: capacity 8 -> 16 (moved 8)
push_back #17: capacity 16 -> 32 (moved 16)
push_back #33: capacity 32 -> 64 (moved 32)
push_back #65: capacity 64 -> 128 (moved 64)
after 100 push_backs: size 100, capacity 128, elements moved 127
```

Dengan pustaka GCC, kapasitasnya *berlipat dua* setiap kali: 1, 2, 4, 8, dan seterusnya sampai 128. (Standar C++ tidak menetapkan faktor pertumbuhannya; pustaka Microsoft mengalikan dengan 1,5, tetapi penalaran di bawah ini berlaku sama untuk faktor tetap berapa pun.) Menyalin itu mahal, jadi mengapa pelipatan dua merupakan gagasan yang baik? Hitunglah salinannya. Untuk mencapai n elemen, vector tumbuh pada ukuran 1,2,4, ... sampai pangkat dua terbesar di bawah n, dan setiap kali tumbuh ia menyalin elemen sebanyak itu. Banyaknya salinan seluruhnya paling banyak

$$1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 < 2n \tag{1.5}$$

karena pangkat terbesar 2^k kurang dari n. Untuk $n = 100$ program menghitung $1 + 2 + 4 + 8 + 16 + 32 + 64 = 127$ salinan, memang kurang dari 200. Jadi meskipun sesekali satu `push_back` menyalin segalanya, biaya *rata-rata* per `push_back`, dalam jangka panjang, kurang dari dua salinan: sebuah konstanta. Subbab 1.6.5 memberi gagasan ini nama yang semestinya. Vector yang tumbuh satu slot demi satu slot justru akan menyalin $1 + 2 + \cdots + (n-1) = n(n-1)/2$ elemen, hampir 5.000 untuk $n = 100$, dan lima miliar untuk $n = 100{,}000$.

Jika Anda tahu sejak awal kira-kira berapa elemen yang akan ditampung sebuah vector, `v.reserve(n)` menetapkan kapasitasnya sekali dan menghindarkan setiap tumbuh ulang.

1.5.9 Array Java dan `ArrayList`

Array Java adalah objek di *heap*. `int[] a = new int[8];` membuat array berisi delapan elemen, semuanya bernilai `0`, dan variabel `a` menyimpan referensi kepadanya. Array itu mengetahui panjangnya, `a.length`, memeriksa setiap indeks, dan tidak dapat mengubah panjangnya sesudah dibuat. Elemen-elemennya disimpan bersinambung, sehingga Persamaan (1.2) berlaku persis bagi mereka: untuk `int[]` elemennya adalah bilangan itu sendiri, dan untuk array berisi objek elemennya adalah referensi [10].

Array Java yang dapat tumbuh adalah `ArrayList<T>`, dari `java.util`. Cara kerjanya persis seperti `std::vector`: sebuah array internal, sebuah *size*, sebuah *capacity*, dan tumbuh ulang ketika penuh. Operasi utamanya adalah `add(x)`, `get(i)`, `set(i, x)`, `remove(i)`, dan `size()`. `ArrayList` tidak membuka kapasitasnya, sehingga **Kode 1.33** memutar ulang aturan yang dipakai implementasi OpenJDK: `add` pertama memesan ruang untuk 10 elemen, dan list yang penuh tumbuh menjadi 1.5 kali kapasitas lamanya, dibulatkan ke bawah [11].

Kode 1.33 `ListGrowth.java`: array Java berpanjang tetap, dan langkah-langkah tumbuh sebuah `ArrayList` dari kosong sampai 100 elemen.

```
// ListGrowth.java - array Java panjangnya tetap; ArrayList tumbuh sendiri
import java.util.ArrayList;

public class ListGrowth {
    public static void main(String[] args) {
        int[] fixedFive = {1, 2, 3, 4, 5};
        System.out.println("int[] fixedFive holds " + fixedFive.length
                + " ints, always\n");

        // ArrayList menyembunyikan kapasitasnya; kita tiru aturan tumbuh OpenJDK:
        // add pertama memesan 10 slot; list penuh tumbuh jadi old + old / 2.
        ArrayList<Integer> list = new ArrayList<>();
        int capacity = 0;
        int moves = 0;
        System.out.println("empty list: size 0, capacity 0");
        for (int i = 1; i <= 100; i++) {
            int sizeBefore = list.size();
            if (sizeBefore == capacity) {
                int newCapacity = (capacity == 0) ? 10 : capacity + capacity / 2;
                moves += sizeBefore;
                System.out.println("add #" + i + ": capacity " + capacity + " -> "
                        + newCapacity + " (moved " + sizeBefore + ")");
                capacity = newCapacity;
            }
            list.add(i);
        }
        System.out.println("after 100 adds: size " + list.size() + ", capacity "
                + capacity + ", elements moved " + moves);
    }
}
```

```
int[] fixedFive holds 5 ints, always

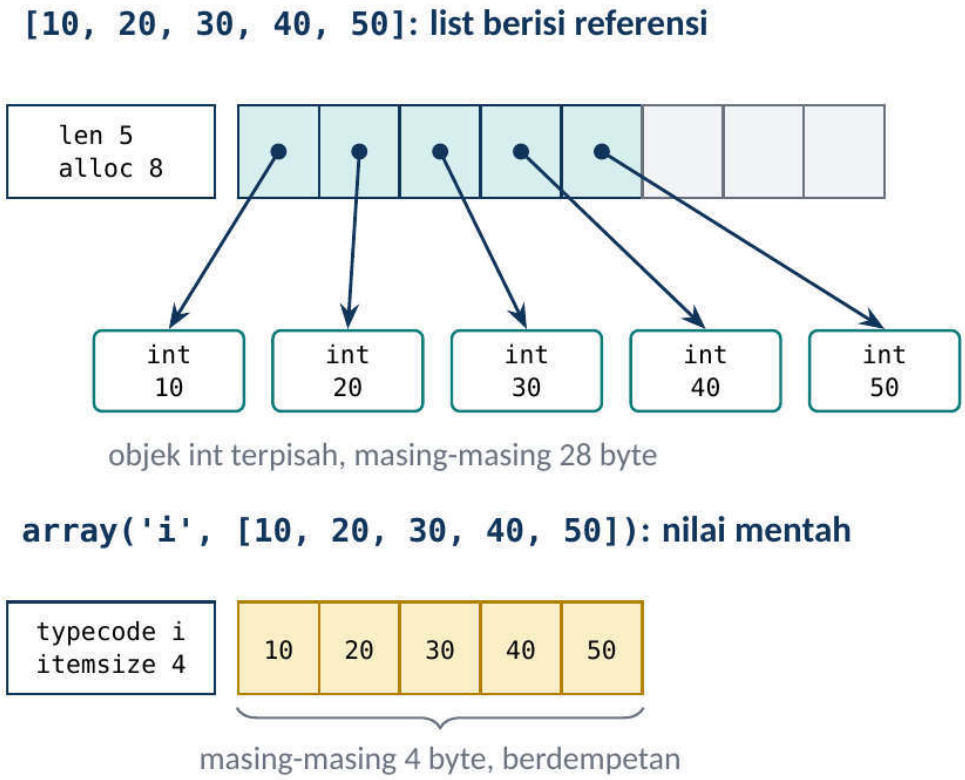
empty list: size 0, capacity 0
add #1: capacity 0 -> 10 (moved 0)
add #11: capacity 10 -> 15 (moved 10)
add #16: capacity 15 -> 22 (moved 15)
add #23: capacity 22 -> 33 (moved 22)
add #34: capacity 33 -> 49 (moved 33)
add #50: capacity 49 -> 73 (moved 49)
add #74: capacity 73 -> 109 (moved 73)
after 100 adds: size 100, capacity 109, elements moved 202
```

Tumbuh dengan faktor 1,5 alih-alih 2 lebih sering memicu tumbuh ulang (di sini 7 kali alih-alih 8 untuk 100 elemen, tetapi dimulai dari 10) dan memboroskan lebih sedikit ruang di bagian blok yang belum terpakai. Argumen yang sama dengan Persamaan (1.5), dengan rasio 1.5 alih-alih 2, memperlihatkan bahwa banyaknya salinan seluruhnya tetap di bawah $3n$: sekali lagi sebuah konstanta per `add`, secara rata-rata. Program itu menghitung 202 salinan untuk 100 elemen.

`ArrayList` hanya dapat menyimpan objek, bukan tipe dasar, sehingga list bilangan harus dideklarasikan `ArrayList<Integer>`, dan setiap `int` yang dimasukkan otomatis dibungkus dalam objek `Integer`. Pembungkusan itu, yang disebut **boxing**, memakan memori dan waktu, dan itulah sebabnya pekerjaan numerik di Java lebih menyukai `int[]` biasa.

1.5.10 List Python dan modul `array`

`list` bawaan Python adalah array yang dapat tumbuh berisi *referensi*, seperti `ArrayList`. List berisi bilangan 10 sampai 50 sebenarnya adalah deretan bersinambung referensi berukuran delapan byte, masing-masing menunjuk ke objek bilangan bulat terpisah di *heap*. **Gambar 1.9** menggambarkannya, bersama alternatif yang disediakan modul standar `array`.



Gambar 1.9 `list` Python menyimpan referensi ke objek-objek terpisah; `array('i')` dari modul `array` menyimpan nilai mentah 4 byte itu sendiri, berdempetan.

List juga memesan ruang lebih, dengan menyisakan slot cadangan di ujungnya sehingga kebanyakan `append` tidak perlu menyalin. **Kode 1.34** mengamati `sys.getsizeof` sebuah list selagi 100 elemen ditambahkan.

Kode 1.34 `list_growth.py`: tuple tidak dapat tumbuh; list tumbuh bertahap, menyisakan ruang untuk `append` berikutnya.

```python
# list_growth.py - list memesan lebih, jadi kebanyakan append tanpa menyalin
import sys

fixed_five = (1, 2, 3, 4, 5)             # tuple sama sekali tidak dapat tumbuh
print(f"the tuple fixed_five holds {len(fixed_five)} ints, always\n")

lst = []
last = sys.getsizeof(lst)
print(f"empty list: size 0, {last} bytes")
for i in range(1, 101):
    lst.append(i)
    now = sys.getsizeof(lst)
    if now != last:
        slots = (now - sys.getsizeof([])) // 8   # 8 byte per referensi
        print(f"append #{i}: {last} -> {now} bytes (room for {slots} references)")
        last = now
print(f"after 100 appends: size {len(lst)}, {sys.getsizeof(lst)} bytes")
```

```
the tuple fixed_five holds 5 ints, always

empty list: size 0, 56 bytes
append #1: 56 -> 88 bytes (room for 4 references)
append #5: 88 -> 120 bytes (room for 8 references)
append #9: 120 -> 184 bytes (room for 16 references)
append #17: 184 -> 248 bytes (room for 24 references)
append #25: 248 -> 312 bytes (room for 32 references)
append #33: 312 -> 376 bytes (room for 40 references)
append #41: 376 -> 472 bytes (room for 52 references)
append #53: 472 -> 568 bytes (room for 64 references)
append #65: 568 -> 664 bytes (room for 76 references)
append #77: 664 -> 792 bytes (room for 92 references)
append #93: 792 -> 920 bytes (room for 108 references)
after 100 appends: size 100, 920 bytes
```

CPython, interpreter Python yang baku, menumbuhkan list yang penuh sebesar kira-kira seperdelapan ukurannya ditambah beberapa slot, dibulatkan ke kelipatan empat, sehingga langkah-langkah di atas berjalan 4, 8, 16, 24, 32, 40, 52, 64, 76, 92, 108 [12]. Angka persisnya adalah rincian implementasi dan dapat berbeda antarversi Python; yang penting, setiap langkah lebih besar sepecahan tetap daripada langkah sebelumnya, yang sekali lagi menjaga biaya rata-rata sebuah `append` tetap konstan. **Tuple**, yang ditulis dengan kurung biasa, adalah deret Python yang berukuran tetap dan tidak dapat diubah.

Ketika program menangani banyak bilangan sejenis, list berisi referensi itu boros: setiap elemen menghabiskan referensi delapan byte ditambah objek bilangan bulat 28 byte. **Modul `array`** menyimpan bilangan dengan cara C++, sebagai nilai mentah bertipe tetap yang dipadatkan berdampingan [13]. `array('i', ...)` menyimpan `int` C berukuran empat byte; `'d'` menyimpan `double` delapan byte. **Kode 1.35** membandingkan keduanya untuk 1.000 bilangan.

Kode 1.35 `compact_array.py`: 1.000 bilangan dalam list berisi referensi dan dalam `array('i')` yang padat.

```python
# compact_array.py - list berisi referensi; array('i') berisi int 4 byte
import sys
from array import array

N = 1000
as_list = list(range(N))
as_array = array("i", range(N))

print(f"bytes per element      : {as_array.itemsize}")
print(f"array('i') element data: {as_array.itemsize * len(as_array)} bytes")
print(f"array object in total  : {sys.getsizeof(as_array)} bytes")
print(f"list object in total   : {sys.getsizeof(as_list)} bytes (references only)")
print(f"one int object         : {sys.getsizeof(999)} bytes,"
      f" and the list refers to {N} of them")
print(f"as_list[999] + as_array[999]: {as_list[999] + as_array[999]}")
```

```
bytes per element      : 4
array('i') element data: 4000 bytes
array object in total  : 4200 bytes
list object in total   : 8056 bytes (references only)
one int object         : 28 bytes, and the list refers to 1000 of them
as_list[999] + as_array[999]: 1998
```

List itu sendiri menempati 8.056 byte, delapan per referensi ditambah kepala kecil, sebelum menghitung objek-objek bilangan bulat yang dirujuknya, yang membawa totalnya menjadi sekitar 36.000 byte; `array` menempati 4.200 byte seluruhnya, 4.000 di antaranya adalah bilangan-bilangan itu sendiri. Untuk seribu bilangan perbedaan itu tidak berarti. Untuk seratus juta, itulah perbedaan antara 400 megabyte dan 3,6 gigabyte, dan itulah sebabnya Python ilmiah memakai array padat, yang paling terkenal adalah array pustaka NumPy, yang dibangun di atas gagasan yang sama.

Kode 1.36 dan **Kode 1.37** adalah padanannya di C++ dan Java. Di C++, array mentah maupun blok milik vector sama-sama menyimpan bilangan secara langsung, 4.000 byte untuk 1.000 `int32_t`, dan objek vector itu sendiri hanya 24 byte: ketiga field dalam **Gambar 1.8**. Di Java, `int[]` menyimpan bilangan secara langsung, sedangkan `Integer[]` menyimpan referensi ke objek-objek terpisah, persis seperti list Python.

Kode 1.36 `compact_array.cpp`: 1.000 `int32_t` dalam array mentah dan dalam vector.

```cpp
// compact_array.cpp - 1.000 int disimpan berdempetan, masing-masing 4 byte
#include <cstdint>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    const int N = 1000;
    int32_t plain[N];
    vector<int32_t> v(N);
    for (int i = 0; i < N; i++) {
        plain[i] = i;
        v[i] = i;
    }
    cout << "bytes per element      : " << sizeof(int32_t) << '\n';
    cout << "plain array, N elements: " << sizeof plain << " bytes\n";
    cout << "vector's element block : " << v.size() * sizeof(int32_t) << " bytes\n";
    cout << "vector object itself   : " << sizeof v
         << " bytes (pointers to the block)\n";
    cout << "plain[999] + v[999]    : " << plain[999] + v[999] << '\n';
    return 0;
}
```

```
bytes per element      : 4
plain array, N elements: 4000 bytes
vector's element block : 4000 bytes
vector object itself   : 24 bytes (pointers to the block)
plain[999] + v[999]    : 1998
```

Kode 1.37 `CompactArray.java`: `int[]` menyimpan nilai, `Integer[]` menyimpan referensi.

```java
// CompactArray.java - int[] menyimpan nilai; Integer[] menyimpan referensi ke objek
public class CompactArray {
    public static void main(String[] args) {
        final int N = 1000;
        int[] plain = new int[N];
        Integer[] boxed = new Integer[N];
        for (int i = 0; i < N; i++) {
            plain[i] = i;
            boxed[i] = i;                    // setiap nilai menjadi Integer
        }
        System.out.println("bytes per element      : " + Integer.BYTES);
        System.out.println("int[] element data     : " + N * Integer.BYTES
                + " bytes");
        System.out.println("Integer[] holds        : " + N + " references + up to "
                + N + " separate objects");
        System.out.println("plain[999] + boxed[999]: " + (plain[999] + boxed[999]));
    }
}
```

```
bytes per element      : 4
int[] element data     : 4000 bytes
Integer[] holds        : 1000 references + up to 1000 separate objects
plain[999] + boxed[999]: 1998
```

1.5.11 Ketiga bahasa berdampingan

Tabel 1.5 menghimpun perbedaan-perbedaan yang dijumpai subbab ini.

Tabel 1.5 Array dan array yang dapat tumbuh di C++, Java, dan Python.

Pertanyaan	C++	Java	Python
Array berukuran tetap	array mentah `int a[8]`; `std::array<int, 8>`	`int[] a = new int[8]`	`tuple`; `array('i', ...)` bertipe butir tetap
Array yang dapat tumbuh	`std::vector<int>`	`ArrayList<Integer>`	`list`
Letak elemen	di blok milik array itu sendiri	tipe dasar: di blok; objek: referensi di blok	referensi di blok; `array`: nilai mentah
Mengetahui panjangnya	`std::array`, `vector`: `size()`; array mentah: tidak	`a.length`, `list.size()`	`len(a)`
Indeks di luar jangkauan	perilaku tak terdefinisi; `at()` melempar `out_of_range`	`ArrayIndexOutOfBoundsException`	`IndexError`
Indeks negatif	perilaku tak terdefinisi	eksepsi	dihitung dari ujung
Dua dimensi	bersinambung, row-major	array berisi array baris, boleh bergerigi	list berisi list baris
Pembebasan memori	otomatis untuk array dan wadah; `delete` untuk `new`	*garbage collector*	*garbage collector*
Faktor tumbuh ketika penuh	2 (GCC), 1,5 (Microsoft)	1,5 (OpenJDK)	sekitar 1,125 (CPython)

1.6 Cicipan Pertama Kompleksitas: Akses $O(1)$ lawan Penyisipan $O(n)$

1.6.1 Menghitung langkah, bukan detik

Seberapa cepat sebuah operasi pada array? Kita dapat mengukurnya dengan stopwatch, tetapi jawabannya akan bergantung pada komputernya, kompilatornya, apa lagi yang sedang dikerjakan mesin, dan fase bulan. Karena itu ilmuwan komputer mengajukan pertanyaan lain, yang jawabannya sama pada setiap mesin: **bagaimana banyaknya langkah dasar tumbuh ketika banyaknya data tumbuh?** Langkah dasar adalah apa saja yang memakan waktu kecil dan tetap, tidak bergantung pada data: membaca atau menulis satu elemen, membandingkan dua bilangan, menjumlahkan dua bilangan. Kita menulis n untuk banyaknya elemen dan menghitung langkah sebagai fungsi dari n [1].

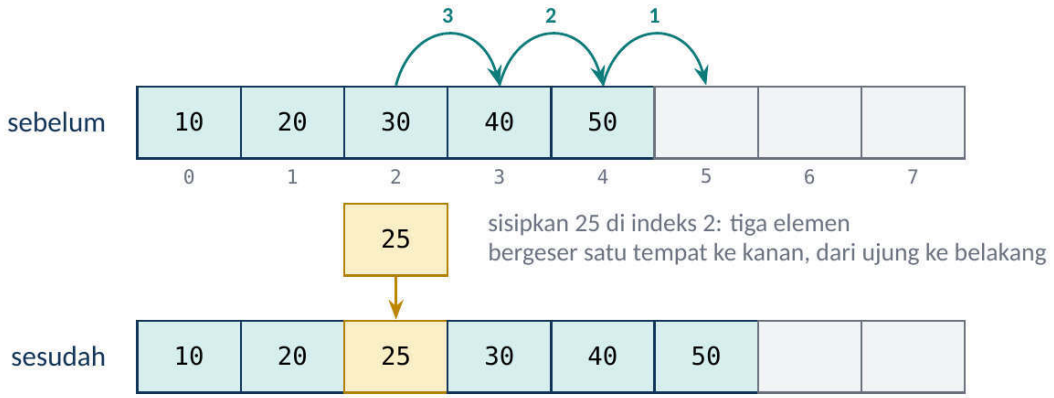
1.6.2 Akses: biaya yang sama untuk setiap indeks

Membaca `A[i]`, menurut Persamaan (1.2), adalah satu perkalian, satu penjumlahan, dan satu pembacaan memori. Itu segelintir langkah yang sama, entah array itu berisi delapan elemen atau delapan juta, dan entah i bernilai 0 atau $n - 1$. Biayanya sama sekali tidak tumbuh bersama n. Kita katakan bahwa akses memakan **waktu konstan**, dan menuliskannya $O(1)$, dibaca "orde satu" atau "big-O dari satu". Menulis sebuah elemen, `A[i] = x`, juga $O(1)$ dengan alasan yang sama.

Inilah kekuatan besar array, dan hampir setiap struktur lain dalam buku ini melepaskannya demi mendapat sesuatu yang lain. *Linked list*, misalnya, harus berjalan dari elemen pertamanya melewati i elemen lain untuk mencapai elemen i.

1.6.3 Penyisipan dan penghapusan: biaya menjaga deretan tetap rapat

Kekuatan array sekaligus kelemahannya. Karena elemen-elemennya harus tetap berdampingan, tanpa celah, menaruh elemen baru *di antara* dua elemen yang ada berarti membuat ruang lebih dahulu: setiap elemen dari titik penyisipan sampai ujung harus bergeser satu tempat ke kanan. **Gambar 1.10** memperlihatkan 25 disisipkan di indeks 2 pada array `10 20 30 40 50`.



Gambar 1.10 Menyisipkan 25 di indeks 2: elemen-elemen di indeks 4, 3, dan 2 bergeser satu tempat ke kanan, dalam urutan itu, lalu 25 ditulis ke slot yang dikosongkan.

Pergeseran harus berjalan dari ujung ke belakang: 50 lebih dahulu ke slot kosong 5, lalu 40 ke slot 4, lalu 30 ke slot 3. Menggeser dari depan akan menimpa 40 dengan 30 sebelum 40 sempat diselamatkan. **Pseudokode 1.1** merumuskan prosedurnya dengan tepat, dalam gaya pseudokode yang dipakai di sepanjang buku ini: baris-baris bernomor, `←` untuk penugasan, dan array yang ditulis $A[0..n-1]$ untuk elemen-elemen berindeks 0 sampai $n-1$.

Pseudokode 1.1 `ARRAY-INSERT`: buka celah di indeks `pos` dengan menggeser elemen-elemen sesudahnya satu tempat ke kanan, lalu tulis `x` ke celah itu.

```
ARRAY-INSERT(A, n, pos, x):          // A[0..n-1] berisi n elemen; 0 ≤ pos ≤ n
 1.  for i ← n downto pos + 1         // berjalan dari ujung mundur ke pos
 2.      A[i] ← A[i - 1]              // geser satu elemen satu tempat ke kanan
 3.  A[pos] ← x                       // tulis elemen baru ke celah
 4.  n ← n + 1
```

Penghapusan adalah bayangan cerminnya, yang diberikan dalam **Pseudokode 1.2**: setiap elemen sesudah elemen yang dibuang bergeser satu tempat ke *kiri*, dari depan ke belakang, untuk menutup celah.

Pseudokode 1.2 `ARRAY-DELETE`: tutup celah yang ditinggalkan elemen di indeks `pos` dengan menggeser elemen-elemen sesudahnya satu tempat ke kiri.

```
ARRAY-DELETE(A, n, pos):             // A[0..n-1] berisi n elemen; 0 ≤ pos ≤ n - 1
 1.  for i ← pos to n - 2
 2.      A[i] ← A[i + 1]              // geser satu elemen satu tempat ke kiri
 3.  n ← n - 1
```

Berapa kali `ARRAY-INSERT` menggeser? Baris 2 berjalan sekali untuk setiap i dari n turun sampai $pos+1$, yaitu untuk

$$M_{\text{insert}}(n, pos) = n - pos \tag{1.6}$$

nilai i. Menyisipkan di ujung, $pos = n$, sama sekali tidak menggeser. Menyisipkan di depan, $pos = 0$, menggeser setiap satu dari n elemen. Dan jika titik penyisipan sama mungkinnya berada di salah satu dari $n+1$ posisi $0,1,\ldots,n$, banyaknya geseran *rata-rata* adalah

$$\frac{1}{n+1}\sum_{pos=0}^{n}(n-pos)=\frac{1}{n+1}\cdot\frac{n(n+1)}{2}=\frac{n}{2} \tag{1.7}$$

Langkah tengahnya memakai jumlah $0+1+2+\cdots+n=\frac{n(n+1)}{2}$. Untuk melihat mengapa jumlah itu benar, tulislah dua kali, sekali maju dan sekali mundur, lalu jumlahkan kedua baris itu kolom demi kolom: setiap kolom berjumlah n, dan ada $n+1$ kolom, sehingga dua kali jumlah itu adalah $n(n+1)$. Persamaan (1.7) mengatakan bahwa penyisipan ke dalam array berisi sejuta elemen menggeser, secara rata-rata, setengah juta elemen. Penghapusan mempunyai hitungan yang persis sama, $n-1-pos$ geseran, dengan rata-rata $(n-1)/2$.

Kode 1.38 mengimplementasikan kedua prosedur, menerapkannya pada array dalam **Gambar 1.10**, lalu menghitung geseran yang diperlukan untuk menyisipkan satu elemen di depan, di tengah, dan di ujung array berisi 10 sampai 10.000 elemen.

Kode 1.38 `insert_shift.cpp`: penyisipan dan penghapusan dengan menggeser, beserta hitungan geserannya.

```
// insert_shift.cpp - akses O(1) lawan penyisipan O(n), dihitung
#include <iomanip>
#include <iostream>
#include <vector>
using namespace std;

// Menyisipkan x di indeks pos pada a[0..n-1] (bila kapasitas cukup).
// Mengembalikan banyak elemen yang digeser satu tempat ke kanan.
int insertAt(vector<int>& a, int& n, int pos, int x) {
    int moves = 0;
    for (int i = n; i > pos; i--) {       // berjalan dari ujung mundur ke pos
        a[i] = a[i - 1];
        moves++;
    }
    a[pos] = x;
    n++;
    return moves;
}

// Hapus elemen di indeks pos; kembalikan banyak elemen yang bergeser.
int eraseAt(vector<int>& a, int& n, int pos) {
    int moves = 0;
    for (int i = pos; i < n - 1; i++) {
        a[i] = a[i + 1];
        moves++;
    }
    n--;
    return moves;
}

void show(const vector<int>& a, int n) {
    cout << '[';
    for (int i = 0; i < n; i++) cout << (i ? " " : "") << a[i];
    cout << "]";
}
```

Kode 1.38 (Lanjutan) *insert_shift.cpp: penyisipan dan penghapusan dengan menggeser, beserta hitungan geserannya.*

```
int main() {
    vector<int> a(8);
    int n = 0;
    for (int x : {10, 20, 30, 40, 50}) a[n++] = x;
    cout << "start            "; show(a, n); cout << '\n';
    cout << "a[3] is read in one step: " << a[3] << '\n';
    int m = insertAt(a, n, 2, 25);
    cout << "insert 25 at 2   "; show(a, n); cout << "  moved " << m << '\n';
    m = insertAt(a, n, 0, 5);
    cout << "insert 5 at 0    "; show(a, n); cout << "  moved " << m << '\n';
    m = insertAt(a, n, n, 60);
    cout << "insert 60 at end "; show(a, n); cout << "  moved " << m << '\n';
    m = eraseAt(a, n, 1);
    cout << "erase index 1    "; show(a, n); cout << "  moved " << m << '\n';

    cout << "\nmoves needed to insert one element into an array of n:\n";
    cout << setw(6) << "n" << setw(8) << "front" << setw(8) << "middle"
         << setw(6) << "end" << '\n';
    for (int size : {10, 100, 1000, 10000}) {
        int results[3];
        int positions[3] = {0, size / 2, size};
        for (int k = 0; k < 3; k++) {
            vector<int> b(size + 1);
            int len = size;
            results[k] = insertAt(b, len, positions[k], -1);
        }
        cout << setw(6) << size << setw(8) << results[0] << setw(8) << results[1]
             << setw(6) << results[2] << '\n';
    }
    return 0;
}
```

```
start            [10 20 30 40 50]
a[3] is read in one step: 40
insert 25 at 2   [10 20 25 30 40 50]  moved 3
insert 5 at 0    [5 10 20 25 30 40 50]  moved 6
insert 60 at end [5 10 20 25 30 40 50 60]  moved 0
erase index 1    [5 20 25 30 40 50 60]  moved 6

moves needed to insert one element into an array of n:
     n   front  middle   end
    10      10       5     0
   100     100      50     0
  1000    1000     500     0
 10000   10000    5000     0
```

Tabel itu adalah Persamaan (1.6) yang sedang bekerja. Membaca `a[3]` memakan satu langkah, berapa pun ukurannya. Menyisipkan di depan memakan tepat n geseran dan di tengah tepat $n/2$: kalikan ukurannya dengan sepuluh, dan pekerjaannya pun dikalikan sepuluh. Hanya penyisipan tepat di ujung yang gratis, dan justru itulah sebabnya `push_back`, `add`, dan `append` adalah operasi yang dibuat murah oleh array yang dapat tumbuh.

Kode 1.39 dan **Kode 1.40** mencetak keluaran yang persis sama. Perhatikan dalam versi Python bahwa perulangan `for i in range(n, pos, -1)` menghitung *mundur* dari n sampai $pos + 1$: argumen ketiga `range` adalah langkahnya, dan argumen kedua adalah nilai pertama yang *tidak* dicapai, persis seperti batas pada baris 1 **Pseudokode 1.1**.

Kode 1.39 `InsertShift.java`: penyisipan dan penghapusan dengan menggeser, di Java.

```java
// InsertShift.java - akses O(1) lawan penyisipan O(n), dihitung
public class InsertShift {
    static int n = 0;                          // banyak slot yang terpakai

    // Sisipkan x di indeks pos; kembalikan banyak elemen yang bergeser.
    static int insertAt(int[] a, int pos, int x) {
        int moves = 0;
        for (int i = n; i > pos; i--) {        // berjalan dari ujung mundur ke pos
            a[i] = a[i - 1];
            moves++;
        }
        a[pos] = x;
        n++;
        return moves;
    }

    // Hapus elemen di indeks pos; kembalikan banyak elemen yang bergeser.
    static int eraseAt(int[] a, int pos) {
        int moves = 0;
        for (int i = pos; i < n - 1; i++) {
            a[i] = a[i + 1];
            moves++;
        }
        n--;
        return moves;
    }

    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < n; i++) sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    public static void main(String[] args) {
        int[] a = new int[8];
        for (int x : new int[] {10, 20, 30, 40, 50}) a[n++] = x;
        System.out.println("start            " + show(a));
        System.out.println("a[3] is read in one step: " + a[3]);
        int m = insertAt(a, 2, 25);
        System.out.println("insert 25 at 2   " + show(a) + "  moved " + m);
        m = insertAt(a, 0, 5);
        System.out.println("insert 5 at 0    " + show(a) + "  moved " + m);
        m = insertAt(a, n, 60);
        System.out.println("insert 60 at end " + show(a) + "  moved " + m);
        m = eraseAt(a, 1);
        System.out.println("erase index 1    " + show(a) + "  moved " + m);
```

Kode 1.39 (Lanjutan) *InsertShift.java: penyisipan dan penghapusan dengan menggeser, di Java.*

```java

        System.out.println(
                "\nmoves needed to insert one element into an array of n:");
        System.out.printf("%6s%8s%8s%6s%n", "n", "front", "middle", "end");
        for (int size : new int[] {10, 100, 1000, 10000}) {
            int[] positions = {0, size / 2, size};
            int[] results = new int[3];
            for (int k = 0; k < 3; k++) {
                int[] b = new int[size + 1];
                n = size;
                results[k] = insertAt(b, positions[k], -1);
            }
            System.out.printf("%6d%8d%8d%6d%n",
                    size, results[0], results[1], results[2]);
        }
    }
}
```

Kode 1.40 `insert_shift.py`: penyisipan dan penghapusan dengan menggeser, di Python.

```
# insert_shift.py - akses O(1) lawan penyisipan O(n), dihitung
def insert_at(a, n, pos, x):
    """Sisipkan x di indeks pos pada a[0..n-1]; kembalikan banyak geseran."""
    moves = 0
    for i in range(n, pos, -1):          # berjalan dari ujung mundur ke pos
        a[i] = a[i - 1]
        moves += 1
    a[pos] = x
    return moves


def erase_at(a, n, pos):
    """Hapus elemen di indeks pos; kembalikan banyak geseran ke kiri."""
    moves = 0
    for i in range(pos, n - 1):
        a[i] = a[i + 1]
        moves += 1
    return moves


def show(a, n):
    return "[" + " ".join(str(a[i]) for i in range(n)) + "]"


a = [0] * 8
n = 0
for x in (10, 20, 30, 40, 50):
    a[n] = x
    n += 1
print(f"start            {show(a, n)}")
print(f"a[3] is read in one step: {a[3]}")
m = insert_at(a, n, 2, 25)
n += 1
print(f"insert 25 at 2   {show(a, n)}  moved {m}")
m = insert_at(a, n, 0, 5)
n += 1
print(f"insert 5 at 0    {show(a, n)}  moved {m}")
m = insert_at(a, n, n, 60)
n += 1
print(f"insert 60 at end {show(a, n)}  moved {m}")
m = erase_at(a, n, 1)
n -= 1
print(f"erase index 1    {show(a, n)}  moved {m}")
```

Kode 1.40 (Lanjutan) *insert_shift.py: penyisipan dan penghapusan dengan menggeser, di Python.*

```
print("\nmoves needed to insert one element into an array of n:")
print(f"{'n':>6}{'front':>8}{'middle':>8}{'end':>6}")
for size in (10, 100, 1000, 10000):
    results = []
    for pos in (0, size // 2, size):
        b = [0] * (size + 1)
        results.append(insert_at(b, size, pos, -1))
    print(f"{size:>6}{results[0]:>8}{results[1]:>8}{results[2]:>6}")
```

Wadah-wadah standar mengerjakan pekerjaan yang persis sama di balik antarmuka yang lebih ramah: `v.insert(v.begin() + pos, x)` di C++, `list.add(pos, x)` di Java, dan `lst.insert(pos, x)` di Python semuanya menggeser elemen-elemen sesudah `pos`. Satu baris kode dapat menyembunyikan sejuta geseran, dan pemrogram yang menyisipkan di depan sebuah list besar di dalam perulangan dapat menulis program yang benar tetapi luar biasa lambat. Mengetahui berapa biaya baris itu adalah inti seluruh mata pelajaran ini.

1.6.4 Big-O, secara informal

Kita memerlukan cara singkat untuk mengatakan "banyaknya langkah tumbuh sebanding dengan n" tanpa meributkan apakah persisnya n, atau $n/2$, atau $n/2 + 3$. Cara singkat itu adalah **notasi big-O**. Bab 2 mendefinisikannya dengan kecermatan penuh; di sini cukup untuk dapat memakainya.

> **DEFINISI.** Misalkan $f(n)$ dan $g(n)$ adalah banyaknya langkah dua komputasi pada masukan berukuran n. Kita menulis $f(n) = O(g(n))$ jika ada konstanta positif c dan n_0 sehingga Persamaan (1.8) berlaku. Dengan kata lain: mulai dari suatu ukuran, f tumbuh tidak lebih cepat daripada suatu kelipatan konstan dari g [1].

$$f(n) \le c \cdot g(n) \quad \text{untuk semua } n \ge n_0 \tag{1.8}$$

Persamaan (1.8) sengaja mengabaikan dua hal. Ia mengabaikan **faktor konstan**: $n/2$ geseran dan n geseran sama-sama $O(n)$, karena $n/2 \le 1 \cdot n$. Dan ia mengabaikan **masukan kecil**: hanya perilaku mulai dari n_0 yang dihitung. Kedua pengabaian itu persis yang kita inginkan. Faktor konstan bergantung pada mesin dan kompilator, yang sudah kita sepakati untuk dikesampingkan, dan untuk masukan kecil setiap cara terasa cepat. Yang tersisa adalah *bentuk* pertumbuhannya, dan bentuk itulah yang menentukan segalanya ketika n besar, seperti diperlihatkan **Tabel 1.1**.

Dua bentuk muncul dalam bab ini:

- $O(1)$, **konstan**: langkahnya sama sekali tidak tumbuh bersama n. Akses array, serta penyisipan atau penghapusan di ujung.
- $O(n)$, **linear**: langkahnya tumbuh sebanding dengan n. Gandakan n dan pekerjaannya berlipat dua. Penelusuran, pencarian linear, serta penyisipan atau penghapusan di depan atau di tengah.

Sebagai contoh definisi itu bekerja, ambillah biaya penyisipan rata-rata $f(n) = n/2$ dari Persamaan (1.7) dan $g(n) = n$. Dengan memilih $c = 1$ dan $n_0 = 1$, kita peroleh $n/2 \le 1 \cdot n$ untuk setiap $n \ge 1$, sehingga $n/2 = O(n)$. Untuk penelusuran dalam `sumOf`, badan perulangan berjalan n kali dan setiap kali melakukan satu perbandingan, satu penjumlahan, dan satu penambahan indeks, katakanlah $3n + 2$ langkah bila permulaan dan pengembalian ikut dihitung; dengan $c = 5$ dan $n_0 = 1$, $3n + 2 \le 5n$ untuk semua $n \ge 1$, sehingga penjumlahan itu juga $O(n)$.

Tabel 1.6 meringkas biaya operasi-operasi pada array berisi n elemen. Inilah yang pertama dari banyak tabel serupa dalam buku ini; setiap struktur data, pada akhirnya, adalah pilihan tentang baris mana dalam tabel ini yang dibuat murah.

Tabel 1.6 Biaya operasi-operasi dasar pada array berisi n elemen.

Operasi	Langkah	Big-O	Alasan
Membaca atau menulis `A[i]`	1	$O(1)$	Persamaan (1.2): satu kali menghitung alamat
Menyisipkan atau menghapus di ujung (ruang tersedia)	1	$O(1)$	tidak ada yang perlu bergeser
Menyisipkan di indeks `pos`	$n - pos$	$O(n)$	Persamaan (1.6); rata-rata $n/2$
Menghapus di indeks `pos`	$n - 1 - pos$	$O(n)$	cermin penyisipan
Menelusuri, menjumlah, maksimum, mencacah, membalik	sekitar n	$O(n)$	setiap elemen dikunjungi sekali
Mencari sebuah nilai (tak terurut)	sampai n	$O(n)$	mungkin harus melihat setiap elemen
`push_back` / `add` / `append`, dirata-rata atas banyak operasi	kurang dari 2 atau 3 salinan masing-masing	$O(1)$ teramortisasi	Persamaan (1.5)

1.6.5 Sekilas tentang biaya teramortisasi

Baris terakhir **Tabel 1.6** layak mendapat satu paragraf lagi, karena di dalamnya ada gagasan yang akan kita jumpai lagi. Satu `push_back` pada vector yang penuh berbiaya $O(n)$: ia menyalin kesemua n elemen. Namun Persamaan (1.5) memperlihatkan bahwa n kali `push_back` yang dimulai dari kosong berbiaya kurang dari $2n$ salinan *seluruhnya*. Bila total itu dibagi rata, setiap `push_back` berbiaya kurang dari dua salinan ditambah operasi tulisnya sendiri: $O(1)$ per operasi *secara rata-rata atas sembarang barisan operasi*. Ini disebut waktu konstan **teramortisasi** (*amortized*), dari istilah akuntan untuk menyebarkan biaya besar ke banyak cicilan. Ini tidak sama dengan "biasanya cepat": ini jaminan atas setiap barisan panjang operasi, sesial apa pun. Bab 5 memakai penalaran yang sama untuk *stack* dan *queue* yang dibangun di atas array yang dapat tumbuh.

1.7 Menyiapkan VS Code untuk C++, Java, dan Python di Windows, macOS, dan Linux

Semua yang ada dalam buku ini berjalan di laptop biasa, dan tidak ada yang perlu dibeli. Subbab ini memasang, sekali saja dan dengan cermat, meja kerja yang lengkap: sebuah editor, kompilator dan *debugger* C++, Java Development Kit, dan Python, lalu membuktikan bahwa masing-masing bekerja dengan menjalankan program pertama yang sama dalam ketiga bahasa. Bab-bab berikutnya mengandaikan bahwa langkah ini sudah dikerjakan.

1.7.1 Tiga kata lebih dahulu

Terminal adalah jendela tempat Anda mengetik perintah dan membaca jawabannya. Di Windows namanya **PowerShell** atau **Command Prompt** yang lebih tua; di macOS dan Linux namanya **Terminal**; dan VS Code mempunyai terminal bawaan, yang dibuka dengan **Terminal → New Terminal**. Dalam buku ini, baris yang diketikkan ke terminal dicetak dengan huruf bertelebar tetap, dan Anda menekan Enter di akhirnya.

Manajer paket (*package manager*) memasang perangkat lunak lewat perintah alih-alih lewat klik demi klik pada sebuah pemasang: `winget` di Windows 10 dan 11, **Homebrew** (perintah `brew`, dari `https://brew.sh`) di macOS, dan `apt` di Ubuntu dan Debian. Setiap langkah di bawah memberikan perintah manajer paketnya dan, bila ada, alternatif grafisnya; yang mana pun boleh.

PATH adalah daftar folder tempat terminal mencari program. Ketika Anda mengetik `g++`, terminal mencari program bernama `g++` di setiap folder dalam PATH. Pesan "command not found" atau "is not recognized" hampir selalu berarti programnya sudah terpasang tetapi foldernya belum ada di PATH, atau terminalnya dibuka sebelum folder itu ditambahkan. Membuka terminal **baru** adalah obat pertama yang layak dicoba.

1.7.2 Visual Studio Code

Visual Studio Code, singkatnya VS Code, adalah editor gratis dari Microsoft yang tampil dan bekerja sama di ketiga sistem, dan menjadi lingkungan lengkap untuk bahasa apa pun melalui **ekstensi** [14]. Rilisnya pada September 2026 adalah versi 1.138; ia memperbarui dirinya setiap bulan, sehingga versi baru mana pun sudah cukup [15]. **Tabel 1.7** memberikan cara pemasangannya.

Tabel 1.7 Memasang VS Code di setiap sistem operasi.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Pasang	Pemasang dari `https://code.visualstudio.com`; centang **Add to PATH**. Atau `winget install Microsoft.VisualStudioCode`	Unduh, ekstrak, lalu seret *Visual Studio Code* ke **Applications**. Atau `brew install --cask visual-studio-code`	Berkas `.deb` dari situsnya, lalu `sudo apt install ./code_*.deb`. Atau `sudo snap install code --classic`
Agar `code` berfungsi di terminal	Dikerjakan pemasangnya	Di VS Code: **Cmd+Shift+P**, lalu *Shell Command: Install 'code' command in PATH*	Dikerjakan paketnya
Periksa	`code --version`	`code --version`	`code --version`

Buatlah satu folder untuk buku ini, misalnya `DS` di folder Documents Anda, dengan satu subfolder per bab, `Chapter01`, `Chapter02`, dan seterusnya, sesuai susunan kode pendamping. Selalu buka **folder**-nya di VS Code (**File → Open Folder…**), jangan satu berkas saja: terminal VS Code lalu terbuka di dalam folder itu, sehingga perintah seperti `./io_sum < week_loans.txt` menemukan berkas-berkasnya.

1.7.3 Kompilator dan debugger C++

C++ memerlukan **kompilator**, yang mengubah berkas `.cpp` menjadi program, dan **debugger**, yang memungkinkan Anda menghentikan program yang sedang berjalan dan melihat isinya. Kita memakai **GCC** (perintah `g++` dan `gdb`) di Windows dan Linux, serta **Clang** (`clang++` dan `lldb`) di macOS. Keduanya memahami C++17 standar, satu-satunya yang dipakai buku ini. Rilis baru mana pun sudah cukup; pada September 2026 yang terbaru adalah GCC 16.2, dirilis 7 Agustus 2026 [16], dan Clang 23.1, dirilis 26 Agustus 2026 [17]. Kompilator terbaru memakai standar C++ yang lebih baru bila tidak diberi tahu, dan itulah sebabnya setiap perintah dalam buku ini menyertakan `-std=c++17` secara eksplisit. **Tabel 1.8** menghimpun prosedurnya; tiga daftar langkah demi langkah sesudahnya menjelaskan setiap baris.

Tabel 1.8 Memasang dan memakai kompilator C++ di setiap sistem operasi. Fedora memakai `sudo dnf install gcc-c++ gdb` dan Arch `sudo pacman -S gcc gdb` pada langkah 1.

Langkah	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
1. Pasang	MSYS2 dari `https://www.msys2.org`; di terminal **MSYS2 UCRT64**: `pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain`	`xcode-select --install`	`sudo apt update && sudo apt install build-essential gdb`
2. Masukkan ke PATH	Tambahkan `C:\msys64\ucrt64\bin` ke **Path** pengguna; buka terminal baru	Tidak perlu apa-apa	Tidak perlu apa-apa
3. Periksa	`g++ --version` dan `gdb --version`	`clang++ --version`	`g++ --version` dan `gdb --version`
4. Kompilasi	`g++ -std=c++17 -O2 -Wall -Wextra -o hello hello.cpp`	`clang++ -std=c++17 -O2 -Wall -Wextra -o hello hello.cpp`	seperti Windows
5. Jalankan	`.\hello.exe`	`./hello`	`./hello`

Windows, langkah demi langkah. Windows tidak mempunyai kompilator C++ sendiri yang dapat dipakai dari terminal, sehingga kita mengambil GCC dari **MSYS2**, kumpulan kecil yang terawat baik berisi perkakas yang dipakai pemrogram Linux, yang juga merupakan jalur yang dianjurkan dokumentasi VS Code [14], [18].

1. Unduh pemasangnya dari `https://www.msys2.org` dan jalankan, dengan menerima folder `C:\msys64`.
2. Dari menu Start bukalah **MSYS2 UCRT64**. Namanya penting: UCRT64 adalah varian modern, dan paket yang dipasang dari terminal MSYS2 lain tidak akan cocok. Ketik `pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain`, tekan Enter untuk menerima semua anggota kelompoknya, lalu jawab `Y`.
3. Tekan tombol Windows, ketik *environment variables*, pilih **Edit environment variables for your account**, pilih **Path**, klik **Edit → New**, ketik `C:\msys64\ucrt64\bin`, lalu klik **OK** di setiap jendela.
4. Buka jendela PowerShell yang *baru* dan ketik `g++ --version` serta `gdb --version`. Masing-masing semestinya mencetak sebaris versi.

macOS, langkah demi langkah. Apple menyediakan Clang dalam *Command Line Tools*-nya; cukup dimintakan. Buka **Terminal** (Cmd+Space, ketik *Terminal*, Enter), ketik `xcode-select --install`, klik **Install** pada jendela yang muncul, lalu tunggu. Sesudah itu `clang++ --version` mencetak sebaris versi. Perintah `g++` juga berfungsi di Mac, tetapi diam-diam menjalankan Clang; setiap program dalam buku ini dapat dikompilasi dengan keduanya. *Debugger* di macOS adalah `lldb`, yang dipakai VS Code secara otomatis.

Linux, langkah demi langkah. Buka terminal (Ctrl+Alt+T di Ubuntu) lalu ketik `sudo apt update && sudo apt install build-essential gdb`, masukkan kata sandi Anda, dan konfirmasikan. `build-essential` membawa `gcc`, `g++`, dan `make`.

1.7.4 Java: JDK 25

Program Java dikompilasi dengan **JDK**, Java Development Kit, yang berisi kompilator `javac` dan peluncur `java`. Pasanglah **JDK 25**, rilis dukungan jangka panjang (LTS) September 2025, yang menerima pembaruan selama bertahun-tahun; JDK 27, dirilis 15 September 2026, adalah rilis

fitur terbaru, yang hanya didukung sampai rilis berikutnya muncul enam bulan kemudian [19], [20]. Build **Eclipse Temurin** dari `https://adoptium.net` gratis dan identik di ketiga sistem [21]. Program Java buku ini memerlukan Java 17 atau yang lebih baru dan dikompilasi di sini dengan JDK 21.

Tabel 1.9 Memasang JDK 25 di setiap sistem operasi.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Pasang	`winget install EclipseAdoptium.Temurin.25.JDK`, atau `.msi` dari adoptium.net dengan **Set JAVA_HOME** dicentang	`brew install --cask temurin@25`, atau `.pkg` dari adoptium.net	`sudo apt install openjdk-25-jdk` (Ubuntu 26.04), atau Temurin dari repositori apt Adoptium
Periksa	`java -version` dan `javac -version`	sama	sama

Kedua pemeriksaan harus menjawab, dengan nomor versi yang sama. Jika `java` menjawab tetapi `javac` tidak, yang terpasang hanyalah *runtime*, yang dapat menjalankan program Java tetapi tidak dapat mengompilasinya; pasanglah JDK.

1.7.5 Python 3.14

Pasanglah **Python 3.14**, seri yang terkini; rilis terbarunya pada September 2026 adalah 3.14.7, tertanggal 5 Agustus 2026 [22]. Program Python buku ini memerlukan Python 3.10 atau yang lebih baru (untuk pernyataan `match` dalam **Kode 1.9**) dan dijalankan di sini dengan Python 3.11.

Tabel 1.10 Memasang Python di setiap sistem operasi.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Pasang	Pemasang dari `https://www.python.org`; di layar pertamanya centang **Add python.exe to PATH**. Atau `winget install Python.Python.3.14`	Pemasang dari python.org, atau `brew install python@3.14`	`python3` bawaan sistem: `sudo apt install python3`; Ubuntu 26.04 membawa Python 3.14
Jalankan program	`python hello.py`	`python3 hello.py`	`python3 hello.py`
Periksa	`python --version`	`python3 --version`	`python3 --version`

Di Windows perintahnya `python`; di macOS dan Linux `python3`, karena pada sistem lama `python` berarti Python 2 yang sudah lama pensiun. Di tempat buku ini menulis `python3`, pengguna Windows mengetik `python`.

1.7.6 Ekstensi VS Code

Buka tampilan Extensions (ikon persegi di bilah kiri, atau **Ctrl+Shift+X**, **Cmd+Shift+X** di Mac), cari setiap nama dalam **Tabel 1.11**, lalu tekan **Install**, atau ketikkan ketiga perintah di bawah tabel ke terminal mana pun.

Tabel 1.11 Ekstensi VS Code yang dipakai buku ini.

Ekstensi	Penerbit	Pengenal	Memberi Anda
C/C++ Extension Pack	Microsoft	`ms-vscode.cpptools-extension-pack`	pewarnaan sintaks C++, garis bawah galat, pelengkapan, *debugger* GDB/LLDB
Extension Pack for Java	Microsoft	`vscjava.vscode-java-pack`	penyuntingan, menjalankan, *debugging*, dan pengujian Java
Python	Microsoft	`ms-python.python`	penyuntingan, menjalankan, *debugging*, dan lingkungan virtual Python

```
code --install-extension ms-vscode.cpptools-extension-pack
code --install-extension vscjava.vscode-java-pack
code --install-extension ms-python.python
```

1.7.7 Memeriksa seluruh meja kerja

Buka terminal *baru*, agar ia melihat PATH yang sudah diperbarui, lalu ketikkan setiap perintah dalam **Tabel 1.12**. Di Windows, ketik `python` di tempat tabel menulis `python3`.

Tabel 1.12 Perintah-perintah pemeriksaan, apa yang semestinya dicetak masing-masing, dan perbaikan yang lazim bila tidak.

Perintah	Jawaban yang baik diawali	Bila gagal
`code --version`	`1.`	Pasang ulang VS Code dengan **Add to PATH**, atau jalankan *Shell Command: Install 'code' command in PATH*
`g++ --version`	`g++` (di Mac: `Apple clang`)	Windows: tambahkan `C:\msys64\ucrt64\bin` ke Path lalu buka terminal baru
`gdb --version`	`GNU gdb` (di Mac pakai `lldb --version`)	Windows: ulangi langkah `pacman` di **MSYS2 UCRT64**
`javac -version`	`javac 25`	Hanya *runtime* yang terpasang, atau JDK lama berada lebih awal di PATH
`java -version`	`openjdk version "25`	Pasang JDK; periksa `JAVA_HOME`
`python3 --version`	`Python 3.14`	Windows: jalankan ulang pemasangnya dan centang **Add python.exe to PATH**

Pada mesin Linux tempat program-program bab ini dijalankan, pemeriksaan itu mencetak yang berikut. Mesin ini sedikit tertinggal dari versi yang dianjurkan, dengan GCC 13, JDK 21, dan Python 3.11, yang memperlihatkan bahwa program-program buku ini juga berjalan pada versi-versi tersebut.

```
$ g++ --version
g++ (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0
$ gdb --version
GNU gdb (Ubuntu 15.1-1ubuntu1~24.04.1) 15.1
$ javac -version
javac 21.0.10
$ java -version
openjdk version "21.0.10" 2026-01-20
$ python3 --version
Python 3.11.15
```

1.7.8 Program pertama Anda, dalam tiga bahasa

Buatlah berkas `hello.cpp` di folder `Chapter01` Anda, ketikkan **Kode 1.41**, lalu simpan.

Kode 1.41 `hello.cpp`: program pertama; ia membuktikan bahwa kompilator, editor, dan terminal bekerja bersama.

```cpp
// hello.cpp - program pertama buku ini
#include <iostream>
#include <string>
using namespace std;

int main() {
    string name;
    cout << "What is your name? ";
    if (!getline(cin, name) || name.empty()) {
        name = "friend";                  // tanpa masukan: sapa seorang teman
    }
    cout << "Hello, " << name << "! Welcome to Data Structure.\n";
    cout << "Language version: C++ " << __cplusplus << '\n';
    return 0;
}
```

Buka terminal di VS Code dengan **Terminal → New Terminal** lalu kompilasi dengan baris dari langkah 4 **Tabel 1.8**:

```
g++ -std=c++17 -O2 -Wall -Wextra -o hello hello.cpp
```

Setiap bagian baris itu mempunyai tugas. `-std=c++17` memilih standar C++17; `-O2` meminta kompilator mengoptimalkan, agar pengukuran waktu apa pun mengukur algoritmanya dan bukan terjemahan yang malas; `-Wall -Wextra` menyalakan peringatan, yang menangkap banyak kesalahan sebelum menjadi *bug*; `-o hello` menamai program hasilnya. Jika kompilator tidak mencetak apa-apa, ia berhasil: diam adalah kabar baik. Jalankan programnya (`./hello`, atau `.\hello.exe` di Windows) lalu ketikkan sebuah nama:

```
What is your name? Sari
Hello, Sari! Welcome to Data Structure.
Language version: C++ 201703
```

Baris terakhir adalah bukti kecil: `201703` adalah nilai yang diberikan standar C++ kepada nama bawaan `__cplusplus` di bawah C++17, sehingga kini Anda tahu pasti standar mana yang dipakai untuk mengompilasi program-program Anda.

Sekarang versi Java, **Kode 1.42**. Simpanlah sebagai `Hello.java`: di Java, kelas publik harus tinggal dalam berkas yang namanya persis sama dengan nama kelasnya, termasuk huruf kapitalnya.

Kode 1.42 `Hello.java`: program pertama di Java.

```java
// Hello.java - program pertama buku ini
import java.util.Scanner;

public class Hello {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("What is your name? ");
        String name = in.hasNextLine() ? in.nextLine() : "";
        if (name.isEmpty()) {
            name = "friend";                  // tanpa masukan: sapa seorang teman
        }
        System.out.println("Hello, " + name + "! Welcome to Data Structure.");
        System.out.println("Language version: Java " + Runtime.version().feature());
    }
}
```

Kompilasi dan jalankan dalam dua langkah; sejak Java 11, `java Hello.java` juga bekerja dalam satu langkah, karena peluncurnya dapat mengompilasi program satu berkas dengan sendirinya:

```
javac Hello.java
java Hello
```

```
What is your name? Sari
Hello, Sari! Welcome to Data Structure.
Language version: Java 21
```

Dengan JDK 25 baris terakhirnya berbunyi `Java 25`. Terakhir, versi Python, **Kode 1.43**, disimpan sebagai `hello.py` dan dijalankan dengan `python3 hello.py` (`python hello.py` di Windows):

Kode 1.43 `hello.py`: program pertama di Python.

```
# hello.py - program pertama buku ini
import sys

try:
    name = input("What is your name? ")
except EOFError:
    name = ""
if not name:
    name = "friend"                      # tanpa masukan: sapa seorang teman
print(f"Hello, {name}! Welcome to Data Structure.")
print(f"Language version: Python {sys.version_info.major}.{sys.version_info.minor}")
```

```
What is your name? Sari
Hello, Sari! Welcome to Data Structure.
Language version: Python 3.11
```

Dengan Python 3.14 baris terakhirnya berbunyi `Python 3.14`. Ketiga program menangani keadaan ketika tidak ada masukan sama sekali, misalnya ketika dijalankan dengan berkas kosong yang dialihkan ke dalamnya: mereka menyapa “friend”.

1.7.9 Menjalankan dari VS Code, mengumpankan masukan, dan debugging

Menjalankan dari editor. Dengan ekstensi-ekstensi dalam **Tabel 1.11** terpasang, berkas Java atau Python menampilkan segitiga **Run** di kanan atas editor, dan berkas C++ menampilkan **Run C/C++ File**, yang mengompilasi berkas itu dengan kompilator yang ditemukannya lalu menjalankannya di terminal. Pada kali pertama, VS Code menanyakan kompilator mana yang dipakai dan menuliskan pilihannya ke berkas `.vscode/tasks.json` di folder Anda. Untuk memakai *flag* persis seperti buku ini, tambahkan `"-std=c++17", "-Wall", "-Wextra"` ke daftar `args` dalam berkas itu.

Mengumpankan masukan dari berkas. Program seperti `io_sum` membaca masukan standar. Alih-alih mengetik bilangan-bilangannya setiap kali, alihkan sebuah berkas, seperti dilakukan Subbab 1.4.3: `./io_sum < week_loans.txt`, `java IoSum < week_loans.txt`, atau `python3 io_sum.py < week_loans.txt` di macOS, Linux, dan Command Prompt Windows, dan `Get-Content week_loans.txt | .\io_sum.exe` di PowerShell, yang tidak memahami `<`.

Debugging: mengamati program berpikir. Ketika program memberi jawaban yang salah, cara tercepat untuk mengetahui sebabnya adalah *mengamati*-nya. Klik di margin sebelah kiri nomor baris: muncul titik merah, sebuah **breakpoint**. Tekan **F5** lalu pilih *debugger* untuk bahasa Anda (untuk C++, *C++ (GDB/LLDB)* dan kompilator Anda). Program berjalan sampai mencapai titik merah itu lalu berhenti; panel **Variables** kemudian memperlihatkan nilai setiap variabel pada saat itu, dan Anda dapat mengarahkan tetikus ke nama mana pun dalam kode untuk melihat nilainya. **F10** menjalankan satu baris, **F11** masuk ke dalam pemanggilan fungsi, **Shift+F11** keluar lagi, dan **F5** berlanjut sampai *breakpoint* berikutnya. Pasanglah *breakpoint* di dalam perulangan `reverseInPlace` dalam **Kode 1.22** dan amatilah `i` dan `j` berjalan saling mendekat: itulah cara terbaik memahami perulangan yang tidak Anda tulis sendiri.

> **BILA ADA YANG SALAH.** *"g++ is not recognized"* (Windows): entri Path belum ada, atau terminal dibuka sebelum entri itu ditambahkan; bukalah terminal baru. *"fatal error: iostream: No such file or directory"*: kompilator dipasang dari terminal MSYS2 yang salah; ulangi langkah `pacman` di **MSYS2 UCRT64**. *Jendela program berkedip lalu tertutup*: jalankan dari terminal, bukan dengan mengklik ganda. *Garis bawah galat di mana-mana di VS Code, padahal program dapat dikompilasi*: buka Command Palette (**Ctrl+Shift+P**), jalankan **C/C++: Select IntelliSense Configuration**, lalu pilih kompilator yang Anda pasang. *`javac` tidak ditemukan tetapi `java` ada*: Anda memasang *runtime*; pasanglah JDK. *`error: class Hello is public, should be declared in a file named Hello.java`*: ganti nama berkasnya agar persis sama dengan nama kelasnya. *Python mencetak `SyntaxError` pada baris `match`*: Python-nya lebih tua dari 3.10; pasanglah 3.14.

Dengan meja kerja yang sudah berdiri, setiap kode program dalam sisa buku ini hanya berjarak satu salin, satu kompilasi, dan satu kali jalan dari menjadi milik Anda.

1.8 Profitina dalam Praktik

Profitina, platform visibilitas AI di app.profitina.com, mengukur seberapa siap situs web sebuah merek untuk dikutip oleh mesin penjawab berbasis AI, lalu membantu memperbaiki apa yang ditemukannya. Di balik dasbor-dasbornya, sebagian besar data yang ditanganinya berbentuk persis seperti dalam bab ini, dan biaya-biaya dalam **Tabel 1.6** adalah biaya yang harus dijalaninya.

Sederet skor tetap untuk setiap halaman. Sebuah pemindaian memeriksa setiap halaman situs terhadap daftar sinyal yang tetap, seperti seberapa mudah kontennya diekstraksi, apakah halaman itu membawa data terstruktur, dan seberapa sehat halaman itu secara teknis, lalu memberi skor pada masing-masing. Bila daftar sinyalnya tetap, rumah yang wajar bagi skor-skor satu halaman adalah array berukuran tetap dengan satu slot per sinyal: slot k selalu berarti sinyal k, membaca skor mana pun berbiaya $O(1)$ menurut Persamaan (1.2), dan menggabungkan skor-skor itu menjadi satu angka adalah satu penelusuran $O(n)$ seperti `sumOf` dalam **Kode 1.22**. Halaman-halaman sebuah situs lalu membentuk tabel, satu baris per halaman dan satu kolom per sinyal, persis bentuk **Kode 1.28**; total sebuah kolom menunjukkan bagaimana seluruh situs bernasib pada satu sinyal, dan total sebuah baris menunjukkan bagaimana satu halaman bernasib pada semua sinyal.

Riwayat yang hanya tumbuh di ujung. Dasbor analitik historis memperlihatkan skor setiap merek dari minggu ke minggu, dan penjadwal menambahkan satu titik baru sesudah setiap pemindaian ulang otomatis. Deret yang hanya pernah tumbuh di ujungnya adalah kasus ideal bagi array yang dapat tumbuh: penambahan di ujung berbiaya $O(1)$ teramortisasi (Persamaan (1.5)), skor minggu ke-i dibaca dalam $O(1)$, dan ekspor CSV adalah satu kali penelusuran. Tidak ada yang pernah disisipkan di tengah, sehingga baris $O(n)$ dalam **Tabel 1.6** tidak pernah dibayar. *Back end* Profitina ditulis dalam Python dengan FastAPI, sehingga wadah sehari-harinya adalah `list` Python dari Subbab 1.5.10.

Tempat bagian tengah menjadi mahal. Umpan komunitas dan notifikasi yang dihimpun di dasbor ditampilkan dari yang terbaru. Seandainya butir-butir baru disisipkan di indeks 0 sebuah array, setiap penyisipan akan menggeser setiap butir yang lebih lama, n geseran setiap kali menurut Persamaan (1.6). Obat bakunya adalah menambahkan di ujung, yang murah, lalu *membaca* array itu mundur ketika menampilkannya; *queue* di Bab 5 dan *priority queue* di Bab 14 adalah perkakas untuk kasus-kasus ketika itu belum cukup. Papan peringkat PROFIT memunculkan pertanyaan yang sama dalam bentuk lain: menjaga array berperingkat tetap

terurut persis selagi skor berubah berarti memindahkan entri naik turun di dalamnya, dan Bab 3 serta Bab 14 mempelajari cara-cara yang lebih baik.

Jujur tentang batas. Array berukuran tetap gagal dengan lantang atau diam-diam ketika meluap, bergantung pada bahasanya, seperti diperlihatkan Subbab 1.5.4. Aturan rumah Profitina adalah gagal dengan lantang: ketika model AI lokal yang menggerakkan Content Engine tidak tersedia, aplikasi mengatakannya alih-alih diam-diam menyajikan templat seolah-olah berasal dari model. Kebiasaan C++ memakai `at()` alih-alih `[]` yang tak berpemeriksa, serta penolakan Java dan Python untuk membaca melewati ujung array, adalah prinsip yang sama pada skala terkecil.

1.9 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Perulangan satu langkah terlalu jauh. `for (int i = 0; i <= n; i++)` mengunjungi `a[n]` yang tidak ada. Di C++ ini perilaku tak terdefinisi yang mungkin tidak ketahuan; di Java dan Python ia melempar eksepsi. *Hindari:* kondisinya selalu `i < n`. Lebih baik lagi, pakailah `for` berbasis rentang di C++, `for` yang disempurnakan di Java, atau `for x in a` di Python bila indeksnya tidak diperlukan.

Mulai dari 1. Menulis `for (int i = 1; i <= n; i++)` karena "buku pertama adalah buku nomor satu" melewatkan `a[0]` dan membaca `a[n]`. *Hindari:* indeks adalah jarak dari titik awal, Persamaan (1.2); yang pertama adalah 0.

Memakai array yang belum diisi. `int a[8];` di dalam fungsi berisi sisa-sisa byte, bukan nol. *Hindari:* tulislah `int a[8] = {0};`, atau `std::array<int, 8> a{};`, atau isilah sebelum membaca.

Menghitung panjang di dalam fungsi dengan `sizeof`. Di dalam fungsi, parameter array adalah pointer, dan `sizeof a / sizeof a[0]` memberikan $8/4 = 2$ pada mesin 64-bit, berapa pun panjang array sebenarnya. *Hindari:* oper panjangnya sebagai parameter, atau pakailah `std::array` atau `std::vector`, yang mengetahui `size()`-nya.

Kehilangan nilai ketika menukar. `a[i] = a[j]; a[j] = a[i];` membuat kedua slot berisi `a[j]` yang lama. *Hindari:* pakailah variabel sementara, `int t = a[i]; a[i] = a[j]; a[j] = t;`, atau `std::swap(a[i], a[j])`, atau `a[i], a[j] = a[j], a[i]` di Python.

Menggeser ke arah yang salah. Membuat ruang bagi penyisipan dengan memindahkan elemen dari *depan* ke belakang menimpa setiap elemen sebelum ia sempat dipindahkan. *Hindari:* penyisipan menggeser dari ujung ke belakang, penghapusan dari celah ke depan, seperti dalam **Pseudokode 1.1** dan **Pseudokode 1.2**.

Tertukar baris dan kolom. `m[c][r]` alih-alih `m[r][c]` membaca elemen yang salah, atau keluar batas bila tabelnya tidak bujur sangkar. *Hindari:* namai indeksnya `r` dan `c` (atau `s` dan `p`), jangan pernah `i` dan `j`, dalam kode dua dimensi, dan tempatkan baris lebih dahulu.

Mengira "array 2-D" Java atau Python adalah satu blok. `Book[][]` atau list berisi list adalah array berisi referensi ke baris-baris; baris-barisnya boleh berbeda panjang dan bahkan boleh dipakai bersama. *Hindari:* periksalah `a[r].length` atau `len(a[r])` untuk setiap baris bila baris-barisnya mungkin berbeda, dan jangan pernah membangun tabel Python dengan `[[0] * c] * r`; pakailah `[[0] * c for _ in range(r)]` (**Kode 1.31**).

Menyalin referensi lalu menyebutnya salinan. Di Java, `int[] b = a;` dan di Python `b = a` menghasilkan *nama* kedua untuk array yang sama; mengubah `b[0]` mengubah `a[0]`. Di C++, `int`

`b[8] = a;` bahkan tidak lolos kompilasi. *Hindari:* salinlah dengan `Arrays.copyOf(a, a.length)` di Java, `a.copy()` atau `list(a)` di Python, dan penugasan `std::array` atau `std::vector` di C++.

Menyisipkan di depan array besar di dalam perulangan. Setiap penyisipan menggeser segalanya, sehingga n penyisipan berbiaya sekitar $n^2/2$ geseran. *Hindari:* tambahkan di ujung lalu balik sekali di akhir, atau pakailah struktur yang dibangun untuk penyisipan di kedua ujung, seperti *queue* di Bab 5.

Luapan bilangan bulat dan pembagian bilangan bulat. Menjumlahkan sejuta `int` yang masing-masing sekitar 5.000 meluapkan sebuah `int`, dan `sum / n` dengan dua bilangan bulat membuang pecahannya. *Hindari:* akumulasikan dalam `long` atau `long long`, dan bagilah sebagai `static_cast<double>(sum) / n`.

Lupa `delete`, atau menghapus dua kali. Setiap `new` memerlukan tepat satu `delete`. *Hindari:* dalam buku ini, utamakan `std::vector` dan wadah lain, yang membebaskan memorinya sendiri; Bab 6 membahas `new` dan `delete` secara semestinya.

1.10 Ringkasan Bab

- **Struktur data** adalah cara menata data di memori, bersama operasi-operasinya, biasanya **penyisipan**, **penghapusan**, **pencarian**, **penelusuran**, dan **pengurutan**. *Antarmuka*-nya adalah apa yang dijanjikannya; *implementasi*-nya adalah bagaimana. Pilihan yang tepat dapat mengubah banyaknya langkah dari n menjadi $\log_2 n$ atau menjadi 1 (**Tabel 1.1**), yang tidak tertandingi komputer secepat apa pun.
- **Memori** adalah deretan byte, masing-masing dengan **alamat**; k bit mempunyai 2^k keadaan (Persamaan (1.1)), sehingga sebuah byte menampung 0 sampai 255. **Tipe** menentukan berapa byte yang ditempati sebuah nilai dan bagaimana byte itu dibaca. Kebanyakan mesin bersifat **little-endian**. Kompilator dapat menyisipkan **padding** untuk menyelaraskan nilai.
- Variabel lokal tinggal di *frame* fungsinya di **stack**, dilepaskan otomatis ketika fungsi kembali; memori dari `new` tinggal di **heap** sampai `delete`. Java dan Python menaruh objek di *heap* dan membebaskannya dengan **garbage collector**.
- **Penyegaran C++** mencakup kerangka program, tipe (**Tabel 1.4**), `cin` dan `cout`, `if`, `switch`, `while`, `for`, dan `for` berbasis rentang, fungsi dengan parameter yang dioper menurut nilai, struct, serta operator `&` (alamat dari) dan `*` (ikuti pointer). Java dan Python memakai referensi alih-alih alamat yang kelihatan.
- **Array** adalah deretan bersinambung berisi n elemen bertipe sama, berindeks 0 sampai $n-1$. Elemen i berada di $b + i \cdot s$ (Persamaan (1.2)), yang membuat **akses bernilai** $O(1)$. `a[i]` adalah `*(a + i)`. C++ tidak memeriksa batas; `at()` memeriksanya. Java dan Python selalu memeriksa, dan Python menerima indeks negatif.
- **Array dua dimensi** di C++ disimpan secara **row-major**: elemen $[r][c]$ berada di $b + (r \cdot C + c) \cdot s$ (Persamaan (1.3)); urutan column-major memakai Persamaan (1.4). Java dan Python membangun tabel sebagai array berisi array baris.
- `std::array` adalah array berukuran tetap yang mengetahui ukurannya; `std::vector`, `ArrayList`, dan `list` Python tumbuh dengan memesan blok yang lebih besar lalu menyalin. Tumbuh dengan faktor tetap menjaga total salinan di bawah suatu konstanta kali n (Persamaan (1.5)), sehingga penambahan di ujung berbiaya $O(1)$ **teramortisasi**. Modul `array` Python menyimpan nilai mentah secara padat.

- **Penyisipan dan penghapusan** di tengah menggeser $n - pos$ elemen (Persamaan (1.6)), rata-rata $n/2$ (Persamaan (1.7)): $O(n)$. **Big-O** (Persamaan (1.8)) mempertahankan bentuk pertumbuhan dan membuang faktor konstan; **Tabel 1.6** meringkas biaya-biaya array.
- **Meja kerja**-nya adalah VS Code dengan ekstensi C/C++, Java, dan Python, GCC melalui MSYS2 di Windows, Clang melalui Command Line Tools di macOS, `build-essential` di Linux, JDK 25, dan Python 3.14, yang diperiksa dengan perintah-perintah dalam **Tabel 1.12**.

1.11 Soal Latihan

Soal Latihan 1.1. (a) Sebutkan lima operasi yang biasa menjadi ukuran struktur data. (b) Untuk daftar kontak sebuah ponsel, berikan satu contoh konkret setiap operasi. (c) "Temukan kontak menurut nama" dapat disediakan oleh dua implementasi berbeda. Uraikan dua di antaranya, masing-masing dalam satu kalimat, dan sebutkan bagian mana dari definisi struktur data yang sama pada keduanya dan bagian mana yang tidak.

Soal Latihan 1.2. (a) Berapa banyak nilai berbeda yang dapat ditampung 16 bit? Dan 32 bit? (b) Nilai `int 0x12345678` disimpan di alamat 2000. Sebutkan byte yang tersimpan di setiap alamat 2000 sampai 2003 pada mesin little-endian dan pada mesin big-endian. (c) Tuliskan byte 11111111 dalam desimal dan dalam heksadesimal.

Soal Latihan 1.3. Array `double prices[50]` bermula di alamat 5000, dan sebuah `double` menempati 8 byte. (a) Di alamat berapa `prices[17]` bermula? (b) Elemen mana yang bermula di alamat 5320? (c) Adakah elemen yang bermula di alamat 5324? Jelaskan. (d) Berapa alamat byte tepat sesudah array itu?

Soal Latihan 1.4. Array `int m[6][9]` bermula di alamat 2000, dengan `int` berukuran 4 byte. (a) Hitunglah alamat `m[4][7]` dalam urutan row-major, sebagaimana C++ menyimpannya. (b) Hitunglah alamat yang akan dimiliki elemen yang sama dalam urutan column-major. (c) Dalam urutan row-major, elemen mana yang bermula di alamat $2000 + 4 \cdot 50$?

Soal Latihan 1.5. Turunkan rumus alamat row-major untuk array tiga dimensi `A[D1][D2][D3]` dengan alamat dasar b dan ukuran elemen s, dengan penalaran yang sama seperti Persamaan (1.3). Periksalah rumus itu dengan sebuah program pada `int a[2][3][4]`, dan berikan offset `a[1][2][3]` dari `a[0][0][0]`.

Soal Latihan 1.6. Dengan aturan dalam **Kode 1.7**, hitunglah dengan tangan biaya keterlambatan untuk (a) seorang dosen yang meminjam buku 15 hari, (b) seorang tamu yang meminjam buku 8 hari, (c) seorang mahasiswa yang meminjam buku 30 hari, dan (d) siapa saja yang mengembalikan buku tepat sesudah 7 hari. Lalu jalankan program dengan keempat peminjaman ini untuk memeriksanya.

Soal Latihan 1.7. Tulislah fungsi yang memutar (merotasi) array berisi n elemen sejauh k tempat ke kiri (sehingga `1 2 3 4 5 6 7` yang diputar sejauh 3 menjadi `4 5 6 7 1 2 3`) dalam waktu $O(n)$, dengan hanya memakai memori tambahan yang konstan, berapa pun k. *Petunjuk:* balikkan tiga bagian array.

Soal Latihan 1.8. Tulislah program yang mentranspos tabel peminjaman 3×4 dalam **Kode 1.28** menjadi tabel 4×3, satu baris per minggu, lalu mencetaknya.

Soal Latihan 1.9. (a) Sebuah array berisi $n = 1000$ elemen. Berapa geseran yang dilakukan `ARRAY-INSERT` untuk menyisipkan di indeks 0, di indeks 250, dan di indeks 1000? (b) Bermula dari array kosong dengan ruang untuk 1.000 elemen, sebuah program menyisipkan 1.000 nilai,

masing-masing di indeks 0. Berapa geseran seluruhnya? (c) Berapa bila setiap nilai disisipkan di ujung? (d) Nyatakan kedua total itu dalam notasi big-O.

Soal Latihan 1.10. Sebuah `std::vector` yang menggandakan kapasitasnya, seperti dalam **Kode 1.32**, menerima 1.000 kali `push_back` mulai dari kosong. (a) Berapa kali ia tumbuh? (b) Berapa elemen yang disalin seluruhnya? (c) Berapa kapasitas akhirnya? (d) Periksalah (b) terhadap Persamaan (1.5).

Soal Latihan 1.11. Ramalkan dengan tepat apa yang dicetak program Python ini, lalu perbaikilah sehingga hanya satu elemen yang berubah:

```
grid = [[0] * 3] * 2
grid[1][2] = 7
print(grid)
```

Soal Latihan 1.12. Temukan setiap kesalahan dalam potongan C++ ini, dan sebutkan mana yang akan dilaporkan kompilator:

```
int squares[5];
for (int i = 1; i <= 5; i++) {
    squares[i] = i * i;
}
int total;
for (int i = 0; i < 5; i++) total += squares[i];
cout << "average = " << total / 5 << '\n';
```

Soal Latihan 1.13. Untuk setiap variabel berikut, sebutkan apakah ia tinggal di data global, di *stack*, atau di *heap*: (a) `int counter = 0;` yang ditulis di luar semua fungsi dalam program C++; (b) `int n = 8;` di dalam `main`; (c) `int` yang dibuat oleh `new int(5)`; (d) variabel pointer `p` dalam `int* p = new int(5);` di dalam `main`; (e) elemen-elemen `std::vector<int> v(100);` yang dideklarasikan di dalam `main`; (f) objek array yang dibuat `new int[10]` di Java; (g) nilai `7` dalam pernyataan Python `x = 7`.

Soal Latihan 1.14. Andaikan rekaman `Book` dalam **Kode 1.13** mendapat field `int yearPublished;` yang ditaruh di antara `id` dan `title`. Ramalkan offset setiap field dan `sizeof(Book)`, mengingat bahwa sebuah `int` harus bermula di alamat yang habis dibagi 4 dan bahwa ukuran struct selalu kelipatan penyelarasan field-nya yang paling menuntut. Periksalah dengan sebuah program.

Soal Latihan 1.15. Seorang teman yang sedang menyiapkan meja kerja melaporkan tiga pesan. Jelaskan masing-masing dan berikan perbaikannya: (a) di Windows, `'g++' is not recognized as an internal or external command`, padahal MSYS2 sudah terpasang; (b) `java -version` mencetak versi tetapi `javac -version` tidak ditemukan; (c) `python3 late_fee.py` berhenti dengan `SyntaxError` pada baris `match member_type:`.

1.12 Penyelesaian Soal Latihan

Penyelesaian 1.1. (a) Penyisipan, penghapusan, pencarian, penelusuran, dan pengurutan. (b) Penyisipan: menyimpan kontak baru. Penghapusan: membuang sebuah kontak. Pencarian: menemukan “Sari” untuk meneleponnya. Penelusuran: menggulir seluruh kontak, atau mencadangkannya. Pengurutan: menampilkan kontak menurut abjad. (c) Satu implementasi menyimpan kontak dalam array yang terurut menurut nama dan membelah dua rentang pencarian pada setiap langkah; implementasi lain menyimpannya dalam tabel hash dengan nama sebagai kunci. Keduanya berbagi *antarmuka*, “berikan kontak bernama ini, atau katakan bahwa tidak ada”, dan berbeda dalam *implementasi*: susunan di memori dan langkah-langkah yang menemukan kontak, dan karena itu juga dalam biaya.

Penyelesaian 1.2. (a) Menurut Persamaan (1.1), $2^{16} = 65{,}536$ dan $2^{32} = 4{,}294{,}967{,}296$. (b) Byte-byte `0x12345678` dari yang paling bermakna sampai yang paling tidak bermakna adalah `12 34 56 78`. Little-endian menyimpan yang paling tidak bermakna lebih dahulu: 2000 berisi `78`, 2001 `56`, 2002 `34`, 2003 `12`. Big-endian menyimpannya menurut urutan baca: `12`, `34`, `56`, `78`. (c) $11111111 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$, yaitu `ff` dalam heksadesimal: setiap kelompok empat angka satu bernilai 15, angka `f`.

Penyelesaian 1.3. Menurut Persamaan (1.2) dengan $b = 5000$ dan $s = 8$: (a) $5000 + 17 \cdot 8 = 5136$. (b) $5320 = 5000 + i \cdot 8$ memberikan $i = 320/8 = 40$, jadi `prices[40]`. (c) Tidak: $(5324 - 5000)/8 = 40.5$ bukan bilangan bulat, sehingga 5324 adalah byte kelima di dalam `prices[40]`. (d) Array itu menempati $50 \cdot 8 = 400$ byte, dari 5000 sampai 5399, sehingga byte berikutnya adalah 5400, yang juga merupakan alamat hasil hitungan `prices + 50`; alamat itu boleh dihitung tetapi tidak boleh dibaca.

Penyelesaian 1.4. Di sini $R = 6$, $C = 9$, $s = 4$, dan $b = 2000$. (a) Menurut Persamaan (1.3): $2000 + (4 \cdot 9 + 7) \cdot 4 = 2000 + 43 \cdot 4 = 2172$. (b) Menurut Persamaan (1.4): $2000 + (7 \cdot 6 + 4) \cdot 4 = 2000 + 46 \cdot 4 = 2184$. (c) Posisi 50 dalam deretan yang digelar adalah baris $\lfloor 50/9 \rfloor = 5$, kolom $50 - 5 \cdot 9 = 5$: elemen `m[5][5]`.

Penyelesaian 1.5. Gelarlah `A[D1][D2][D3]` menjadi satu deretan. Elemen `A[i][j][k]` didahului i "bidang" utuh yang masing-masing berisi $D_2 \cdot D_3$ elemen, lalu j baris utuh berisi D_3 elemen di bidangnya sendiri, lalu k elemen di barisnya sendiri. Posisinya adalah $i \cdot D_2 D_3 + j \cdot D_3 + k$, yang dapat difaktorkan menjadi $(i \cdot D_2 + j) \cdot D_3 + k$, sehingga

$$\text{alamat}(A[i][j][k]) = b + ((i \cdot D_2 + j) \cdot D_3 + k) \cdot s \tag{1.9}$$

Seperti Persamaan (1.3), Persamaan (1.9) tidak memakai dimensi pertama D_1. Untuk `int a[2][3][4]`, `a[1][2][3]` berada di posisi $(1 \cdot 3 + 2) \cdot 4 + 3 = 23$, yaitu $23 \cdot 4 = 92$ byte sesudah `a[0][0][0]`; itulah elemen terakhir dari 24 elemen. **Kode 1.44** mengukur setiap alamat; **Kode 1.45** dan **Kode 1.46**, yang tidak dapat melihat alamat, menyimpan elemen-elemennya dalam satu array datar di posisi yang dihitung Persamaan (1.9) dan memeriksa bahwa posisi-posisi itu muncul menurut urutan kunjungan penelusuran row-major. Ketiganya mencetak dua baris yang tampak sesudah kode-kode program itu.

Kode 1.44 `sol_address3d.cpp`: Persamaan (1.9) diperiksa terhadap alamat sebenarnya dari ke-24 elemen.

```cpp
// sol_address3d.cpp - Penyelesaian 1.5: rumus alamat row-major dalam 3-D
#include <iostream>
using namespace std;

const int D1 = 2, D2 = 3, D3 = 4;

int main() {
    int a[D1][D2][D3];
    const char* base = reinterpret_cast<const char*>(&a[0][0][0]);
    int checked = 0, agreed = 0;
    for (int i = 0; i < D1; i++) {
        for (int j = 0; j < D2; j++) {
            for (int k = 0; k < D3; k++) {
                const char* here = reinterpret_cast<const char*>(&a[i][j][k]);
                long actual = static_cast<long>(here - base);
                long position = (i * D2 + j) * D3 + k;
                long formula = position * static_cast<long>(sizeof(int));
                checked++;
                if (actual == formula) agreed++;
            }
        }
    }
    cout << "int a[" << D1 << "][" << D2 << "][" << D3 << "]: " << agreed << " of "
         << checked << " addresses agree with the formula\n";
    long offset = ((1 * D2 + 2) * D3 + 3) * static_cast<long>(sizeof(int));
    cout << "a[1][2][3] is " << offset << " bytes after a[0][0][0]\n";
    return 0;
}
```

Kode 1.45 `SolAddress3D.java`: rumus posisi tiga dimensi diperiksa di Java.

```java
// SolAddress3D.java - Penyelesaian 1.5: rumus posisi row-major dalam 3-D
public class SolAddress3D {
    static final int D1 = 2, D2 = 3, D3 = 4;

    public static void main(String[] args) {
        // Simpan a[i][j][k] datar, di satu int[] berisi D1*D2*D3 slot, seperti C++.
        int[] flat = new int[D1 * D2 * D3];
        int next = 0, checked = 0, agreed = 0;
        for (int i = 0; i < D1; i++) {
            for (int j = 0; j < D2; j++) {
                for (int k = 0; k < D3; k++) {
                    int position = (i * D2 + j) * D3 + k;
                    flat[position] = next;    // next: urutan kunjungan row-major
                    checked++;
                    if (flat[position] == position) agreed++;
                    next++;
                }
            }
        }
        System.out.println("int a[" + D1 + "][" + D2 + "][" + D3 + "]: " + agreed
                + " of " + checked + " addresses agree with the formula");
        int offset = ((1 * D2 + 2) * D3 + 3) * Integer.BYTES;
        System.out.println("a[1][2][3] is " + offset + " bytes after a[0][0][0]");
    }
}
```

Kode 1.46 `sol_address3d.py`: rumus posisi tiga dimensi diperiksa di Python.

```python
# sol_address3d.py - Penyelesaian 1.5: rumus posisi row-major dalam 3-D
D1, D2, D3 = 2, 3, 4
INT_BYTES = 4

# Simpan a[i][j][k] datar, di satu list berisi D1*D2*D3 slot, seperti C++.
flat = [None] * (D1 * D2 * D3)
visit = 0
checked = agreed = 0
for i in range(D1):
    for j in range(D2):
        for k in range(D3):
            position = (i * D2 + j) * D3 + k
            flat[position] = visit       # visit adalah urutan kunjungan row-major
            checked += 1
            if flat[position] == position:
                agreed += 1
            visit += 1
print(f"int a[{D1}][{D2}][{D3}]: {agreed} of {checked}"
      " addresses agree with the formula")
offset = ((1 * D2 + 2) * D3 + 3) * INT_BYTES
print(f"a[1][2][3] is {offset} bytes after a[0][0][0]")
```

```
int a[2][3][4]: 24 of 24 addresses agree with the formula
a[1][2][3] is 92 bytes after a[0][0][0]
```

Penyelesaian 1.6. (a) Dosen, 15 hari: terlambat 8 hari, dasar $7 \cdot 1000 + 1 \cdot 2000 = 9{,}000$, harga penuh: Rp9.000. (b) Tamu, 8 hari: terlambat 1 hari, dasar 1.000, dilipatduakan: Rp2.000. (c) Mahasiswa, 30 hari: terlambat 23 hari, dasar $7000 + 16 \cdot 2000 = 39{,}000$, diparuh: Rp19.500. (d) Tepat 7 hari: `daysLate` bernilai 0, dan `if` pertama mengembalikan 0 sebelum jenis anggota sempat diperiksa, sehingga biayanya Rp0 bagi siapa pun, termasuk jenis anggota yang tak dikenal. Mengganti keenam peminjaman di `main` dengan keempat peminjaman ini lalu menjalankan programnya mencetak persis biaya-biaya ini dan total Rp30.500.

Penyelesaian 1.7. Memutar ke kiri sejauh k sama dengan memutar sejauh $k \bmod n$, dengan mod memberikan sisa pembagian, jadi kecilkan k lebih dahulu. Lalu balikkan k elemen pertama, balikkan $n - k$ elemen sisanya, dan balikkan seluruh array. Dengan `1 2 3 4 5 6 7` dan $k = 3$: membalik `1 2 3` menghasilkan `3 2 1 4 5 6 7`; membalik `4 5 6 7` menghasilkan `3 2 1 7 6 5 4`; membalik semuanya menghasilkan `4 5 6 7 1 2 3`. Setiap pembalikan m elemen melakukan $\lfloor m/2 \rfloor$ penukaran, sehingga totalnya paling banyak n penukaran: waktu $O(n)$, dan hanya variabel `t` sebagai memori tambahan. **Kode 1.47**, **Kode 1.48**, dan **Kode 1.49** mengimplementasikannya; memutar sekali lagi sejauh 11, yang sama dengan 4 modulo 7, mengembalikan array ke keadaan semula, karena $3 + 4 = 7$ tempat adalah satu putaran penuh.

Kode 1.47 `sol_rotate.cpp`: rotasi dengan tiga pembalikan.

```cpp
// sol_rotate.cpp - Penyelesaian 1.7: rotasi kiri sejauh k dengan tiga pembalikan
#include <iostream>
using namespace std;

void reverseRange(int a[], int lo, int hi) {         // membalik a[lo..hi]
    while (lo < hi) {
        int t = a[lo];
        a[lo] = a[hi];
        a[hi] = t;
        lo++;
        hi--;
    }
}

void rotateLeft(int a[], int n, int k) {
    k %= n;
    reverseRange(a, 0, k - 1);
    reverseRange(a, k, n - 1);
    reverseRange(a, 0, n - 1);
}

void print(const int a[], int n) {
    for (int i = 0; i < n; i++) cout << (i ? " " : "") << a[i];
    cout << '\n';
}

int main() {
    int a[] = {1, 2, 3, 4, 5, 6, 7};
    int n = sizeof a / sizeof a[0];
    cout << "before        : "; print(a, n);
    rotateLeft(a, n, 3);
    cout << "rotated by 3  : "; print(a, n);
    rotateLeft(a, n, 11);                 // 11 mod 7 = 4 tempat lagi
    cout << "rotated by 11 : "; print(a, n);
    return 0;
}
```

Kode 1.48 `SolRotate.java`: rotasi dengan tiga pembalikan, di Java.

```java
// SolRotate.java - Penyelesaian 1.7: rotasi kiri sejauh k, tiga pembalikan
public class SolRotate {
    static void reverseRange(int[] a, int lo, int hi) {   // membalik a[lo..hi]
        while (lo < hi) {
            int t = a[lo];
            a[lo] = a[hi];
            a[hi] = t;
            lo++;
            hi--;
        }
    }

    static void rotateLeft(int[] a, int k) {
        int n = a.length;
        k %= n;
        reverseRange(a, 0, k - 1);
        reverseRange(a, k, n - 1);
        reverseRange(a, 0, n - 1);
    }

    static String show(int[] a) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < a.length; i++) sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.toString();
    }

    public static void main(String[] args) {
        int[] a = {1, 2, 3, 4, 5, 6, 7};
        System.out.println("before        : " + show(a));
        rotateLeft(a, 3);
        System.out.println("rotated by 3  : " + show(a));
        rotateLeft(a, 11);                    // 11 mod 7 = 4 tempat lagi
        System.out.println("rotated by 11 : " + show(a));
    }
}
```

Kode 1.49 `sol_rotate.py`: rotasi dengan tiga pembalikan, di Python.

```python
# sol_rotate.py - Penyelesaian 1.7: rotasi kiri sejauh k dengan tiga pembalikan
def reverse_range(a, lo, hi):            # membalik a[lo..hi]
    while lo < hi:
        a[lo], a[hi] = a[hi], a[lo]
        lo += 1
        hi -= 1


def rotate_left(a, k):
    n = len(a)
    k %= n
    reverse_range(a, 0, k - 1)
    reverse_range(a, k, n - 1)
    reverse_range(a, 0, n - 1)


a = [1, 2, 3, 4, 5, 6, 7]
print("before        :", *a)
rotate_left(a, 3)
print("rotated by 3  :", *a)
rotate_left(a, 11)                       # 11 mod 7 = 4 tempat lagi
print("rotated by 11 :", *a)
```

```
before        : 1 2 3 4 5 6 7
rotated by 3  : 4 5 6 7 1 2 3
rotated by 11 : 1 2 3 4 5 6 7
```

Penyelesaian 1.8. Transpos T dari tabel M berukuran $R \times C$ adalah tabel berukuran $C \times R$ dengan $T[c][r] = M[r][c]$: baris r dari M menjadi kolom r dari T. Dua perulangan bersarang atas M mengisi T. **Kode 1.50**, **Kode 1.51**, dan **Kode 1.52** mencetak tabel yang sama: setiap baris kini satu minggu, dan setiap kolom satu rak.

Kode 1.50 `sol_transpose.cpp`: mentranspos tabel peminjaman.

```cpp
// sol_transpose.cpp - Penyelesaian 1.8: transpos matriks 3 x 4 menjadi 4 x 3
#include <iomanip>
#include <iostream>
using namespace std;

const int R = 3, C = 4;

int main() {
    int m[R][C] = {
        {31, 28, 35, 40},
        {12, 15, 9, 14},
        {20, 22, 18, 25},
    };
    int t[C][R];
    for (int r = 0; r < R; r++) {
        for (int c = 0; c < C; c++) {
            t[c][r] = m[r][c];
        }
    }
    cout << "transpose (" << C << " x " << R << "):\n";
    for (int r = 0; r < C; r++) {
        for (int c = 0; c < R; c++) cout << setw(4) << t[r][c];
        cout << '\n';
    }
    return 0;
}
```

Kode 1.51 `SolTranspose.java`: mentranspos tabel peminjaman, di Java.

```java
// SolTranspose.java - Penyelesaian 1.8: transpos matriks 3 x 4 menjadi 4 x 3
public class SolTranspose {
    public static void main(String[] args) {
        int[][] m = {
            {31, 28, 35, 40},
            {12, 15, 9, 14},
            {20, 22, 18, 25},
        };
        int rows = m.length, cols = m[0].length;
        int[][] t = new int[cols][rows];
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++) {
                t[c][r] = m[r][c];
            }
        }
        System.out.println("transpose (" + cols + " x " + rows + "):");
        for (int[] row : t) {
            for (int x : row) System.out.printf("%4d", x);
            System.out.println();
        }
    }
}
```

Kode 1.52 `sol_transpose.py`: mentranspos tabel peminjaman, di Python.

```python
# sol_transpose.py - Penyelesaian 1.8: transpos matriks 3 x 4 menjadi matriks 4 x 3
m = [
    [31, 28, 35, 40],
    [12, 15, 9, 14],
    [20, 22, 18, 25],
]
rows, cols = len(m), len(m[0])
t = [[0] * rows for _ in range(cols)]
for r in range(rows):
    for c in range(cols):
        t[c][r] = m[r][c]

print(f"transpose ({cols} x {rows}):")
for row in t:
    print("".join(f"{x:>4}" for x in row))
```

```
transpose (4 x 3):
  31  12  20
  28  15  22
  35   9  18
  40  14  25
```

Penyelesaian 1.9. (a) Menurut Persamaan (1.6): $1000 - 0 = 1000$, $1000 - 250 = 750$, dan $1000 - 1000 = 0$. (b) Penyisipan ke-j, $j = 0,1,\ldots,999$, dilakukan pada array berisi j elemen di indeks 0 dan menggeser kesemua j elemen itu, sehingga totalnya $0 + 1 + \cdots + 999 = \frac{999 \cdot 1000}{2} =$ 499,500 geseran. (c) Nol geseran: setiap penyisipan di ujung tidak menggeser apa pun. (d) Untuk n penyisipan, total pertama adalah $\frac{n(n-1)}{2}$, yaitu $O(n^2)$, dan yang kedua 0 geseran ditambah n operasi tulis, yaitu $O(n)$. Memilih ujung alih-alih depan mengubah setengah juta geseran menjadi tidak ada sama sekali.

Penyelesaian 1.10. Kapasitasnya berjalan $0 \to 1 \to 2 \to 4 \to \cdots \to 512 \to 1024$. (a) Itu 11 kali tumbuh. (b) Setiap kali tumbuh, elemen yang ada disalin, 0,1,2,4, ...,512, dengan total $1 + 2 + 4 + \cdots + 512 = 1023$ salinan. (c) 1024. (d) Persamaan (1.5) dengan $2^k = 512 < 1000$ memberikan $2^{k+1} - 1 = 1023 < 2 \cdot 1000$, sesuai temuan: rata-rata sekitar satu salinan per `push_back`.

Penyelesaian 1.11. `[[0] * 3] * 2` adalah list berisi dua referensi ke *satu* baris, sehingga mengubah `grid[1][2]` mengubah "kedua" baris, dan program mencetak `[[0, 0, 7], [0, 0, 7]]`. Perbaikannya membuat dua baris terpisah: `grid = [[0] * 3 for _ in range(2)]`, yang sesudahnya mencetak `[[0, 0, 0], [0, 0, 7]]` (**Kode 1.31**).

Penyelesaian 1.12. Ada tiga kesalahan. Tidak satu pun merupakan *galat* kompilasi: potongan itu lolos kompilasi. Namun bila dikompilasi dengan GCC 13 dan `-O2 -Wall -Wextra`, ia memancing empat *peringatan*, `iteration 4 invokes undefined behavior`, `array subscript 5 is above array bounds of 'int [5]'`, `'squares' is used uninitialized`, dan `'total' is used uninitialized`, satu alasan lagi untuk selalu mengompilasi dengan peringatan menyala dan membacanya. (1) Perulangan pertama berjalan $i = 1$ sampai 5: ia tidak pernah mengisi `squares[0]` dan menulis `squares[5]`, melewati ujung, yang merupakan perilaku tak terdefinisi. Semestinya `for (int i = 0; i < 5; i++) squares[i] = (i + 1) * (i + 1);`, bila yang diinginkan kuadrat 1,4, ...,25. (2) `total` tidak diberi nilai awal, sehingga penjumlahan bermula dari apa pun yang ada di *stack*; tulislah `int total = 0;`. (3) `total / 5` adalah pembagian bilangan bulat, dan tidak ada kompilator yang memperingatkannya; untuk kuadrat 1 sampai 25 rata-ratanya kebetulan tepat $55/5 = 11$, tetapi pada umumnya tulislah `static_cast<double>(total) / 5`.

Penyelesaian 1.13. (a) Data global: ia dideklarasikan di luar semua fungsi dan hidup sepanjang program berjalan. (b) *Stack*, di *frame* milik `main`. (c) *Heap*. (d) *Stack*: *pointer*-nya adalah variabel lokal `main`; hanya `int` yang ditunjuknya yang berada di *heap*. (e) *Heap*: objek `vector` itu sendiri (ketiga field-nya) berada di *frame* `main`, tetapi bloknya yang berisi 100 elemen berada di *heap*, dan itulah sebabnya ia dapat tumbuh. (f) *Heap*: setiap array Java adalah objek. (g) *Heap*: setiap nilai Python adalah objek, dan `x` adalah nama yang merujuk kepadanya.

Penyelesaian 1.14. `id` menempati offset 0 sampai 5. `int` harus bermula di kelipatan 4, sehingga offset 6 dan 7 menjadi *padding* dan `yearPublished` menempati 8 sampai 11. `title` bermula di 12 dan berakhir di 111; nama pengarang menempati 112 sampai 171; `category` 172 sampai 201. Field-field itu berakhir di 202, tetapi ukuran struct harus kelipatan 4, penyelarasan `int`-nya, agar dalam sebuah array `int` milik setiap elemen tetap selaras; dua byte *padding* lagi memberikan `sizeof = 204`, empat lebih banyak daripada 200 byte field-field-nya. **Kode 1.53** menegaskan setiap angka itu. Java dan Python tidak membiarkan program melihat tata letak objek-objeknya, sehingga soal ini hanya mempunyai program C++.

Kode 1.53 `sol_padding.cpp`: offset dan ukuran `Book` yang mempunyai field `int`.

```cpp
// sol_padding.cpp - Penyelesaian 1.14: field int membawa padding penyelarasan
#include <cstddef>
#include <iostream>
using namespace std;

struct BookWithYear {
    char id[6];
    int yearPublished;
    char title[100];
    char author[60];
    char category[30];
};

int main() {
    cout << "alignof(int)          = " << alignof(int) << '\n';
    cout << "offset of id          = " << offsetof(BookWithYear, id) << '\n';
    cout << "offset of year        = " << offsetof(BookWithYear, yearPublished)
         << '\n';
    cout << "offset of title       = " << offsetof(BookWithYear, title) << '\n';
    cout << "offset of author      = " << offsetof(BookWithYear, author) << '\n';
    cout << "offset of category    = " << offsetof(BookWithYear, category) << '\n';
    cout << "sum of field sizes    = " << 6 + sizeof(int) + 100 + 60 + 30 << '\n';
    cout << "sizeof(BookWithYear)  = " << sizeof(BookWithYear) << '\n';
    return 0;
}
```

```
alignof(int)          = 4
offset of id          = 0
offset of year        = 8
offset of title       = 12
offset of author      = 112
offset of category    = 172
sum of field sizes    = 200
sizeof(BookWithYear)  = 204
```

Penyelesaian 1.15. (a) Folder `C:\msys64\ucrt64\bin` belum ada di Path pengguna, atau terminalnya dibuka sebelum folder itu ditambahkan: tambahkan seperti dalam Subbab 1.7.3 lalu buka terminal baru. (b) Yang terpasang hanyalah *runtime* Java, yang menjalankan program tetapi tidak mempunyai kompilator: pasanglah JDK menurut **Tabel 1.9**. Bila JDK sudah terpasang, berarti ada *runtime* yang lebih tua di urutan lebih awal dalam PATH. (c) Pernyataan `match` baru

hadir di Python 3.10; Python yang terpasang lebih tua. Pasanglah Python 3.14 (**Tabel 1.10**) dan periksa `python3 --version`.

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 8th ed. Hoboken, NJ, USA: Pearson, 2021.

[3] TIOBE Software, "TIOBE Index for September 2026," *TIOBE Programming Community Index*. [Online]. Available: https://www.tiobe.com/tiobe-index/ (accessed Sep. 2026).

[4] TechRepublic, "TIOBE Index September 2026: Julia nears top 20," Sep. 2026. [Online]. Available: https://www.techrepublic.com/article/news-tiobe-september-2026-julia-nears-top-20/

[5] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed. Boston, MA, USA: Pearson, 2016.

[6] B. Stroustrup, *A Tour of C++*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2022.

[7] cppreference.com, "Array declaration." [Online]. Available: https://en.cppreference.com/w/cpp/language/array (accessed Sep. 2026).

[8] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, sec. 2.2.6, "Arrays and orthogonal lists."

[9] cppreference.com, "std::vector." [Online]. Available: https://en.cppreference.com/w/cpp/container/vector (accessed Sep. 2026).

[10] Oracle, "Class ArrayList," *Java SE 25 & JDK 25 API Specification*, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/ArrayList.html

[11] OpenJDK, "java/util/ArrayList.java," *JDK source code*. [Online]. Available: https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/ArrayList.java

[12] Python Software Foundation, "Objects/listobject.c: list_resize," *CPython source code*. [Online]. Available: https://github.com/python/cpython/blob/main/Objects/listobject.c

[13] Python Software Foundation, "array — Efficient arrays of numeric values," *The Python Standard Library*, Python 3.14. [Online]. Available: https://docs.python.org/3/library/array.html

[14] Microsoft, "Using GCC with MinGW," *Visual Studio Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs/cpp/config-mingw (accessed Sep. 2026).

[15] Microsoft, "Visual Studio Code updates," *Visual Studio Code Release Notes*. [Online]. Available: https://code.visualstudio.com/updates (accessed Sep. 2026).

[16] GNU Project, "GCC 16.2 Released," gcc mailing list, Aug. 7, 2026. [Online]. Available: https://gcc.gnu.org/pipermail/gcc/2026-August/248711.html

[17] LLVM Project, "LLVM 23.1.0 Released!," LLVM Discourse, Aug. 26, 2026. [Online]. Available: https://discourse.llvm.org/t/llvm-23-1-0-released/91654

[18] MSYS2 Developers, "MSYS2: Software distribution and building platform for Windows." [Online]. Available: https://www.msys2.org

[19] Inside.java, "The Arrival of Java 27!," Sep. 15, 2026. [Online]. Available: https://inside.java/2026/09/15/jdk-27-available/

[20] Oracle, "Oracle Java SE Support Roadmap." [Online]. Available: https://www.oracle.com/java/technologies/java-se-support-roadmap.html (accessed Sep. 2026).

[21] Eclipse Foundation, "Eclipse Temurin," *Adoptium*. [Online]. Available: https://adoptium.net

[22] Python Software Foundation, "Python 3.14.7," Aug. 5, 2026. [Online]. Available: https://www.python.org/downloads/latest/python3.14/

Bab 2

Pencarian

Setelah menuntaskan bab ini, Anda akan mampu menjawab pertanyaan tertua yang pernah diajukan kepada sekumpulan data — *apakah ia ada di sini, dan di mana?* — dengan enam cara berbeda, serta memilih di antaranya berdasarkan angka, bukan tebakan. Anda akan menulis pencarian sekuensial dengan dan tanpa *sentinel* lalu menghitung dengan tepat berapa biaya masing-masing; menulis pencarian biner secara iteratif dan rekursif, menyatakan invarian gelung yang membuatnya benar, dan membuktikan bahwa ia tidak pernah memerlukan lebih dari $\lfloor \lg n \rfloor + 1$ pemeriksaan; mengenali dan menghindari jebakan *off-by-one* (meleset satu) yang selama puluhan tahun merusak pencarian biner di buku teks dan di pustaka produksi; mengubah pencarian biner menjadi `lower_bound` dan `upper_bound`, yang menjawab pertanyaan yang jauh lebih berguna — "di mana salinan pertama dimulai?", "ada berapa salinan?", dan "di mana butir baru harus diletakkan?"; menaksir posisi dengan pencarian interpolasi dan menjelaskan mengapa ia cemerlang pada kunci yang tersebar merata tetapi buruk sekali pada kunci yang menumpuk; melompati larik dalam blok berukuran $\sqrt{n}$; mencari di larik dua dimensi dengan tiga cara; mengukur semuanya pada katalog nyata berisi sepuluh ribu buku dalam C++, Java, dan Python; serta memakai `std::lower_bound`, `Arrays.binarySearch`, dan modul `bisect` Python dengan mengetahui persis apa yang dikembalikan masing-masing. Sepanjang bab, contoh berjalannya adalah katalog **Pustaka Ceria**, perpustakaan kecil yang koleksinya mulai kita simpan di Bab 1.

2.1 Dari Mana Kita Melanjutkan

Bab 1 memberi kita dua hal. Yang pertama adalah perangkat kerja C++, Java, dan Python yang berjalan di komputer apa pun yang Anda miliki. Yang kedua adalah cara *berpikir* tentang data di memori: larik (*array*) adalah sebuah blok sel berukuran sama yang berjajar berdampingan, sel ke-i berada di alamat

$$address(A[i]) = base + i \cdot w, \tag{2.1}$$

dengan w lebar satu elemen dalam byte, sehingga sel mana pun dapat dicapai dalam satu langkah, betapapun besarnya i. Persamaan (2.1) adalah alasan larik itu cepat, dan juga alasan bab ini ada: setiap algoritma pencarian yang kita temui adalah strategi untuk memutuskan sel *mana* yang perlu dilihat, dengan keyakinan bahwa setiap kali melihat biayanya sama kecilnya.

Pustaka Ceria, perpustakaan kita, kini menyimpan bukunya di dalam sebuah larik. Keinginan pustakawan berikutnya adalah keinginan setiap pemilik koleksi: seorang peminjam datang dan menanyakan sebuah buku menurut kodenya, dan pustakawan perlu tahu, dengan cepat, apakah perpustakaan memilikinya dan, jika ya, di mana letaknya. Itulah **pencarian** (*searching*), operasi yang paling sering dilakukan dalam komputasi. Setiap kali Anda masuk ke sebuah aplikasi, sistem mencari akun Anda. Setiap kali kasir memindai kode batang, sistem mencari produknya. Setiap kali mesin penjawab mengutip sebuah halaman web, ada sesuatu yang lebih dahulu mencarinya di dalam indeks.

Bab ini dibangun di atas satu gagasan, yang kita nyatakan sekarang agar Anda dapat menyaksikannya berkembang: **makin banyak yang Anda ketahui tentang susunan data,**

makin sedikit bagian yang perlu Anda lihat. Jika Anda tidak tahu apa-apa, Anda harus melihat semuanya (pencarian sekuensial, Subbab 2.3). Jika Anda tahu datanya terurut, Anda dapat membuang separuhnya pada setiap kali melihat (pencarian biner, Subbab 2.5). Jika Anda tahu datanya terurut *dan* tersebar merata, sering kali tebakan pertama Anda sudah mendarat hampir tepat di atas jawabannya (pencarian interpolasi, Subbab 2.7). Setiap keuntungan dibayar dengan sebuah asumsi, dan seluruh seninya adalah mengetahui asumsi mana yang benar-benar dipenuhi oleh data Anda.

2.2 Masalah Pencarian

2.2.1 Rekaman, kunci, dan kontrak ditemukan/tidak ditemukan

Katalog perpustakaan adalah kumpulan **rekaman** (*record*), satu untuk setiap buku. Setiap rekaman memiliki beberapa **medan** (*field*): kode, judul, nama pengarang, mungkin tahun terbit. Ketika mencari, kita mencari menurut satu medan yang disebut **kunci** (*key*). Kunci yang sedang kita cari disebut **kunci pencarian** atau **target**.

> **DEFINISI. Masalah pencarian**: diberikan kumpulan n rekaman $A[0], A[1], \ldots, A[n-1]$ dan kunci pencarian x, kembalikan sebuah indeks i sedemikian sehingga kunci $A[i]$ sama dengan x, atau laporkan bahwa indeks seperti itu tidak ada. Pencarian yang menemukan kunci disebut **berhasil**; yang tidak menemukannya disebut **gagal**.

Perhatikan apa yang diminta definisi itu. Ia tidak meminta ya atau tidak; ia meminta sebuah *posisi*. Hal ini lebih penting daripada kelihatannya. Program pertama yang ditulis kebanyakan dari kita menjawab "FOUND" atau "NOT FOUND" lalu berhenti, padahal hal berikutnya yang ingin dilakukan pemanggil — mencetak judul, menandai buku sebagai dipinjam, memindahkan raknya — memerlukan posisi itu, yang sudah dibuang oleh program. Karena itu setiap fungsi di bab ini mengembalikan sebuah indeks, dan memakai -1 untuk berarti "tidak ditemukan", sebab -1 tidak pernah menjadi indeks larik yang sah. Pemanggil kemudian cukup menulis `if (i >= 0) ...` dan sudah memegang semua yang ia perlukan.

2.2.2 Cara kita mengukur biaya pencarian

Untuk membandingkan algoritma pencarian secara adil, kita memerlukan satuan kerja yang tidak bergantung pada kecepatan komputer. Satuan yang wajar adalah **pemeriksaan** (*probe*): satu kali melihat satu elemen larik, ketika kunci elemen itu dibandingkan dengan kunci pencarian. Satu pemeriksaan bisa melibatkan satu perbandingan (`==`) atau dua (`==` lalu `<`), tetapi ia tetap satu kunjungan ke satu sel, dan kunjungan itulah yang ingin kita hitung. Ketika kita juga mengukur waktu, di Subbab 2.10, kita akan melihat seberapa baik hitungan pemeriksaan meramalkan jam — dan di mana ramalannya meleset.

Untuk setiap algoritma kita ajukan tiga pertanyaan:

- **Kasus terbaik**: banyak pemeriksaan paling sedikit di antara semua masukan berukuran n.
- **Kasus terburuk**: banyak pemeriksaan paling banyak di antara semua masukan berukuran n. Inilah jaminannya: tidak ada pencarian yang akan lebih mahal dari ini.
- **Kasus rata-rata**: banyak pemeriksaan yang diharapkan bila kunci pencarian dipilih menurut suatu asumsi yang dinyatakan, biasanya "setiap posisi sama mungkinnya".

Kita akan meringkas biaya-biaya ini dengan notasi $O(\cdot)$, dibaca "O besar" atau "berorde".

> **DEFINISI.** Untuk fungsi f dan g dalam n, kita menulis $f(n) = O(g(n))$ jika ada konstanta positif c dan n_0 sedemikian sehingga $f(n) \leq c \cdot g(n)$ untuk setiap $n \geq n_0$. Dengan kata lain: mulai dari suatu ukuran, f tidak pernah melampaui suatu kelipatan tetap dari g.

Sebagai contoh, $(n+1)/2 = O(n)$, karena $(n+1)/2 \leq n$ untuk setiap $n \geq 1$ (ambil $c = 1$, $n_0 = 1$). Notasi ini sengaja mengabaikan faktor konstanta dan masukan yang kecil, lalu hanya menyimpan *laju pertumbuhannya*: $O(n)$ tumbuh sebanding dengan n, $O(\lg n)$ bertambah satu setiap kali n berlipat dua, dan $O(1)$ tidak tumbuh sama sekali. Kita menulis $\lg n$ untuk $\log_2 n$, logaritma berbasis dua, yang menghitung berapa kali n dapat dibagi dua sampai mencapai satu; $\lg 8 = 3$ karena $8 \to 4 \to 2 \to 1$ memerlukan tiga kali pembagian. Bab 10 kembali ke notasi ini dan menambahkan kerabatnya untuk batas bawah dan batas ketat.

2.2.3 Rak komputer Pustaka Ceria

Contoh berjalan bab ini adalah rak komputer Pustaka Ceria: delapan buku, masing-masing dengan kode tiga angka. **Tabel 2.1** mendaftarnya menurut urutan kode.

Tabel 2.1 Rak komputer Pustaka Ceria, terurut menurut kode. Pencarian sekuensial juga akan menjumpai buku-buku ini menurut urutan kedatangannya: 072, 025, 096, 041, 088, 034, 063, 056.

Indeks	Kode	Judul	Pengarang
0	025	The C++ Programming	James Wood
1	034	Mastering Delphi 6	Marcopolo
2	041	Professional C#	Simon Webe
3	056	Pure JavaScript v2	Michael Bolton
4	063	Advanced JSP & Servlet	David Dunn
5	072	Calculus Make it Easy	Gunner Christian
6	088	Visual Basic 2005 Express	Antonie
7	096	Artificial Life : Volume 1	Gloria Virginia

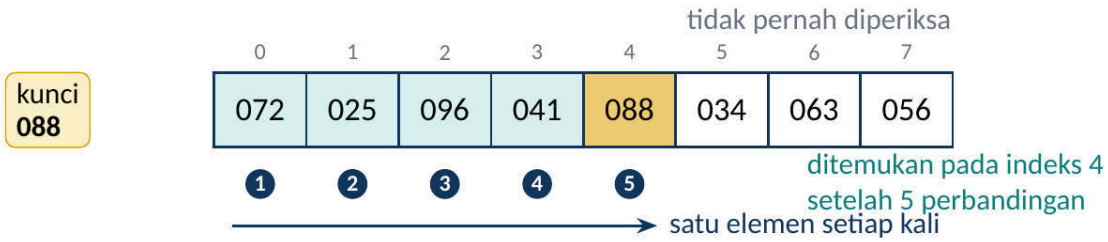
Tentu saja buku tidak datang menurut urutan kode. Buku datang menurut urutan pembeliannya, dan rak di ruang baca menyimpannya dalam **urutan kedatangan** itu: 072, 025, 096, 041, 088, 034, 063, 056. Ingatlah kedua urutan ini. Urutan kedatangan adalah apa yang dimiliki perpustakaan sebelum ada yang bekerja; urutan terurut adalah apa yang dimilikinya setelah seseorang mengurutkannya — pokok bahasan Bab 3. Pencarian mana yang mungkin dilakukan bergantung pada urutan mana yang kita punya.

2.3 Pencarian Sekuensial

2.3.1 Gagasannya

Andaikan Anda diberi sekotak delapan kartu katalog tanpa urutan tertentu dan ditanya apakah kode 088 ada di antaranya. Anda tidak punya pilihan: Anda mengambil kartu pertama dan melihatnya, lalu kartu kedua, lalu ketiga, dan berhenti ketika menemukan 088 atau ketika kotaknya kosong. Itulah **pencarian sekuensial** (*sequential search*), disebut juga **pencarian linear**. Ia tidak mengasumsikan apa pun tentang urutan data, dan justru itulah kekuatannya — ia bekerja pada larik apa pun, terurut atau tidak — sekaligus kelemahannya, sebab algoritma yang tidak mengasumsikan apa-apa tidak dapat melewatkan apa-apa.

Gambar 2.1 memperlihatkan pencarian sekuensial untuk kode 088 pada rak dalam urutan kedatangan. Empat kartu pertama bukan 088; kartu kelima adalah 088.



Gambar 2.1 Pencarian sekuensial untuk kode 088 pada rak dalam urutan kedatangan. Lingkaran bernomor memberi urutan pemeriksaan; pencarian berhenti pada pemeriksaan kelima, dan tiga sel terakhir tidak pernah diperiksa.

Pseudokode 2.1 menuliskan prosedurnya. Kita memakai gaya pseudokode Cormen dkk. [1]: baris bernomor, `←` untuk penugasan, dan indentasi untuk badan gelung.

Pseudokode 2.1 `SEQUENTIAL-SEARCH`: periksa elemen dari depan dan berhenti pada kecocokan pertama.

```
SEQUENTIAL-SEARCH(A, n, x):
 1.  for i ← 0 to n − 1
 2.      if A[i] = x
 3.          return i                  // ditemukan: berhenti pada kecocokan pertama
 4.  return −1                         // semua elemen diperiksa, tidak ada yang cocok
```

2.3.2 Dari penanda ke indeks

Versi pertama pencarian sekuensial yang ditulis kebanyakan orang memakai sebuah **penanda** (*flag*): peubah yang bernilai awal 0 dan diubah menjadi 1 ketika kecocokan terlihat. Gelungnya selalu berjalan sampai ujung larik, dan sesudah gelung program mencetak FOUND jika penandanya 1 dan NOT FOUND jika tidak. Versi ini benar, tetapi punya dua cacat. Ia terus mencari setelah menemukan kunci, sehingga selalu melakukan n perbandingan; dan ia lupa *di mana* kunci ditemukan, sehingga pemanggil tidak dapat memakai jawabannya.

Kedua cacat itu disembuhkan oleh perubahan yang sama: kembali dari dalam gelung saat kunci ditemukan, dan kembalikan indeksnya. **Pseudokode 2.1** sudah melakukannya. Namun, ada satu situasi ketika gelung *harus* berjalan sampai ujung, dan situasi ini perlu dikenali: ketika pertanyaannya bukan “apakah ia ada?” melainkan “berapa kali ia muncul?” atau “di mana saja salinannya?”. Log peminjaman yang mencatat kode setiap buku yang dipinjam hari ini berisi pengulangan, dan menghitung berapa kali buku 041 keluar mengharuskan kita melihat setiap entri. Aturannya: berhentilah lebih awal untuk keberadaan, jangan pernah berhenti lebih awal untuk penghitungan.

2.3.3 Programnya

Kode 2.1 adalah program C++ lengkapnya. Program ini menjalankan versi penanda dan versi berhenti-awal pada larik $[8,10,6,-2,11,7,1,100]$, menghitung kemunculan 041 dalam log peminjaman, lalu mencari di rak **Tabel 2.1** — dalam urutan kedatangan — menurut kode dan menurut judul. Setiap fungsi menghitung perbandingannya, sehingga keluaran memperlihatkan biaya, bukan hanya jawabannya.

Kode 2.1 `sequential_search.cpp`: pencarian sekuensial dengan penanda, dengan berhenti awal, untuk menghitung, dan pada rekaman.

```cpp
// Pencarian sekuensial: bilangan, log peminjaman, dan rak Pustaka Ceria.
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

struct Book {
    int code;
    string title;
    string writer;
};

string plural(int count, const string& word) {
    return to_string(count) + " " + word + (count == 1 ? "" : "s");
}

// Versi pertama: periksa setiap elemen dan ingat hanya ya atau tidak.
int searchFlag(const vector<int>& data, int key, int& comparisons) {
    int flag = 0;
    comparisons = 0;
    for (size_t i = 0; i < data.size(); i++) {
        comparisons++;
        if (data[i] == key) flag = 1;
    }
    return flag;
}

// Berhenti pada kecocokan pertama; kembalikan indeksnya, atau -1.
int sequentialSearch(const vector<int>& data, int key, int& comparisons) {
    comparisons = 0;
    for (size_t i = 0; i < data.size(); i++) {
        comparisons++;
        if (data[i] == key) return (int)i;
    }
    return -1;
}

// Semua posisi yang memuat kunci: gelung tak boleh berhenti awal.
vector<int> countOccurrences(const vector<int>& data, int key) {
    vector<int> where;
    for (size_t i = 0; i < data.size(); i++)
        if (data[i] == key) where.push_back((int)i);
    return where;
}
```

Kode 2.1 (Lanjutan) *sequential_search.cpp: pencarian sekuensial dengan penanda, dengan berhenti awal, untuk menghitung, dan pada rekaman.*

```
int findByCode(const vector<Book>& shelf, int code, int& comparisons) {
    comparisons = 0;
    for (size_t i = 0; i < shelf.size(); i++) {
        comparisons++;
        if (shelf[i].code == code) return (int)i;
    }
    return -1;
}

int findByTitle(const vector<Book>& shelf, const string& title, int& comparisons) {
    comparisons = 0;
    for (size_t i = 0; i < shelf.size(); i++) {
        comparisons++;
        if (shelf[i].title == title) return (int)i;
    }
    return -1;
}

int main() {
    vector<int> data = {8, 10, 6, -2, 11, 7, 1, 100};
    printf("data:");
    for (int x : data) printf(" %d", x);
    printf("\n");
    int c;
    for (int key : {6, 5}) {
        int flag = searchFlag(data, key, c);
        printf("flag version, key %d: %s after %s\n", key,
               flag == 1 ? "FOUND" : "NOT FOUND", plural(c, "comparison").c_str());
    }
    for (int key : {8, 6, 100, 5}) {
        int i = sequentialSearch(data, key, c);
        if (i >= 0)
            printf("early exit,   key %d: found at index %d after %s\n", key, i,
                   plural(c, "comparison").c_str());
        else
            printf("early exit,   key %d: not found after %s\n", key,
                   plural(c, "comparison").c_str());
    }
```

Kode 2.1 (Lanjutan) *sequential_search.cpp: pencarian sekuensial dengan penanda, dengan berhenti awal, untuk menghitung, dan pada rekaman.*

```
    vector<int> loans = {41, 88, 25, 41, 63, 41, 96, 88};
    vector<int> where = countOccurrences(loans, 41);
    printf("loan log:");
    for (int x : loans) printf(" %03d", x);
    printf("\nbook 041 was borrowed %zu times, at positions", where.size());
    for (int p : where) printf(" %d", p);
    printf("\n");

    vector<Book> shelf = {{72, "Calculus Make it Easy", "Gunner Christian"},
                          {25, "The C++ Programming", "James Wood"},
                          {96, "Artificial Life : Volume 1", "Gloria Virginia"},
                          {41, "Professional C#", "Simon Webe"},
                          {88, "Visual Basic 2005 Express", "Antonie"},
                          {34, "Mastering Delphi 6", "Marcopolo"},
                          {63, "Advanced JSP & Servlet", "David Dunn"},
                          {56, "Pure JavaScript v2", "Michael Bolton"}};
    for (int code : {88, 60}) {
        int i = findByCode(shelf, code, c);
        if (i >= 0)
            printf("code %03d: index %d, %s by %s (%s)\n", code, i,
                   shelf[i].title.c_str(), shelf[i].writer.c_str(),
                   plural(c, "comparison").c_str());
        else
            printf("code %03d: not on the shelf (%s)\n", code,
                   plural(c, "comparison").c_str());
    }
    int i = findByTitle(shelf, "Professional C#", c);
    printf("title \"Professional C#\": index %d, code %03d (%s)\n", i,
           shelf[i].code, plural(c, "comparison").c_str());
    return 0;
}
```

Kode 2.2 adalah program yang sama dalam Java, dan **Kode 2.3** dalam Python. Di Java, sebuah `record` memberi kita tipe `Book` dalam satu baris; di Python, sebuah kelas kecil melakukan tugas yang sama. Setiap fungsi pencarian mengembalikan sepasang nilai — indeks dan banyak perbandingan — sebagai `int[]` dua elemen di Java dan sebagai *tuple* di Python.

Kode 2.2 `SequentialSearch.java`: pencarian yang sama dalam Java.

```java
// Pencarian sekuensial: bilangan, log peminjaman, dan rak Pustaka Ceria.
import java.util.ArrayList;
import java.util.List;

public class SequentialSearch {
    record Book(int code, String title, String writer) {}

    static String plural(int count, String word) {
        return count + " " + word + (count == 1 ? "" : "s");
    }

    // Versi pertama: periksa setiap elemen dan ingat hanya ya atau tidak.
    // Mengembalikan {flag, perbandingan}.
    static int[] searchFlag(int[] data, int key) {
        int flag = 0, comparisons = 0;
        for (int x : data) {
            comparisons++;
            if (x == key) flag = 1;
        }
        return new int[] {flag, comparisons};
    }

    // Berhenti pada kecocokan pertama. Hasil: {indeks atau -1, perbandingan}.
    static int[] sequentialSearch(int[] data, int key) {
        int comparisons = 0;
        for (int i = 0; i < data.length; i++) {
            comparisons++;
            if (data[i] == key) return new int[] {i, comparisons};
        }
        return new int[] {-1, comparisons};
    }

    // Semua posisi yang memuat kunci: gelung tak boleh berhenti awal.
    static List<Integer> countOccurrences(int[] data, int key) {
        List<Integer> where = new ArrayList<>();
        for (int i = 0; i < data.length; i++)
            if (data[i] == key) where.add(i);
        return where;
    }
```

Kode 2.2 (Lanjutan) *SequentialSearch.java: pencarian yang sama dalam Java.*

```
    static int[] findByCode(Book[] shelf, int code) {
        int comparisons = 0;
        for (int i = 0; i < shelf.length; i++) {
            comparisons++;
            if (shelf[i].code() == code) return new int[] {i, comparisons};
        }
        return new int[] {-1, comparisons};
    }

    static int[] findByTitle(Book[] shelf, String title) {
        int comparisons = 0;
        for (int i = 0; i < shelf.length; i++) {
            comparisons++;
            if (shelf[i].title().equals(title)) return new int[] {i, comparisons};
        }
        return new int[] {-1, comparisons};
    }

    public static void main(String[] args) {
        int[] data = {8, 10, 6, -2, 11, 7, 1, 100};
        StringBuilder line = new StringBuilder("data:");
        for (int x : data) line.append(" ").append(x);
        System.out.println(line);
        for (int key : new int[] {6, 5}) {
            int[] r = searchFlag(data, key);
            System.out.printf("flag version, key %d: %s after %s%n", key,
                    r[0] == 1 ? "FOUND" : "NOT FOUND", plural(r[1], "comparison"));
        }
        for (int key : new int[] {8, 6, 100, 5}) {
            int[] r = sequentialSearch(data, key);
            if (r[0] >= 0)
                System.out.printf(
                        "early exit,   key %d: found at index %d after %s%n", key,
                        r[0], plural(r[1], "comparison"));
            else
                System.out.printf("early exit,   key %d: not found after %s%n", key,
                        plural(r[1], "comparison"));
        }
```

Kode 2.2 (Lanjutan) *SequentialSearch.java: pencarian yang sama dalam Java.*

```
        int[] loans = {41, 88, 25, 41, 63, 41, 96, 88};
        List<Integer> where = countOccurrences(loans, 41);
        line = new StringBuilder("loan log:");
        for (int x : loans) line.append(String.format(" %03d", x));
        System.out.println(line);
        line = new StringBuilder(
                "book 041 was borrowed " + where.size() + " times, at positions");
        for (int p : where) line.append(" ").append(p);
        System.out.println(line);

        Book[] shelf = {new Book(72, "Calculus Make it Easy", "Gunner Christian"),
                new Book(25, "The C++ Programming", "James Wood"),
                new Book(96, "Artificial Life : Volume 1", "Gloria Virginia"),
                new Book(41, "Professional C#", "Simon Webe"),
                new Book(88, "Visual Basic 2005 Express", "Antonie"),
                new Book(34, "Mastering Delphi 6", "Marcopolo"),
                new Book(63, "Advanced JSP & Servlet", "David Dunn"),
                new Book(56, "Pure JavaScript v2", "Michael Bolton")};
        for (int code : new int[] {88, 60}) {
            int[] r = findByCode(shelf, code);
            if (r[0] >= 0)
                System.out.printf("code %03d: index %d, %s by %s (%s)%n", code,
                        r[0], shelf[r[0]].title(), shelf[r[0]].writer(),
                        plural(r[1], "comparison"));
            else
                System.out.printf("code %03d: not on the shelf (%s)%n", code,
                        plural(r[1], "comparison"));
        }
        int[] r = findByTitle(shelf, "Professional C#");
        System.out.printf("title \"Professional C#\": index %d, code %03d (%s)%n",
                r[0], shelf[r[0]].code(), plural(r[1], "comparison"));
    }
}
```

Kode 2.3 `sequential_search.py`: pencarian yang sama dalam Python.

```
"""Pencarian sekuensial: bilangan biasa, log peminjaman, dan rak Pustaka Ceria."""


class Book:
    def __init__(self, code, title, writer):
        self.code = code
        self.title = title
        self.writer = writer


def plural(count, word):
    return "%d %s%s" % (count, word, "" if count == 1 else "s")


def search_flag(data, key):
    """Versi pertama: periksa setiap elemen dan ingat hanya ya atau tidak."""
    flag = 0
    comparisons = 0
    for x in data:
        comparisons += 1
        if x == key:
            flag = 1
    return flag, comparisons


def sequential_search(data, key):
    """Hasil (indeks, perbandingan) di kecocokan pertama; indeks -1: tak ada."""
    comparisons = 0
    for i in range(len(data)):
        comparisons += 1
        if data[i] == key:
            return i, comparisons
    return -1, comparisons


def count_occurrences(data, key):
    """Semua posisi yang memuat kunci: gelung tak boleh berhenti awal."""
    return [i for i in range(len(data)) if data[i] == key]
```

Kode 2.3 (Lanjutan) *sequential_search.py: pencarian yang sama dalam Python.*

```
def find_by_code(shelf, code):
    comparisons = 0
    for i in range(len(shelf)):
        comparisons += 1
        if shelf[i].code == code:
            return i, comparisons
    return -1, comparisons


def find_by_title(shelf, title):
    comparisons = 0
    for i in range(len(shelf)):
        comparisons += 1
        if shelf[i].title == title:
            return i, comparisons
    return -1, comparisons


def main():
    data = [8, 10, 6, -2, 11, 7, 1, 100]
    print("data:", " ".join(str(x) for x in data))
    for key in (6, 5):
        flag, c = search_flag(data, key)
        print(
            "flag version, key %d: %s after %s"
            % (key, "FOUND" if flag == 1 else "NOT FOUND", plural(c, "comparison"))
        )
    for key in (8, 6, 100, 5):
        i, c = sequential_search(data, key)
        if i >= 0:
            print(
                "early exit,   key %d: found at index %d after %s"
                % (key, i, plural(c, "comparison"))
            )
        else:
            print(
                "early exit,   key %d: not found after %s"
                % (key, plural(c, "comparison"))
            )
```

Kode 2.3 (Lanjutan) *sequential_search.py: pencarian yang sama dalam Python.*

```
    loans = [41, 88, 25, 41, 63, 41, 96, 88]
    where = count_occurrences(loans, 41)
    print("loan log:", " ".join("%03d" % x for x in loans))
    print(
        "book 041 was borrowed %d times, at positions %s"
        % (len(where), " ".join(str(p) for p in where))
    )

    shelf = [
        Book(72, "Calculus Make it Easy", "Gunner Christian"),
        Book(25, "The C++ Programming", "James Wood"),
        Book(96, "Artificial Life : Volume 1", "Gloria Virginia"),
        Book(41, "Professional C#", "Simon Webe"),
        Book(88, "Visual Basic 2005 Express", "Antonie"),
        Book(34, "Mastering Delphi 6", "Marcopolo"),
        Book(63, "Advanced JSP & Servlet", "David Dunn"),
        Book(56, "Pure JavaScript v2", "Michael Bolton"),
    ]
    for code in (88, 60):
        i, c = find_by_code(shelf, code)
        if i >= 0:
            print(
                "code %03d: index %d, %s by %s (%s)"
                % (
                    code,
                    i,
                    shelf[i].title,
                    shelf[i].writer,
                    plural(c, "comparison"),
                )
            )
        else:
            print(
                "code %03d: not on the shelf (%s)" % (code, plural(c, "comparison"))
            )
    i, c = find_by_title(shelf, "Professional C#")
    print(
        'title "Professional C#": index %d, code %03d (%s)'
        % (i, shelf[i].code, plural(c, "comparison"))
    )


if __name__ == "__main__":
    main()
```

Ketiga program mencetak baris yang persis sama:

```
data: 8 10 6 -2 11 7 1 100
flag version, key 6: FOUND after 8 comparisons
flag version, key 5: NOT FOUND after 8 comparisons
early exit,   key 8: found at index 0 after 1 comparison
early exit,   key 6: found at index 2 after 3 comparisons
early exit,   key 100: found at index 7 after 8 comparisons
early exit,   key 5: not found after 8 comparisons
loan log: 041 088 025 041 063 041 096 088
book 041 was borrowed 3 times, at positions 0 3 5
code 088: index 4, Visual Basic 2005 Express by Antonie (5 comparisons)
code 060: not on the shelf (8 comparisons)
title "Professional C#": index 3, code 041 (4 comparisons)
```

Bacalah keluaran itu sambil memegang definisinya. Versi penanda melakukan 8 perbandingan, entah kuncinya ditemukan atau tidak. Versi berhenti-awal menemukan 8 dalam 1 perbandingan (kasus terbaik), 6 dalam 3, dan 100 dalam 8 (kasus terburuk untuk pencarian berhasil), dan

memerlukan kedelapannya untuk memastikan bahwa 5 tidak ada. Mencari 088 di rak memakan 5 perbandingan, persis seperti yang diramalkan **Gambar 2.1**.

2.3.4 Berapa biaya pencarian sekuensial?

Kasus terbaik. Kunci berada di $A[0]$: satu pemeriksaan. Kasus terbaiknya $O(1)$.

Kasus terburuk. Kunci berada di $A[n-1]$, atau tidak ada. Dalam kedua keadaan itu setiap elemen diperiksa: n pemeriksaan. Kasus terburuknya $O(n)$, dan pencarian yang gagal *selalu* memakan tepat n.

Kasus rata-rata, berhasil. Anggaplah kunci ada dan sama mungkinnya berada di salah satu dari n posisi. Jika kunci di posisi i, pencarian melakukan $i+1$ pemeriksaan. Dirata-ratakan atas n posisi,

$$C_{\text{succ}}(n) = \frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}\cdot\frac{n(n+1)}{2} = \frac{n+1}{2}. \tag{2.2}$$

Langkah dari jumlahan ke $n(n+1)/2$ adalah rumus untuk $1+2+\cdots+n$, yang dapat Anda periksa dengan menuliskan jumlahan itu maju dan mundur lalu menjumlahkannya: setiap dari n kolom berjumlah $n+1$, jadi dua kali jumlahan itu sama dengan $n(n+1)$. Untuk rak delapan buku kita, Persamaan (2.2) memberi rata-rata 4.5 pemeriksaan; untuk katalog sepuluh ribu buku, 5,000.5.

Kasus rata-rata, kunci mungkin tidak ada. Dalam pemakaian nyata, kunci tidak selalu ada. Misalkan p adalah peluang kunci ada. Dengan peluang p biaya pencarian rata-rata $(n+1)/2$, dan dengan peluang $1-p$ biayanya n. Dengan membobot keduanya,

$$C(n,p) = p\cdot\frac{n+1}{2} + (1-p)\cdot n. \tag{2.3}$$

Pada $p=1$ rumus ini menjadi Persamaan (2.2); pada $p=0$ menjadi n; dan pada $p=\frac{1}{2}$ kira-kira $\frac{3}{4}n$. Dalam setiap keadaan ia tumbuh sebanding langsung dengan n: gandakan katalog dan Anda menggandakan rata-rata pencarian. Itulah arti $O(n)$ dalam praktik, dan itulah sebabnya pencarian sekuensial, dengan segala kesederhanaannya, adalah algoritma yang kita hindari pada koleksi besar.

> **WAWASAN KUNCI:** Pencarian sekuensial adalah satu-satunya pencarian di bab ini yang bekerja pada data **tidak terurut**, dan pada data yang hanya dapat dibaca dari depan ke belakang — berkas, aliran data, *linked list* (Bab 7). Bila datanya kecil, atau hanya akan dicari sekali, atau tidak dapat diurutkan, ia adalah alat yang tepat. Bila datanya besar dan sering dicari, ia adalah alat yang perlu diganti.

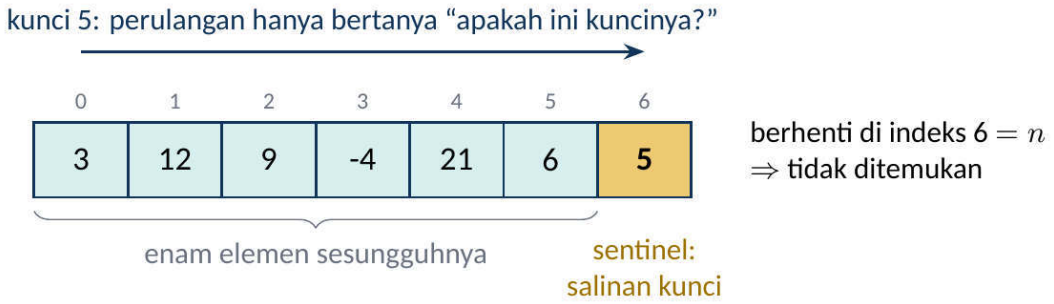
2.4 Pencarian Sekuensial dengan Sentinel

2.4.1 Dua uji padahal satu cukup

Perhatikan baik-baik gelung **Pseudokode 2.1**. Pada setiap langkah ia mengajukan *dua* pertanyaan: "apakah i masih kurang dari n?" (tersembunyi dalam `for`) dan "apakah $A[i]$ kuncinya?". Pertanyaan pertama ada semata-mata untuk mencegah gelung berlari melewati ujung

larik ketika kunci tidak ada. Pada pencarian yang berhasil, jawabannya selalu "ya", dan uji itu murni beban tambahan.

Trik **sentinel** menyingkirkannya. Beri larik satu sel cadangan di indeks n, dan sebelum mencari, letakkan salinan kunci di sana. Sekarang gelung *dijamin* menemukan kunci — paling lambat di sel cadangan itu — sehingga ia tidak perlu lagi mengawasi batas. Ketika gelung berhenti di posisi i, satu uji sesudahnya menentukan jawaban: jika $i < n$, kunci memang ada; jika $i = n$, gelung hanya menemukan salinannya sendiri, dan kuncinya tidak ada. **Gambar 2.2** memperlihatkan gagasan ini pada larik [3,12,9, −4,21,6] dengan kunci 5.



Gambar 2.2 Pencarian sekuensial dengan sentinel. Kunci 5 disalin ke sel cadangan di indeks 6, sehingga gelung pasti berhenti paling lambat di sana; berhenti di indeks $6 = n$ berarti kunci tidak ada di antara elemen sesungguhnya.

Pseudokode 2.2 `SENTINEL-SEARCH`: tanam salinan kunci sesudah elemen terakhir agar gelung hanya memerlukan satu uji per langkah.

```
SENTINEL-SEARCH(A, n, x):          // A punya sel cadangan A[n]
 1.  saved ← A[n]
 2.  A[n] ← x                      // tanam sentinel
 3.  i ← 0
 4.  while A[i] ≠ x                // satu uji per langkah: tanpa uji batas
 5.      i ← i + 1
 6.  A[n] ← saved                  // kembalikan sel cadangan seperti semula
 7.  if i < n
 8.      return i
 9.  return -1
```

2.4.2 Menghitung uji

Kode 2.4 membuat penghematan itu terlihat dengan menghitung setiap uji yang dilakukan kedua gelung. Pencarian biasa ditulis dengan kedua ujinya dijabarkan, sehingga masing-masing dapat dihitung; **Kode 2.5** dan **Kode 2.6** adalah versi Java dan Python-nya.

Kode 2.4 `sentinel_search.cpp`: gelung biasa dan gelung sentinel, dengan setiap uji gelung dihitung.

```cpp
// Pencarian sekuensial dengan dan tanpa sentinel; setiap uji gelung dihitung.
#include <cstdio>
#include <string>
using namespace std;

// Dua uji per langkah: "i < n" dan "data[i] != key".
int plainSearch(const int data[], int n, int key, int& boundTests, int& keyTests) {
    boundTests = keyTests = 0;
    int i = 0;
    while (true) {
        boundTests++;
        if (!(i < n)) break;
        keyTests++;
        if (!(data[i] != key)) break;
        i++;
    }
    return i < n ? i : -1;
}

// data punya sel cadangan di indeks n; kunci ditanam di sana dahulu.
int sentinelSearch(int data[], int n, int key, int& keyTests) {
    int saved = data[n];
    data[n] = key;  // sentinel: perulangan kini pasti berhenti
    keyTests = 0;
    int i = 0;
    while (true) {
        keyTests++;
        if (!(data[i] != key)) break;
        i++;
    }
    data[n] = saved;  // kembalikan larik seperti semula
    return i < n ? i : -1;
}

int main() {
    const int n = 6;
    int data[n + 1] = {3, 12, 9, -4, 21, 6, 0};  // 6 elemen asli + 1 sel cadangan
    printf("data:");
    for (int i = 0; i < n; i++) printf(" %d", data[i]);
    printf(" | spare slot at index %d\n", n);
    for (int key : {21, 5}) {
        int b, k, s;
        int i = plainSearch(data, n, key, b, k);
        int j = sentinelSearch(data, n, key, s);
        string where = i >= 0 ? "index " + to_string(i) : "not found";
        string where2 = j >= 0 ? "index " + to_string(j) : "not found";
```

Kode 2.4 (Lanjutan) *sentinel_search.cpp: gelung biasa dan gelung sentinel, dengan setiap uji gelung dihitung.*

```cpp
        printf("key %2d  plain:    %-9s %2d loop tests (%d bound + %d key)\n", key,
               where.c_str(), b + k, b, k);
        printf("        sentinel: %-9s %2d loop tests (key only)\n", where2.c_str(),
               s);
    }
    printf("spare slot after the searches: %d\n", data[n]);
    return 0;
}
```

Kode 2.5 `SentinelSearch.java`: perbandingan yang sama dalam Java.

```java
// Pencarian sekuensial dengan dan tanpa sentinel; setiap uji gelung dihitung.
public class SentinelSearch {
    // Dua uji per langkah: "i < n" dan "data[i] != key".
    // Mengembalikan {indeks atau -1, uji batas, uji kunci}.
    static int[] plainSearch(int[] data, int n, int key) {
        int boundTests = 0, keyTests = 0;
        int i = 0;
        while (true) {
            boundTests++;
            if (!(i < n)) break;
            keyTests++;
            if (!(data[i] != key)) break;
            i++;
        }
        return new int[] {i < n ? i : -1, boundTests, keyTests};
    }

    // data punya sel cadangan di indeks n; kunci ditanam di sana dahulu.
    // Mengembalikan {indeks atau -1, uji kunci}.
    static int[] sentinelSearch(int[] data, int n, int key) {
        int saved = data[n];
        // sentinel: perulangan kini pasti berhenti
        data[n] = key;
        int keyTests = 0;
        int i = 0;
        while (true) {
            keyTests++;
            if (!(data[i] != key)) break;
            i++;
        }
        data[n] = saved;  // kembalikan larik seperti semula
        return new int[] {i < n ? i : -1, keyTests};
    }

    public static void main(String[] args) {
        int n = 6;
        int[] data = {3, 12, 9, -4, 21, 6, 0};  // 6 elemen asli + 1 sel cadangan
        StringBuilder line = new StringBuilder("data:");
        for (int i = 0; i < n; i++) line.append(" ").append(data[i]);
        System.out.println(line + " | spare slot at index " + n);
        for (int key : new int[] {21, 5}) {
            int[] p = plainSearch(data, n, key);
            int[] s = sentinelSearch(data, n, key);
            String where = p[0] >= 0 ? "index " + p[0] : "not found";
            String where2 = s[0] >= 0 ? "index " + s[0] : "not found";
            System.out.printf(
```

Kode 2.5 (Lanjutan) *SentinelSearch.java: perbandingan yang sama dalam Java.*

```java
                    "key %2d  plain:    %-9s %2d loop tests (%d bound + %d key)%n",
                    key, where, p[1] + p[2], p[1], p[2]);
            System.out.printf("        sentinel: %-9s %2d loop tests (key only)%n",
                    where2, s[1]);
        }
        System.out.println("spare slot after the searches: " + data[n]);
    }
}
```

Kode 2.6 `sentinel_search.py`: perbandingan yang sama dalam Python.

```
"""Pencarian sekuensial dengan dan tanpa sentinel; setiap uji gelung dihitung."""


def plain_search(data, n, key):
    """Dua uji per langkah: 'i < n' dan 'data[i] != key'."""
    bound_tests = key_tests = 0
    i = 0
    while True:
        bound_tests += 1
        if not i < n:
            break
        key_tests += 1
        if not data[i] != key:
            break
        i += 1
    return (i if i < n else -1), bound_tests, key_tests


def sentinel_search(data, n, key):
    """data punya sel cadangan di indeks n; kunci ditanam di sana dahulu."""
    saved = data[n]
    data[n] = key  # sentinel: perulangan kini pasti berhenti
    key_tests = 0
    i = 0
    while True:
        key_tests += 1
        if not data[i] != key:
            break
        i += 1
    data[n] = saved  # kembalikan larik seperti semula
    return (i if i < n else -1), key_tests


def main():
    n = 6
    data = [3, 12, 9, -4, 21, 6, 0]  # 6 elemen asli + 1 sel cadangan
    print("data:", " ".join(str(x) for x in data[:n]), "| spare slot at index", n)
    for key in (21, 5):
        i, b, k = plain_search(data, n, key)
        j, s = sentinel_search(data, n, key)
        where = "index %d" % i if i >= 0 else "not found"
        where2 = "index %d" % j if j >= 0 else "not found"
        print(
            "key %2d  plain:    %-9s %2d loop tests (%d bound + %d key)"
            % (key, where, b + k, b, k)
        )
```

Kode 2.6 (Lanjutan) *sentinel_search.py: perbandingan yang sama dalam Python.*

```
        print("        sentinel: %-9s %2d loop tests (key only)" % (where2, s))
    print("spare slot after the searches:", data[n])


if __name__ == "__main__":
    main()
```

Ketiganya mencetak:

```
data: 3 12 9 -4 21 6 | spare slot at index 6
key 21  plain:    index 4   10 loop tests (5 bound + 5 key)
        sentinel: index 4    5 loop tests (key only)
key  5  plain:    not found 13 loop tests (7 bound + 6 key)
        sentinel: not found  7 loop tests (key only)
spare slot after the searches: 0
```

Untuk kunci 21 di indeks 4, gelung biasa melakukan 5 uji batas dan 5 uji kunci; gelung sentinel hanya melakukan 5 uji kunci. Untuk kunci 5 yang tidak ada, gelung biasa melakukan $n + 1 = 7$ uji batas (yang ketujuh adalah yang gagal) dan $n = 6$ uji kunci, seluruhnya $2n + 1 = 13$; gelung sentinel melakukan $n + 1 = 7$ uji kunci, yang terakhir pada sentinel. Secara umum gelung biasa memerlukan kira-kira $2n$ uji pada kasus terburuk dan gelung sentinel kira-kira n. Baris terakhir keluaran memastikan bahwa sel cadangan telah dikembalikan.

2.4.3 Apa yang dibeli sentinel, dan apa yang tidak

Sentinel memangkas banyaknya *uji* menjadi separuh. Ia tidak mengurangi banyaknya *pemeriksaan*: kedua gelung melihat elemen yang sama dalam urutan yang sama. Ia tidak mengubah orde pertumbuhan: keduanya $O(n)$. Dan ia punya biayanya sendiri. Larik memerlukan sel cadangan; larik harus dapat ditulisi, sehingga tidak dapat dipakai untuk mencari di tabel konstan; dan di antara saat sentinel ditanam dan saat ia dicabut, larik untuk sementara memuat nilai yang sebenarnya tidak ada, sehingga dua *thread* tidak boleh mencari larik yang sama dengan cara ini pada saat bersamaan.

Apakah separuh itu sepadan? Pada mesin tahun 1970-an, ketika Knuth menganalisis trik ini [2], sebuah perbandingan dan percabangan itu mahal dan penghematannya nyata. Pada prosesor modern jawabannya adalah hasil pengukuran, bukan hasil perdebatan, dan Subbab 2.10 melakukannya: pada katalog nyata kita berisi sepuluh ribu buku, kedua gelung berjalan sama cepat dalam batas galat pengukuran di C++ dan di Java, karena prosesor meramalkan uji batas yang selalu benar itu hampir sempurna dan mengerjakannya berdampingan dengan uji kunci. Di Python, tempat setiap uji adalah operasi yang ditafsirkan, versi sentinel sekitar 9% lebih cepat. Pelajarilah trik ini, karena ia muncul di kode pustaka dan di algoritma lain (langkah penyisipan pada *insertion sort* di Bab 3 dapat memakainya), tetapi jangan berharap ia menyelamatkan pencarian yang lambat. Hanya algoritma yang lebih baik yang dapat melakukannya.

2.5 Pencarian Biner

2.5.1 Memanfaatkan urutan

Sekarang andaikan rak itu terurut menurut kode, seperti pada **Tabel 2.1**. Carilah 088 lagi, tetapi kali ini bukalah rak di tengahnya. Buku di tengah adalah 056. Karena rak terurut, setiap buku di kiri 056 berkode lebih kecil, sehingga 088 tidak mungkin ada di sana: dengan sekali melihat, kita sudah membuang separuh rak. Lihat bagian tengah dari separuh yang tersisa, lalu ulangi. Inilah **pencarian biner** (*binary search*), algoritma terpenting di bab ini.

Prinsipnya persis seperti cara Anda mencari kata di kamus cetak: Anda tidak membaca mulai dari *abad*; Anda membuka di suatu tempat, melihat bahwa Anda terlalu depan atau terlalu belakang, lalu melanjutkan di separuh yang masih mungkin memuat kata itu.

> **DEFINISI. Pencarian biner** mencari kunci x di larik *terurut* $A[0..n-1]$ dengan memelihara interval $A[low..high]$ yang pasti memuat x jika x ada di mana pun. Pada setiap langkah ia memeriksa elemen tengah $A[mid]$ dengan $mid = \lfloor (low + high)/2 \rfloor$. Jika $A[mid] = x$, pencarian berhasil; jika $A[mid] < x$, intervalnya menjadi $A[mid + 1..high]$; jika $A[mid] > x$, intervalnya menjadi $A[low..mid - 1]$. Ketika interval kosong ($low > high$), pencarian gagal.

2.5.2 Jejak dengan tangan

Ambil larik terurut $A = [3,9,11,12,15,17,23,31,35]$, dengan $n = 9$, lalu cari $x = 17$. **Gambar 2.3** memperlihatkan ketiga langkahnya.



Gambar 2.3 Pencarian biner untuk 17 dalam [3,9,11,12,15,17,23,31,35]. Sel abu-abu sudah dikesampingkan; sel kuning adalah sel yang diperiksa. Tiga pemeriksaan menemukan kuncinya.

- **Langkah 1.** $low = 0$, $high = 8$, jadi $mid = \lfloor 8/2 \rfloor = 4$ dan $A[4] = 15$. Karena $17 > 15$, kunci hanya mungkin di kanan: $low \leftarrow 5$.
- **Langkah 2.** $low = 5$, $high = 8$, jadi $mid = \lfloor 13/2 \rfloor = 6$ dan $A[6] = 23$. Karena $17 < 23$, kunci hanya mungkin di kiri: $high \leftarrow 5$.
- **Langkah 3.** $low = high = 5$, jadi $mid = 5$ dan $A[5] = 17$. Ditemukan, di indeks 5.

Pencarian sekuensial akan memerlukan enam pemeriksaan. Sekarang carilah $x = 10$, yang tidak ada:

- $low = 0, high = 8, mid = 4, A[4] = 15 > 10$, jadi $high \leftarrow 3$.
- $low = 0, high = 3, mid = 1, A[1] = 9 < 10$, jadi $low \leftarrow 2$.
- $low = 2, high = 3, mid = 2, A[2] = 11 > 10$, jadi $high \leftarrow 1$.
- Kini $low = 2 > high = 1$: interval kosong dan 10 tidak ada.

Perhatikan di mana *low* berakhir: di 2, yang tepat merupakan indeks tempat 10 harus disisipkan agar larik tetap terurut (di antara 9 dan 11). Ini bukan kebetulan, dan Subbab 2.6 membangun di atasnya.

2.5.3 Prosedur dan invariannya

Pseudokode 2.3 adalah prosedur iteratifnya.

Pseudokode 2.3 `BINARY-SEARCH`, iteratif, pada interval tertutup $[low, high]$.

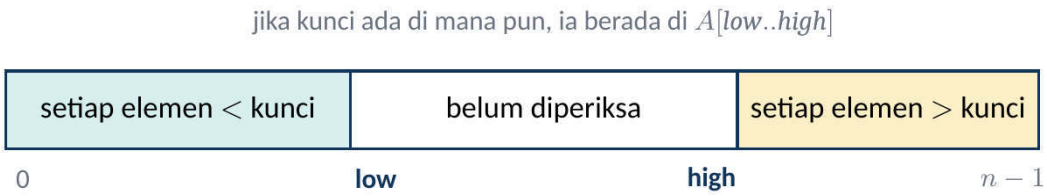
```
BINARY-SEARCH(A, n, x):          // A[0..n-1] terurut tidak menurun
 1.  low ← 0
 2.  high ← n − 1
 3.  while low ≤ high             // interval A[low..high] tidak kosong
 4.      mid ← low + ⌊(high − low)/2⌋
 5.      if A[mid] = x
 6.          return mid
 7.      else if A[mid] < x
 8.          low ← mid + 1        // x hanya mungkin di kanan mid
 9.      else
10.          high ← mid − 1       // x hanya mungkin di kiri mid
11.  return −1                    // interval kosong: x tidak ada
```

Baris 4 menghitung titik tengah sebagai $low + \lfloor (high - low)/2 \rfloor$, bukan $\lfloor (low + high)/2 \rfloor$. Secara matematis keduanya sama; di komputer tidak, dan Subbab 2.6.4 menjelaskan sebabnya.

Pencarian biner terkenal mudah dijelaskan tetapi sulit ditulis dengan benar. Knuth mencatat bahwa walaupun pencarian biner pertama kali diterbitkan pada 1946, versi pertama yang benar untuk setiap n baru muncul pada 1962 [2], dan Bentley mendapati bahwa hanya sekitar satu dari sepuluh pemrogram profesional yang mampu menulis versi yang benar dalam beberapa jam [3]. Pertahanan terhadap kesalahan semacam itu bukan kepintaran, melainkan **invarian gelung** (*loop invariant*): pernyataan yang benar setiap kali syarat gelung diuji, dan yang, bersama keadaan saat gelung berakhir, memberikan jawabannya.

> **INVARIAN.** Pada awal setiap iterasi gelung `while` di **Pseudokode 2.3**: setiap elemen $A[0..low-1]$ lebih kecil daripada x, dan setiap elemen $A[high+1..n-1]$ lebih besar daripada x. Akibatnya, jika x ada di mana pun di A, ia berada di $A[low..high]$.

Gambar 2.4 menggambarkannya: larik selalu terbagi menjadi tiga wilayah, dan pencarian hanya bekerja pada wilayah tengah.



Gambar 2.4 Invarian gelung pencarian biner. Semua yang di kiri low diketahui lebih kecil daripada kunci, semua yang di kanan $high$ diketahui lebih besar, dan kunci, jika ada, berada di wilayah tengah.

Kita membuktikan invarian ini seperti setiap invarian gelung dibuktikan, dalam tiga bagian [1].

Inisialisasi. Sebelum iterasi pertama $low = 0$ dan $high = n - 1$. Wilayah $A[0..-1]$ dan $A[n..n-1]$ kosong, dan pernyataan tentang setiap elemen dari wilayah kosong selalu benar.

Pemeliharaan. Misalkan invarian berlaku pada awal suatu iterasi dan gelung tidak kembali. Jika $A[mid] < x$, maka karena larik terurut, setiap elemen $A[low..mid]$ paling besar $A[mid]$, sehingga lebih kecil daripada x. Bersama invarian, setiap elemen $A[0..mid]$ lebih kecil daripada x, sehingga penugasan $low \leftarrow mid + 1$ menjaga paruh pertama invarian tetap benar; paruh kedua tidak tersentuh. Kasus $A[mid] > x$ adalah bayangan cerminnya: setiap elemen $A[mid..high]$ paling kecil $A[mid]$, sehingga lebih besar daripada x, dan $high \leftarrow mid - 1$ menjaga paruh kedua tetap benar.

Terminasi. Gelung berakhir dengan kembali di baris 6, dan dalam hal itu $A[mid] = x$ sehingga jawabannya benar, atau berakhir karena $low > high$. Dalam hal kedua, wilayah tengah kosong, dan menurut invarian setiap elemen lebih kecil daripada x atau lebih besar daripada x: x tidak ada, dan mengembalikan -1 adalah benar. Lebih dari itu, low kini menunjuk elemen pertama yang lebih besar daripada x (atau n), yaitu titik sisip yang kita lihat pada jejak tadi.

Gelung memang berakhir. Panjang interval, $high - low + 1$, adalah bilangan bulat tak negatif, dan setiap iterasi yang tidak kembali membuatnya mengecil secara tegas, karena mid terletak di dalam interval dan dikeluarkan dari interval berikutnya. Bilangan bulat tak negatif tidak mungkin berkurang selamanya. ■

2.5.4 Berapa banyak pemeriksaan?

Misalkan interval saat ini berisi m elemen. Pemeriksaan di mid membagi $m - 1$ elemen lainnya menjadi bagian kiri berisi $\lfloor (m-1)/2 \rfloor$ elemen dan bagian kanan berisi $\lceil (m-1)/2 \rceil$ elemen, dan

pencarian berlanjut di salah satunya. Yang lebih besar di antara keduanya berisi $\lceil (m-1)/2 \rceil = \lfloor m/2 \rfloor$ elemen. Jadi, pada kasus terburuk, setiap pemeriksaan mengganti m dengan $\lfloor m/2 \rfloor$. Dengan menulis $T(m)$ untuk banyak pemeriksaan terbesar pada interval berisi m elemen,

$$T(0) = 0, \qquad T(m) = 1 + T(\lfloor m/2 \rfloor) \quad (m \geq 1). \tag{2.4}$$

Uraikan Persamaan (2.4) mulai dari $m = n$: ukurannya $n, \lfloor n/2 \rfloor, \lfloor n/4 \rfloor, \ldots$, dan suku ke-$k$ adalah $\lfloor n/2^k \rfloor$. Pemeriksaan berlanjut selama suku ini paling sedikit 1, yaitu selama $2^k \leq n$, yaitu untuk $k = 0,1, \ldots, \lfloor \lg n \rfloor$. Itu ada $\lfloor \lg n \rfloor + 1$ nilai k, sehingga

$$T(n) = \lfloor \lg n \rfloor + 1. \tag{2.5}$$

> **CONTOH:** Untuk $n = 9$, $\lg 9 \approx 3.17$, jadi $T(9) = 3 + 1 = 4$: tidak ada pencarian pada larik **Gambar 2.3** yang memerlukan lebih dari empat pemeriksaan. Untuk katalog sepuluh ribu buku di Subbab 2.10, $\lg 10^4 \approx 13.29$ dan $T = 14$. Untuk sejuta buku, $T = 20$; untuk semiliar buku, $T = 30$. Pencarian sekuensial pada semiliar buku memerlukan, pada kasus terburuk, semiliar pemeriksaan.

Perbandingan terakhir itu adalah seluruh alasan untuk memakai pencarian biner. Menggandakan n menambah *satu* pemeriksaan pada pencarian biner dan menggandakan biaya pencarian sekuensial. Rata-ratanya hanya sedikit di bawah kasus terburuk: pencarian berhasil atas n kunci yang sama mungkinnya memerlukan sekitar $\lg n - 1$ pemeriksaan, karena separuh dari semua elemen terletak pada tingkat pembagian terakhir [2]. Subbab 2.10 mengukur 12.4 pemeriksaan pada 10^4 kunci, dibandingkan dengan $\lg 10^4 - 1 \approx 12.3$.

2.5.5 Pencarian biner secara rekursif

Pencarian biner memiliki uraian rekursif yang alami: untuk mencari di $A[low..high]$, periksa bagian tengah, dan jika itu belum menuntaskan persoalan, *carilah di satu paruh* — yang merupakan masalah yang sama pada interval yang lebih pendek. **Pseudokode 2.4** menuliskannya.

Pseudokode 2.4 `BINARY-SEARCH-REC`: bentuk rekursif. Panggilan pertamanya adalah `BINARY-SEARCH-REC(A, x, 0, n - 1)`.

```
BINARY-SEARCH-REC(A, x, low, high):
 1.  if low > high
 2.      return -1                          // interval kosong: tidak ada
 3.  mid ← low + ⌊(high - low)/2⌋
 4.  if A[mid] = x
 5.      return mid
 6.  else if A[mid] < x
 7.      return BINARY-SEARCH-REC(A, x, mid + 1, high)
 8.  else
 9.      return BINARY-SEARCH-REC(A, x, low, mid - 1)
```

Bentuk rekursif membuat argumen kebenarannya hampir otomatis: jika prosedur ini benar untuk semua interval yang lebih pendek, ia benar untuk interval ini, karena ia menjawab langsung atau menyerahkan pertanyaannya kepada interval yang lebih pendek yang pasti memuat kunci. Itulah pembuktian dengan induksi, dan Bab 10 seluruhnya dicurahkan untuk cara berpikir seperti ini. Biayanya satu pemanggilan fungsi per pemeriksaan, dan satu bingkai tumpukan (*stack frame*) per tingkat rekursi — paling banyak $\lfloor \lg n \rfloor + 2$ bingkai, yaitu 32 untuk semiliar elemen, sehingga ruang tumpukan tidak pernah menjadi masalah di sini. Karena panggilan rekursif adalah hal

terakhir yang dilakukan setiap panggilan (*tail call*), rekursi ini dapat diubah secara mekanis menjadi gelung **Pseudokode 2.3**, dan banyak kompiler melakukannya.

2.5.6 Programnya

Kode 2.7 menjalankan versi iteratif dengan jejak tercetak untuk 17 dan untuk 10, versi rekursif untuk lima kunci, dan akhirnya mencari 088 dan 060 di rak yang terurut. **Kode 2.8** dan **Kode 2.9** adalah program yang sama dalam Java dan Python.

Kode 2.7 `binary_search.cpp`: pencarian biner iteratif dengan jejak, dan bentuk rekursifnya.

```
// Pencarian biner, iteratif (dengan jejak) dan rekursif, pada bilangan dan rak.
#include <cstdio>
#include <vector>
using namespace std;

// Interval tertutup [low, high]. Hasil: indeks atau -1; pemeriksaan dan low akhir
// (tempat kunci akan disisipkan) dikembalikan lewat parameter referensi.
int binarySearch(const vector<int>& data, int key, int& probes, int& lowOut,
                 bool trace) {
    int low = 0, high = (int)data.size() - 1;
    probes = 0;
    while (low <= high) {  // interval belum kosong
        int mid = low + (high - low) / 2;
        probes++;
        if (data[mid] == key) {
            if (trace)
                printf("  low=%d high=%d mid=%d data[mid]=%d -> found\n", low, high,
                       mid, data[mid]);
            lowOut = low;
            return mid;
        }
        if (data[mid] < key) {
            if (trace)
                printf("  low=%d high=%d mid=%d data[mid]=%d -> %d > %d, "
                       "low = %d\n",
                       low, high, mid, data[mid], key, data[mid], mid + 1);
            // kunci hanya mungkin di kanan mid
            low = mid + 1;
        } else {
            if (trace)
                printf("  low=%d high=%d mid=%d data[mid]=%d -> %d < %d, "
                       "high = %d\n",
                       low, high, mid, data[mid], key, data[mid], mid - 1);
            // kunci hanya mungkin di kiri mid
            high = mid - 1;
        }
    }
    // low adalah tempat kunci akan disisipkan
    lowOut = low;
    return -1;
}
```

Kode 2.7 (Lanjutan) *binary_search.cpp: pencarian biner iteratif dengan jejak, dan bentuk rekursifnya.*

```cpp
// Pencarian yang sama: fungsi yang memanggil dirinya pada satu paruh.
int binarySearchRec(const vector<int>& data, int key, int low, int high,
                    int& calls) {
    calls++;
    if (low > high) return -1;
    int mid = low + (high - low) / 2;
    if (data[mid] == key) return mid;
    if (data[mid] < key) return binarySearchRec(data, key, mid + 1, high, calls);
    return binarySearchRec(data, key, low, mid - 1, calls);
}

int main() {
    vector<int> data = {3, 9, 11, 12, 15, 17, 23, 31, 35};
    printf("data:");
    for (int x : data) printf(" %d", x);
    printf("\n");
    int probes, low;
    for (int key : {17, 10}) {
        printf("iterative search for %d\n", key);
        int i = binarySearch(data, key, probes, low, true);
        if (i >= 0)
            printf("  result: index %d after %d probes\n", i, probes);
        else
            printf("  result: not found after %d probes; it would go at index %d\n",
                   probes, low);
    }
    for (int key : {17, 10, 3, 35, 40}) {
        int calls = 0;
        int i = binarySearchRec(data, key, 0, (int)data.size() - 1, calls);
        printf("recursive search for %2d: index %2d, %d calls\n", key, i, calls);
    }

    // rak, terurut menurut kode
    vector<int> codes = {25, 34, 41, 56, 63, 72, 88, 96};
    for (int key : {88, 60}) {
        int i = binarySearch(codes, key, probes, low, false);
        printf("shelf code %03d: index %d, %d probes\n", key, i, probes);
    }
    return 0;
}
```

Kode 2.8 `BinarySearch.java`: program yang sama dalam Java.

```java
// Pencarian biner, iteratif (dengan jejak) dan rekursif, pada bilangan dan rak.
public class BinarySearch {
    // Interval tertutup [low, high]. Hasil: {indeks atau -1, pemeriksaan, low}.
    static int[] binarySearch(int[] data, int key, boolean trace) {
        int low = 0, high = data.length - 1;
        int probes = 0;
        while (low <= high) {  // interval belum kosong
            int mid = low + (high - low) / 2;
            probes++;
            if (data[mid] == key) {
                if (trace)
                    System.out.printf(
                            "  low=%d high=%d mid=%d data[mid]=%d -> found%n", low,
                            high, mid, data[mid]);
                return new int[] {mid, probes, low};
            }
            if (data[mid] < key) {
                if (trace)
                    System.out.printf("  low=%d high=%d mid=%d data[mid]=%d -> " +
                                      "%d > %d, low = %d%n",
                            low, high, mid, data[mid], key, data[mid], mid + 1);
                // kunci hanya mungkin di kanan mid
                low = mid + 1;
            } else {
                if (trace)
                    System.out.printf("  low=%d high=%d mid=%d data[mid]=%d -> " +
                                      "%d < %d, high = %d%n",
                            low, high, mid, data[mid], key, data[mid], mid - 1);
                // kunci hanya mungkin di kiri mid
                high = mid - 1;
            }
        }
        // low adalah tempat kunci akan disisipkan
        return new int[] {-1, probes, low};
    }
```

Kode 2.8 (Lanjutan) *BinarySearch.java: program yang sama dalam Java.*

```
    // Pencarian yang sama: metode yang memanggil dirinya pada satu paruh.
    // Mengembalikan {indeks atau -1, banyak panggilan}.
    static int[] binarySearchRec(
            int[] data, int key, int low, int high, int calls) {
        if (low > high) return new int[] {-1, calls};
        int mid = low + (high - low) / 2;
        if (data[mid] == key) return new int[] {mid, calls};
        if (data[mid] < key)
            return binarySearchRec(data, key, mid + 1, high, calls + 1);
        return binarySearchRec(data, key, low, mid - 1, calls + 1);
    }

    public static void main(String[] args) {
        int[] data = {3, 9, 11, 12, 15, 17, 23, 31, 35};
        StringBuilder line = new StringBuilder("data:");
        for (int x : data) line.append(" ").append(x);
        System.out.println(line);
        for (int key : new int[] {17, 10}) {
            System.out.println("iterative search for " + key);
            int[] r = binarySearch(data, key, true);
            if (r[0] >= 0)
                System.out.printf(
                        "  result: index %d after %d probes%n", r[0], r[1]);
            else
                System.out.printf("  result: not found after %d probes; it would " +
                                  "go at index %d%n",
                        r[1], r[2]);
        }
        for (int key : new int[] {17, 10, 3, 35, 40}) {
            int[] r = binarySearchRec(data, key, 0, data.length - 1, 1);
            System.out.printf("recursive search for %2d: index %2d, %d calls%n",
                    key, r[0], r[1]);
        }

        // rak, terurut menurut kode
        int[] codes = {25, 34, 41, 56, 63, 72, 88, 96};
        for (int key : new int[] {88, 60}) {
            int[] r = binarySearch(codes, key, false);
            System.out.printf(
                    "shelf code %03d: index %d, %d probes%n", key, r[0], r[1]);
        }
    }
}
```

Kode 2.9 `binary_search.py`: program yang sama dalam Python.

```
"""Pencarian biner, iteratif (dengan jejak) dan rekursif, pada bilangan dan rak."""


def binary_search(data, key, trace=False):
    """Interval tertutup [low, high]. Hasil: (indeks atau -1, pemeriksaan, low)."""
    low, high = 0, len(data) - 1
    probes = 0
    while low <= high:  # interval belum kosong
        mid = low + (high - low) // 2
        probes += 1
        if data[mid] == key:
            if trace:
                print(
                    "  low=%d high=%d mid=%d data[mid]=%d -> found"
                    % (low, high, mid, data[mid])
                )
            return mid, probes, low
        if data[mid] < key:
            if trace:
                print(
                    "  low=%d high=%d mid=%d data[mid]=%d -> %d > %d, low = %d"
                    % (low, high, mid, data[mid], key, data[mid], mid + 1)
                )
            low = mid + 1  # kunci hanya mungkin di kanan mid
        else:
            if trace:
                print(
                    "  low=%d high=%d mid=%d data[mid]=%d -> %d < %d, high = %d"
                    % (low, high, mid, data[mid], key, data[mid], mid - 1)
                )
            high = mid - 1  # kunci hanya mungkin di kiri mid
    return -1, probes, low  # low adalah tempat kunci akan disisipkan


def binary_search_rec(data, key, low, high, calls=1):
    """Pencarian yang sama: fungsi yang memanggil dirinya pada satu paruh."""
    if low > high:
        return -1, calls
    mid = low + (high - low) // 2
    if data[mid] == key:
        return mid, calls
    if data[mid] < key:
        return binary_search_rec(data, key, mid + 1, high, calls + 1)
    return binary_search_rec(data, key, low, mid - 1, calls + 1)
```

Kode 2.9 (Lanjutan) *binary_search.py: program yang sama dalam Python.*

```
def main():
    data = [3, 9, 11, 12, 15, 17, 23, 31, 35]
    print("data:", " ".join(str(x) for x in data))
    for key in (17, 10):
        print("iterative search for %d" % key)
        i, probes, low = binary_search(data, key, trace=True)
        if i >= 0:
            print("  result: index %d after %d probes" % (i, probes))
        else:
            print(
                "  result: not found after %d probes; it would go at index %d"
                % (probes, low)
            )
    for key in (17, 10, 3, 35, 40):
        i, calls = binary_search_rec(data, key, 0, len(data) - 1)
        print("recursive search for %2d: index %2d, %d calls" % (key, i, calls))

    codes = [25, 34, 41, 56, 63, 72, 88, 96]  # rak, terurut menurut kode
    for key in (88, 60):
        i, probes, low = binary_search(codes, key)
        print("shelf code %03d: index %d, %d probes" % (key, i, probes))


if __name__ == "__main__":
    main()
```

Keluarannya, sama untuk ketiganya:

```
data: 3 9 11 12 15 17 23 31 35
iterative search for 17
  low=0 high=8 mid=4 data[mid]=15 -> 17 > 15, low = 5
  low=5 high=8 mid=6 data[mid]=23 -> 17 < 23, high = 5
  low=5 high=5 mid=5 data[mid]=17 -> found
  result: index 5 after 3 probes
iterative search for 10
  low=0 high=8 mid=4 data[mid]=15 -> 10 < 15, high = 3
  low=0 high=3 mid=1 data[mid]=9 -> 10 > 9, low = 2
  low=2 high=3 mid=2 data[mid]=11 -> 10 < 11, high = 1
  result: not found after 3 probes; it would go at index 2
recursive search for 17: index  5, 3 calls
recursive search for 10: index -1, 4 calls
recursive search for  3: index  0, 3 calls
recursive search for 35: index  8, 4 calls
recursive search for 40: index -1, 5 calls
shelf code 088: index 6, 3 probes
shelf code 060: index -1, 3 probes
```

Kedua jejak itu adalah jejak tangan di Subbab 2.5.2, baris demi baris. Hitungan rekursifnya memerlukan satu catatan: fungsi rekursif menghitung *panggilan*, dan pencarian yang gagal melakukan satu panggilan terakhir pada interval kosong yang tidak memeriksa apa pun. Jadi pencarian 10 melakukan 3 pemeriksaan dalam 4 panggilan, dan pencarian 40 melakukan 4 pemeriksaan — maksimum untuk $n = 9$, seperti dijanjikan Persamaan (2.5) — dalam 5 panggilan. Pencarian 35, elemen terakhir, juga memerlukan maksimum 4 pemeriksaan.

> **JEBAKAN:** Pencarian biner pada larik *tidak terurut* tidak membuat program berhenti dan tidak memberi peringatan. Ia hanya memberi jawaban yang salah. Pada rak dalam urutan kedatangan, [72,25,96,41,88,34,63,56], pencarian 25 memeriksa $A[3] = 41$, menyimpulkan bahwa 25 pasti di kiri, memeriksa $A[1] = 25$ dan kebetulan beruntung; pencarian 34 memeriksa 41, pergi ke kiri, memeriksa 25, pergi ke kanan, memeriksa 96, pergi ke kiri, lalu melaporkan bahwa 34 tidak ada padahal ia berada di indeks 5. Urutkan dulu (Bab 3), atau cari secara sekuensial.

2.6 Jebakan Meleset Satu, *Lower Bound*, dan *Upper Bound*

2.6.1 Interval tertutup dan setengah terbuka

Pseudokode 2.3 bekerja pada interval **tertutup** $[low, high]$: kedua ujungnya termasuk. Banyak pustaka, termasuk pustaka standar C++, lebih menyukai interval **setengah terbuka** $[low, high)$: low termasuk, $high$ tidak. Kedua konvensi itu benar. Yang tidak benar adalah mencampurnya, dan hampir setiap bug pencarian biner adalah campuran. **Tabel 2.2** menyandingkan keduanya.

Tabel 2.2 Dua konvensi interval. Setiap baris harus diambil dari kolom yang sama; program yang mengambil syarat gelung dari satu kolom dan pembaruan dari kolom lain adalah program yang salah.

	Tertutup [low,high]	Setengah terbuka [low,high)
Nilai awal	$low = 0, high = n - 1$	$low = 0, high = n$
Interval kosong bila	$low > high$	$low = high$
Syarat gelung	`low <= high`	`low < high`
Buang bagian kiri dan mid	$low \leftarrow mid + 1$	$low \leftarrow mid + 1$
Buang mid dan bagian kanan	$high \leftarrow mid - 1$	$high \leftarrow mid$
Banyak elemen	$high - low + 1$	$high - low$

Konvensi setengah terbuka punya dua keuntungan kecil yang menjelaskan kepopulerannya: banyak elemennya cukup $high - low$, dan larik kosong tidak memerlukan penanganan khusus ($low = high = 0$).

2.6.2 Empat jebakan

Setiap jebakan berikut pernah muncul dalam kode yang diterbitkan.

1. **Syarat gelung yang salah.** Interval tertutup dengan `while (low < high)`: gelung berhenti ketika tersisa satu elemen tanpa memeriksanya. Mencari 5 di [5] langsung gagal. Interval setengah terbuka dengan `while (low <= high)`: ketika $low = high = n$, gelung memeriksa $A[n]$, yang sudah melewati ujung larik.
2. **Pembaruan yang tidak mengecilkan interval.** Menulis $low \leftarrow mid$ alih-alih $mid + 1$. Ketika interval berisi dua elemen, $mid = low$, penugasan itu tidak mengubah apa pun, dan gelung berputar selamanya. Langkah pemeliharaan dalam pembuktian kita gagal tepat di sini: mid sudah diketahui terlalu kecil, dan mempertahankannya di dalam interval tidak menghasilkan kemajuan.
3. **Titik tengah yang meluap.** Menghitung $(low + high)/2$ dengan aritmetika 32-bit. Subbab 2.6.4 menjelaskannya.
4. **Mengabaikan duplikat.** Ketika kunci muncul beberapa kali, **Pseudokode 2.3** mengembalikan *salah satu* salinannya, dan salinan yang mana bergantung pada n dan pada letak salinan-salinan itu. Kode yang mengira ia mendapat salinan pertama adalah kode yang salah. Obatnya ada di subbab berikut.

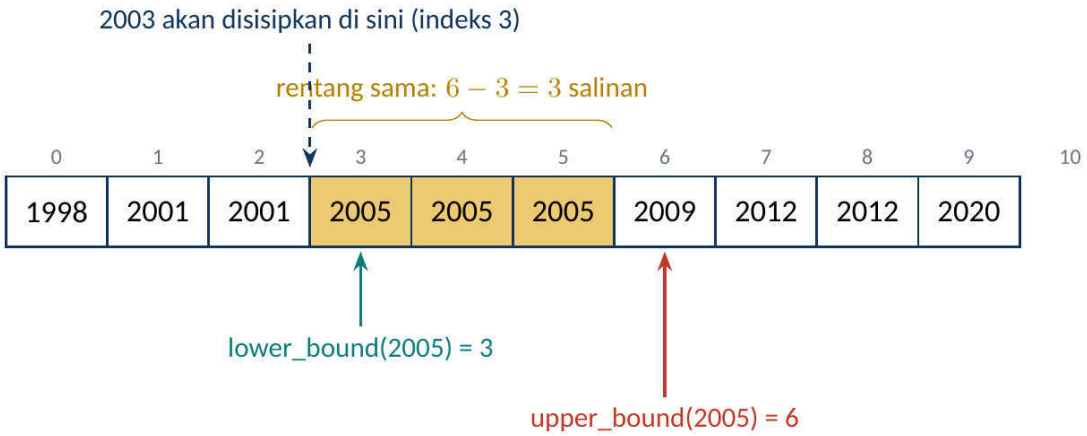
2.6.3 *Lower bound* dan *upper bound*

Pertanyaan yang lebih berguna daripada "apakah x ada di sini?" adalah "di mana tempat x?". Dua fungsi menjawabnya.

> **DEFINISI.** Untuk larik terurut $A[0..n-1]$ dan kunci x, ***lower bound*** (batas bawah) adalah indeks terkecil i dengan $A[i] \geq x$, dan ***upper bound*** (batas atas) adalah indeks terkecil i dengan $A[i] > x$; masing-masing bernilai n jika indeks seperti itu tidak ada. Salinan-salinan x

> menempati tepat $A[lower..upper-1]$, sehingga banyak salinannya $upper - lower$, dan x ada jika dan hanya jika bilangan itu positif. Jika x tidak ada, *lower bound* adalah indeks tempat x harus disisipkan agar A tetap terurut.

Gambar 2.5 memperlihatkan keduanya pada daftar tahun terbit yang berisi pengulangan.



Gambar 2.5 *Lower bound* dan *upper bound* dari 2005 dalam daftar tahun terbit yang terurut. Tiga salinan 2005 menempati indeks 3 sampai 5, yaitu rentang setengah terbuka [3,6). Untuk 2003 yang tidak ada, *lower bound* dan *upper bound* sama-sama 3, tempat 2003 akan disisipkan.

Pseudokode 2.5 menghitung *lower bound* pada interval setengah terbuka. Ia tidak pernah menguji kesamaan: ia hanya bertanya apakah $A[mid] < x$, lalu menggeser salah satu batas sampai interval kosong.

Pseudokode 2.5 `LOWER-BOUND`: posisi pertama yang elemennya tidak lebih kecil daripada x. Mengganti `<` dengan `≤` di baris 5 menghasilkan `UPPER-BOUND`.

```
LOWER-BOUND(A, n, x):              // A[0..n-1] terurut
 1.  low ← 0
 2.  high ← n                      // interval setengah terbuka [low, high)
 3.  while low < high
 4.      mid ← low + ⌊(high - low)/2⌋
 5.      if A[mid] < x
 6.          low ← mid + 1         // A[0..mid] semuanya < x
 7.      else
 8.          high ← mid            // A[mid..n-1] semuanya ≥ x
 9.  return low                    // indeks pertama dengan A[low] ≥ x (atau n)
```

Invariannya adalah kerabat dekat invarian pada **Gambar 2.4**: setiap elemen $A[0..low-1]$ lebih kecil daripada x, dan setiap elemen $A[high..n-1]$ paling sedikit x. Ketika gelung berakhir, $low = high$, wilayah yang belum diketahui telah lenyap, dan low adalah batas antara "lebih kecil daripada x" dan "paling sedikit x" — yaitu *lower bound*. Interval mengecil paling sedikit separuhnya setiap kali, sehingga gelung melakukan paling banyak $\lceil \lg(n+1) \rceil$ pemeriksaan, dan — tidak seperti **Pseudokode 2.3** — ia melakukan hampir tepat sebanyak itu pada setiap pencarian, karena ia tidak pernah berhenti lebih awal. Untuk *upper bound*, ubah ujinya menjadi $A[mid] \le x$, dan batasnya bergeser ke antara "paling banyak x" dan "lebih besar daripada x".

2.6.4 Titik tengah yang meluap

Pada 2006 Joshua Bloch melaporkan bahwa pencarian biner di pustaka Java sendiri, dan pencarian biner di *Programming Pearls* karya Bentley yang menjadi sumbernya, memuat bug yang tidak diketahui selama sekitar dua dasawarsa [4]: baris `int mid = (low + high) / 2;`. Jika low dan $high$ sama-sama besar — lebih dari sekitar 2^{30}, kira-kira semiliar — jumlahnya

melampaui `int` 32-bit terbesar, $2^{31} - 1 = 2{,}147{,}483{,}647$, dan jumlah itu berputar balik menjadi bilangan negatif. Membaginya dengan dua tetap menghasilkan bilangan negatif, dan program mengindeks larik dengan bilangan negatif. Larik berisi lebih dari semiliar elemen jarang ada ketika kode itu ditulis; pada 2006 tidak lagi.

Perbaikannya adalah menghitung titik tengah sebagai

$$mid = low + \left\lfloor \frac{high - low}{2} \right\rfloor, \tag{2.6}$$

yang secara matematis identik tetapi tidak pernah membentuk bilangan yang lebih besar daripada $high$. Setiap pencarian biner di buku ini memakai Persamaan (2.6). Bilangan bulat Python tidak pernah meluap, sehingga bug ini tidak dapat terjadi di sana, tetapi kebiasaan ini tidak memakan biaya apa pun.

Di C++, luapan lebih buruk daripada jawaban yang salah: luapan bilangan bulat bertanda adalah *perilaku tak terdefinisi* (*undefined behavior*), dan kompiler boleh berasumsi bahwa hal itu tidak pernah terjadi. Karena itu **Kode 2.10** tidak meluapkan `int` sungguhan; ia menghitung dalam 64 bit nilai yang akan disimpan oleh `int` 32-bit. Versi Java, **Kode 2.11**, meluapkan `int` sungguhan, karena di Java hal itu adalah perilaku yang terdefinisi: nilainya sekadar berputar balik. Versi Python, **Kode 2.12**, meniru putaran balik 32-bit itu dengan sedikit aritmetika, karena bilangan bulat Python sendiri tidak pernah meluap.

2.6.5 Programnya

Kode 2.10 mengimplementasikan *lower bound* dan *upper bound*, menjawab lima pertanyaan tentang tahun-tahun pada **Gambar 2.5**, menjalankan versi `low = mid` yang keliru dengan batas pengaman untuk memperlihatkan bahwa ia berputar selamanya, dan memperagakan luapan tadi.

Kode 2.10 `bounds.cpp`: *lower bound*, *upper bound*, pencarian biner yang tidak berhenti, dan titik tengah yang meluap.

```cpp
// lower_bound dan upper_bound buatan sendiri, dan dua bug klasik.
#include <cstdint>
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

// Indeks i pertama dengan a[i] >= key (atau a.size()). Setengah terbuka.
int lowerBound(const vector<int>& a, int key) {
    int low = 0, high = (int)a.size();
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (a[mid] < key)
            low = mid + 1;  // a[0..mid] semuanya < key
        else
            high = mid;  // a[mid..] semuanya >= key
    }
    return low;
}

// Indeks i pertama dengan a[i] > key (a.size() jika tidak ada).
int upperBound(const vector<int>& a, int key) {
    int low = 0, high = (int)a.size();
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (a[mid] <= key)
            low = mid + 1;
        else
            high = mid;
    }
    return low;
}

// "low = mid", bukan "low = mid + 1": bisa tak pernah selesai. Hasil false
// jika masih berputar setelah `limit` langkah.
bool buggyLowerBound(const vector<int>& a, int key, int limit, int& low,
                     int& high) {
    low = 0;
    high = (int)a.size();
    int steps = 0;
    while (low < high) {
        if (++steps > limit) return false;
        int mid = (low + high) / 2;
        // BUG: mid sendiri sudah diketahui terlalu kecil
        if (a[mid] < key)
            low = mid;
```

Kode 2.10 (Lanjutan) *bounds.cpp: lower bound, upper bound, pencarian biner yang tidak berhenti, dan titik tengah yang meluap.*

```
        else
            high = mid;
    }
    return true;
}

// Isi int bertanda 32-bit setelah penjumlahan meluap. (Membiarkan int
// sungguhan meluap adalah undefined behavior di C++; kita hitung dalam 64 bit.)
int64_t wrap32(int64_t x) {
    const int64_t two31 = INT64_C(1) << 31, two32 = INT64_C(1) << 32;
    return ((x + two31) % two32 + two32) % two32 - two31;
}

int main() {
    vector<int> years = {1998, 2001, 2001, 2005, 2005,
                         2005, 2009, 2012, 2012, 2020};
    printf("years:");
    for (int y : years) printf(" %d", y);
    printf("\nkey   lower upper count  meaning\n");
    for (int key : {2005, 2001, 2003, 1990, 2024}) {
        int lo = lowerBound(years, key), up = upperBound(years, key);
        string meaning = up > lo ? "first at " + to_string(lo) + ", last at " +
                                   to_string(up - 1)
                                 : "absent, insert at " + to_string(lo);
        printf("%d  %5d %5d %5d  %s\n", key, lo, up, up - lo, meaning.c_str());
    }

    int low, high;
    if (!buggyLowerBound(years, 2020, 20, low, high))
        printf("buggy version, key 2020: still looping after 20 steps "
               "(low=%d, high=%d)\n", low, high);

    int64_t l = 1500000000, h = 2000000000;
    int64_t s = wrap32(l + h);
    printf("low + high = %lld, but a 32-bit int holds %lld\n", (long long)(l + h),
           (long long)s);
    printf("(low + high) / 2 in 32 bits = %lld  <- a negative index\n",
           (long long)(s / 2));
    printf("low + (high - low) / 2     = %lld\n", (long long)(l + (h - l) / 2));
    return 0;
}
```

Kode 2.11 `Bounds.java`: program yang sama dalam Java, tempat luapan terjadi pada `int` sungguhan.

```java
// lower_bound dan upper_bound buatan sendiri, dan dua bug klasik.
public class Bounds {
    // Indeks i pertama dengan a[i] >= key (atau a.length). Setengah terbuka.
    static int lowerBound(int[] a, int key) {
        int low = 0, high = a.length;
        while (low < high) {
            int mid = low + (high - low) / 2;
            if (a[mid] < key)
                low = mid + 1;  // a[0..mid] semuanya < key
            else
                high = mid;  // a[mid..] semuanya >= key
        }
        return low;
    }

    // Indeks i pertama dengan a[i] > key (a.length jika tidak ada).
    static int upperBound(int[] a, int key) {
        int low = 0, high = a.length;
        while (low < high) {
            int mid = low + (high - low) / 2;
            if (a[mid] <= key)
                low = mid + 1;
            else
                high = mid;
        }
        return low;
    }

    // "low = mid" alih-alih "low = mid + 1": bisa tidak pernah selesai.
    // Mengembalikan {selesai ? 1 : 0, low, high}.
    static int[] buggyLowerBound(int[] a, int key, int limit) {
        int low = 0, high = a.length, steps = 0;
        while (low < high) {
            if (++steps > limit) return new int[] {0, low, high};
            int mid = (low + high) / 2;
            // BUG: mid sendiri sudah diketahui terlalu kecil
            if (a[mid] < key)
                low = mid;
            else
                high = mid;
        }
        return new int[] {1, low, high};
    }
```

Kode 2.11 (Lanjutan) *Bounds.java: program yang sama dalam Java, tempat luapan terjadi pada int sungguhan.*

```
    public static void main(String[] args) {
        int[] years = {1998, 2001, 2001, 2005, 2005, 2005, 2009, 2012, 2012, 2020};
        StringBuilder line = new StringBuilder("years:");
        for (int y : years) line.append(" ").append(y);
        System.out.println(line);
        System.out.println("key   lower upper count  meaning");
        for (int key : new int[] {2005, 2001, 2003, 1990, 2024}) {
            int lo = lowerBound(years, key), up = upperBound(years, key);
            String meaning = up > lo ? "first at " + lo + ", last at " + (up - 1)
                                     : "absent, insert at " + lo;
            System.out.printf(
                    "%d  %5d %5d %5d  %s%n", key, lo, up, up - lo, meaning);
        }

        int[] b = buggyLowerBound(years, 2020, 20);
        if (b[0] == 0)
            System.out.printf("buggy version, key 2020: still looping after 20 "
                    + "steps (low=%d, high=%d)%n", b[1], b[2]);

        int low = 1500000000, high = 2000000000;
        int sum = low + high;  // int Java berputar balik tanpa peringatan
        System.out.printf("low + high = %d, but a 32-bit int holds %d%n",
                (long) low + high, sum);
        System.out.printf("(low + high) / 2 in 32 bits = %d  <- a negative index%n",
                (low + high) / 2);
        System.out.printf(
                "low + (high - low) / 2     = %d%n", low + (high - low) / 2);
    }
}
```

Kode 2.12 `bounds.py`: program yang sama dalam Python.

```python
"""lower_bound dan upper_bound buatan sendiri, dan dua bug klasik."""


def lower_bound(a, key):
    """Indeks i pertama dengan a[i] >= key (atau len(a)). Setengah terbuka."""
    low, high = 0, len(a)
    while low < high:
        mid = low + (high - low) // 2
        if a[mid] < key:
            low = mid + 1  # a[0..mid] semuanya < key
        else:
            high = mid  # a[mid..] semuanya >= key
    return low


def upper_bound(a, key):
    """Indeks i pertama dengan a[i] > key (len(a) jika tidak ada)."""
    low, high = 0, len(a)
    while low < high:
        mid = low + (high - low) // 2
        if a[mid] <= key:
            low = mid + 1
        else:
            high = mid
    return low


def buggy_lower_bound(a, key, limit):
    """'low = mid' alih-alih 'low = mid + 1': bisa tidak pernah selesai."""
    low, high = 0, len(a)
    steps = 0
    while low < high:
        steps += 1
        if steps > limit:
            return None, low, high
        mid = (low + high) // 2
        if a[mid] < key:
            low = mid  # BUG: mid sendiri sudah diketahui terlalu kecil
        else:
            high = mid
    return low, low, high
```

Kode 2.12 (Lanjutan) *bounds.py: program yang sama dalam Python.*

```
def wrap32(x):
    """Nilai yang akan disimpan int bertanda 32-bit setelah penjumlahan meluap."""
    return (x + 2**31) % 2**32 - 2**31


def main():
    years = [1998, 2001, 2001, 2005, 2005, 2005, 2009, 2012, 2012, 2020]
    print("years:", " ".join(str(y) for y in years))
    print("key   lower upper count  meaning")
    for key in (2005, 2001, 2003, 1990, 2024):
        lo, up = lower_bound(years, key), upper_bound(years, key)
        if up > lo:
            meaning = "first at %d, last at %d" % (lo, up - 1)
        else:
            meaning = "absent, insert at %d" % lo
        print("%d  %5d %5d %5d  %s" % (key, lo, up, up - lo, meaning))

    result, low, high = buggy_lower_bound(years, 2020, 20)
    if result is None:
        print(
            "buggy version, key 2020: still looping after 20 steps "
            "(low=%d, high=%d)" % (low, high)
        )

    low, high = 1500000000, 2000000000
    s = wrap32(low + high)
    print("low + high = %d, but a 32-bit int holds %d" % (low + high, s))
    print("(low + high) / 2 in 32 bits = %d  <- a negative index" % int(s / 2))
    print("low + (high - low) / 2     = %d" % (low + (high - low) // 2))


if __name__ == "__main__":
    main()
```

Ketiganya mencetak:

```
years: 1998 2001 2001 2005 2005 2005 2009 2012 2012 2020
key   lower upper count  meaning
2005      3     6     3  first at 3, last at 5
2001      1     3     2  first at 1, last at 2
2003      3     3     0  absent, insert at 3
1990      0     0     0  absent, insert at 0
2024     10    10     0  absent, insert at 10
buggy version, key 2020: still looping after 20 steps (low=8, high=9)
low + high = 3500000000, but a 32-bit int holds -794967296
(low + high) / 2 in 32 bits = -397483648  <- a negative index
low + (high - low) / 2     = 1750000000
```

Cocokkan tabel itu dengan **Gambar 2.5**. Untuk 2005, salinannya ada di 3, 4, dan 5. Untuk 2003, kedua batasnya 3, sehingga banyaknya 0 dan titik sisipnya 3. Untuk 1990, yang lebih kecil daripada semuanya, kedua batasnya 0; untuk 2024, yang lebih besar daripada semuanya, keduanya $n = 10$. Versi yang keliru mencapai $low = 8, high = 9$ dan tidak dapat maju lagi: $mid = 8$, $A[8] = 2012 < 2020$, dan $low \leftarrow 8$ tidak mengubah apa pun. Tiga baris terakhir adalah bug Bloch dalam bentuk mini: dua indeks yang sepenuhnya sah, yang rata-ratanya, bila dihitung dengan cara naif, menjadi $-397{,}483{,}648$.

> **WAWASAN KUNCI:** *Lower bound* dan *upper bound* adalah pencarian biner yang sebaiknya Anda pakai lebih dahulu. Keduanya menjawab "apakah ia ada?" (apakah $upper > lower$?), "di mana salinan pertama?" ($lower$), "di mana yang terakhir?" ($upper - 1$), "ada berapa?" ($upper - lower$), "di mana ia harus diletakkan?" ($lower$), dan "ada berapa dalam rentang

$[a, b)$?" ($lower(b) - lower(a)$) — masing-masing dengan paling banyak $\lceil \lg(n+1) \rceil$ pemeriksaan.

2.7 Pencarian Interpolasi

2.7.1 Menebak di mana kunci seharusnya berada

Pencarian biner selalu memeriksa bagian tengah. Namun, jika Anda mencari nama *Tanaka* di buku telepon cetak yang tebal, Anda tidak membukanya di tengah; Anda membukanya kira-kira tiga perempat jalan, karena Anda tahu bahwa T terletak di belakang dalam abjad. Anda memakai bukan hanya *urutan* data, melainkan juga *nilainya*. **Pencarian interpolasi** (*interpolation search*) melakukan hal yang sama: ia menaksir di mana kunci seharusnya berada, dengan asumsi bahwa kunci naik secara teratur dari $A[low]$ ke $A[high]$, lalu memeriksa di sana. Gagasan ini berasal dari Peterson pada 1957 [5].

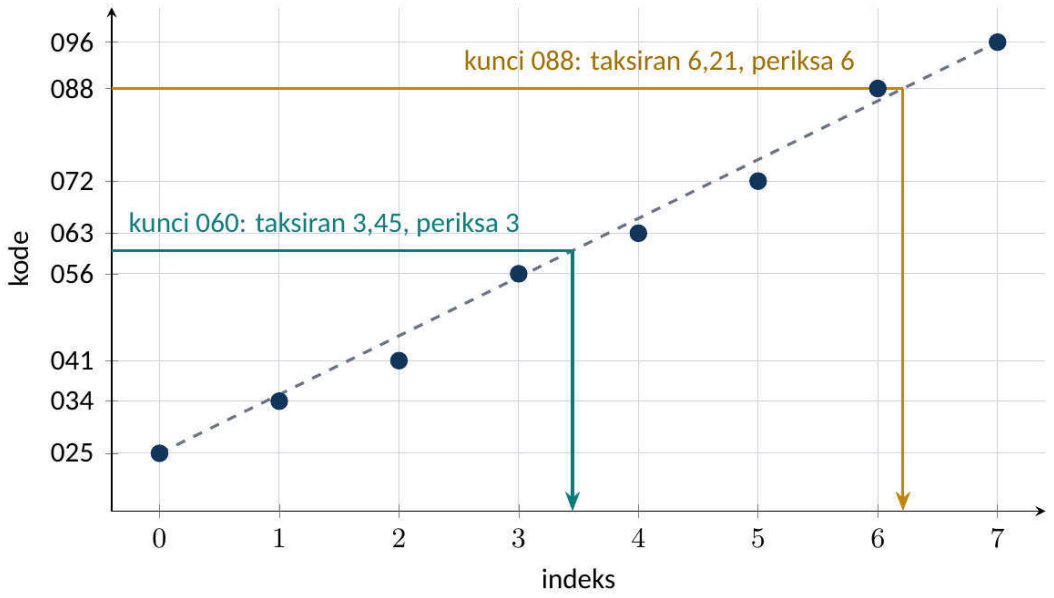
Jika kunci naik menurut garis lurus dari $A[low]$ di posisi low ke $A[high]$ di posisi $high$, maka kunci x terletak pada pecahan $(x - A[low])/(A[high] - A[low])$ dari seluruh jarak, dan posisi taksirannya adalah

$$pos = low + \left\lfloor \frac{x - A[low]}{A[high] - A[low]} \cdot (high - low) \right\rfloor. \tag{2.7}$$

Ketika $x = A[low]$, pecahannya 0 dan $pos = low$; ketika $x = A[high]$, pecahannya 1 dan $pos = high$; di antaranya, pemeriksaan bergeser secara sebanding. Pencarian biner adalah kasus khusus yang mengabaikan nilai-nilai itu dan selalu memakai pecahan $\frac{1}{2}$.

2.7.2 Rak itu, dengan tangan

Gambar 2.6 memetakan delapan kode dari **Tabel 2.1** terhadap indeksnya. Titik-titiknya terletak dekat garis lurus putus-putus dari (0,025) ke (7,096), yang persis merupakan asumsi Persamaan (2.7).



Gambar 2.6 Pencarian interpolasi pada rak komputer. Garis putus-putus membentang lurus dari code[0] = 025 ke code[7] = 096. Untuk kunci 088 ia menaksir posisi 6,21, dan pemeriksaan di indeks 6 langsung menemukan 088; untuk kunci 060 ia menaksir 3,45 dan memeriksa indeks 3.

Kunci 088. Dengan $low = 0$, $high = 7$:

$$pos = 0 + \left\lfloor \frac{88 - 25}{96 - 25} \cdot 7 \right\rfloor = \left\lfloor \frac{63}{71} \cdot 7 \right\rfloor = \lfloor 6.21 \rfloor = 6.$$

$A[6] = 088$: ditemukan dengan **satu** pemeriksaan. Pencarian biner memerlukan tiga.

Kunci 060. Dengan $low = 0$, $high = 7$: $pos = \lfloor (35/71) \cdot 7 \rfloor = \lfloor 3.45 \rfloor = 3$, dan $A[3] = 056 < 060$, jadi $low \leftarrow 4$. Sekarang kunci 060 *lebih kecil* daripada $A[low] = A[4] = 063$. Karena interval $A[4..7]$ hanya memuat kode yang paling sedikit 063, kunci tidak mungkin ada di sana, dan pencarian berhenti: 060 tidak ada, setelah satu pemeriksaan.

Langkah terakhir itu memperlihatkan sebuah pengaman yang diperlukan pencarian interpolasi tetapi tidak diperlukan pencarian biner. Persamaan (2.7) hanya bermakna bila $A[low] \le x \le A[high]$; di luar rentang itu pecahannya negatif atau lebih dari satu, dan "taksiran"-nya menunjuk ke luar interval — bahkan mungkin ke luar larik. Karena itu syarat gelung memeriksa rentang ini, dan kunci di luarnya langsung dilaporkan tidak ada.

Pengaman kedua diperlukan ketika $A[low] = A[high]$: dalam hal itu Persamaan (2.7) membagi dengan nol. Jika pemeriksaan rentang sudah lolos, setiap elemen interval sama dengan x, dan memeriksa $A[low]$ sudah menuntaskan persoalan.

Pseudokode 2.6 `INTERPOLATION-SEARCH`: pencarian biner dengan titik tengah diganti taksiran, dijaga terhadap kunci di luar rentang dan terhadap penyebut nol.

```
INTERPOLATION-SEARCH(A, n, x):    // A[0..n-1] terurut, kunci numerik
 1.  low ← 0
 2.  high ← n − 1
 3.  while low ≤ high and A[low] ≤ x ≤ A[high]
 4.      if A[high] = A[low]
 5.          pos ← low                          // tanpa kemiringan: hindari bagi 0
 6.      else
 7.          pos ← low + ⌊(x − A[low]) / (A[high] − A[low]) · (high − low)⌋
 8.      if A[pos] = x
 9.          return pos
10.      else if A[pos] < x
11.          low ← pos + 1
12.      else
13.          high ← pos − 1
14.  return −1
```

2.7.3 Programnya

Kode 2.13 mencari lima kode di rak dan mencetak setiap taksiran. Program ini menghitung pecahannya dengan bilangan titik-mengambang, seperti rumusnya, lalu mengubahnya menjadi indeks dengan pemotongan, yang untuk bilangan tak negatif sama dengan *floor*. **Kode 2.14** dan **Kode 2.15** melakukan hal yang sama dalam Java dan Python.

Kode 2.13 `interpolation_search.cpp`: pencarian interpolasi pada rak komputer, mencetak setiap taksiran.

```cpp
// Pencarian interpolasi pada rak komputer Pustaka Ceria, terurut menurut kode.
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

const vector<int> CODES = {25, 34, 41, 56, 63, 72, 88, 96};
const vector<string> TITLES = {
        "The C++ Programming",       "Mastering Delphi 6",
        "Professional C#",           "Pure JavaScript v2",
        "Advanced JSP & Servlet",    "Calculus Make it Easy",
        "Visual Basic 2005 Express", "Artificial Life : Volume 1"};

// Mengembalikan indeks atau -1, sambil mencetak setiap taksiran.
int interpolationSearch(const vector<int>& data, int key, int& probes) {
    int low = 0, high = (int)data.size() - 1;
    probes = 0;
    while (low <= high && data[low] <= key && key <= data[high]) {
        double estimate;
        if (data[high] == data[low])  // semua sama: tanpa kemiringan
            estimate = 0.0;
        else
            estimate = (double)(key - data[low]) /
                       (double)(data[high] - data[low]) * (high - low);
        // cast memotong pecahan, di sini sama dengan floor
        int pos = low + (int)estimate;
        probes++;
        printf("  low=%d high=%d: position = %d + %.2f -> %d, code[%d] = %03d\n",
               low, high, low, estimate, pos, pos, data[pos]);
        if (data[pos] == key) return pos;
        if (data[pos] < key)
            low = pos + 1;
        else
            high = pos - 1;
    }
    if (low > high)
        printf("  low=%d > high=%d: the range is empty\n", low, high);
    else
        printf("  key %03d lies outside code[%d..%d] = %03d..%03d\n", key, low,
               high, data[low], data[high]);
    return -1;
}
```

Kode 2.13 (Lanjutan) *interpolation_search.cpp: pencarian interpolasi pada rak komputer, mencetak setiap taksiran.*

```cpp
int main() {
    for (size_t i = 0; i < CODES.size(); i++)
        printf("[%zu] %03d %s\n", i, CODES[i], TITLES[i].c_str());
    for (int key : {88, 60, 72, 96, 30}) {
        printf("search %03d\n", key);
        int probes;
        int i = interpolationSearch(CODES, key, probes);
        if (i >= 0)
            printf("  found: %s (%d probe%s)\n", TITLES[i].c_str(), probes,
                   probes == 1 ? "" : "s");
        else
            printf("  not found (%d probe%s)\n", probes, probes == 1 ? "" : "s");
    }
    return 0;
}
```

Kode 2.14 `InterpolationSearch.java`: program yang sama dalam Java. `Locale.ROOT` membuat tanda desimalnya berupa titik di komputer mana pun.

```
// Pencarian interpolasi pada rak komputer Pustaka Ceria, terurut menurut kode.
import java.util.Locale;

public class InterpolationSearch {
    static final int[] CODES = {25, 34, 41, 56, 63, 72, 88, 96};
    static final String[] TITLES = {"The C++ Programming", "Mastering Delphi 6",
            "Professional C#", "Pure JavaScript v2", "Advanced JSP & Servlet",
            "Calculus Make it Easy", "Visual Basic 2005 Express",
            "Artificial Life : Volume 1"};

    // Mengembalikan {indeks atau -1, pemeriksaan}, sambil mencetak setiap taksiran.
    static int[] interpolationSearch(int[] data, int key) {
        int low = 0, high = data.length - 1;
        int probes = 0;
        while (low <= high && data[low] <= key && key <= data[high]) {
            double estimate;
            // semua sama: tanpa kemiringan
            if (data[high] == data[low])
                estimate = 0.0;
            else
                estimate = (double) (key - data[low]) /
                        (double) (data[high] - data[low]) * (high - low);
            // cast memotong pecahan, di sini sama dengan floor
            int pos = low + (int) estimate;
            probes++;
            System.out.printf(Locale.ROOT,
                    "  low=%d high=%d: position = %d + %.2f -> %d, " +
                    "code[%d] = %03d%n",
                    low, high, low, estimate, pos, pos, data[pos]);
            if (data[pos] == key) return new int[] {pos, probes};
            if (data[pos] < key)
                low = pos + 1;
            else
                high = pos - 1;
        }
        if (low > high)
            System.out.printf(
                    "  low=%d > high=%d: the range is empty%n", low, high);
        else
            System.out.printf("  key %03d lies outside code[%d..%d] = %03d..%03d%n",
                    key, low, high, data[low], data[high]);
        return new int[] {-1, probes};
    }
```

Kode 2.14 (Lanjutan) *InterpolationSearch.java: program yang sama dalam Java. Locale.ROOT membuat tanda desimalnya berupa titik di komputer mana pun.*

```
    public static void main(String[] args) {
        for (int i = 0; i < CODES.length; i++)
            System.out.printf("[%d] %03d %s%n", i, CODES[i], TITLES[i]);
        for (int key : new int[] {88, 60, 72, 96, 30}) {
            System.out.printf("search %03d%n", key);
            int[] r = interpolationSearch(CODES, key);
            String s = r[1] == 1 ? "" : "s";
            if (r[0] >= 0)
                System.out.printf(
                        "  found: %s (%d probe%s)%n", TITLES[r[0]], r[1], s);
            else
                System.out.printf("  not found (%d probe%s)%n", r[1], s);
        }
    }
}
```

Kode 2.15 `interpolation_search.py`: program yang sama dalam Python.

```
"""Pencarian interpolasi pada rak komputer Pustaka Ceria, terurut menurut kode."""

CODES = [25, 34, 41, 56, 63, 72, 88, 96]
TITLES = [
    "The C++ Programming",
    "Mastering Delphi 6",
    "Professional C#",
    "Pure JavaScript v2",
    "Advanced JSP & Servlet",
    "Calculus Make it Easy",
    "Visual Basic 2005 Express",
    "Artificial Life : Volume 1",
]


def interpolation_search(data, key):
    """Hasil: (indeks atau -1, pemeriksaan); setiap taksiran dicetak."""
    low, high = 0, len(data) - 1
    probes = 0
    while low <= high and data[low] <= key <= data[high]:
        if data[high] == data[low]:  # semua sama: tanpa kemiringan
            estimate = 0.0
        else:
            estimate = (
                float(key - data[low])
                / float(data[high] - data[low])
                * (high - low)
            )
        pos = low + int(estimate)  # int() memotong, di sini = floor
        probes += 1
        print(
            "  low=%d high=%d: position = %d + %.2f -> %d, code[%d] = %03d"
            % (low, high, low, estimate, pos, pos, data[pos])
        )
        if data[pos] == key:
            return pos, probes
        if data[pos] < key:
            low = pos + 1
        else:
            high = pos - 1
    if low > high:
        print("  low=%d > high=%d: the range is empty" % (low, high))
    else:
        print(
            "  key %03d lies outside code[%d..%d] = %03d..%03d"
            % (key, low, high, data[low], data[high])
```

Kode 2.15 (Lanjutan) *interpolation_search.py: program yang sama dalam Python.*

```
        )
    return -1, probes


def main():
    for i in range(len(CODES)):
        print("[%d] %03d %s" % (i, CODES[i], TITLES[i]))
    for key in (88, 60, 72, 96, 30):
        print("search %03d" % key)
        i, probes = interpolation_search(CODES, key)
        if i >= 0:
            print(
                "  found: %s (%d probe%s)"
                % (TITLES[i], probes, "" if probes == 1 else "s")
            )
        else:
            print("  not found (%d probe%s)" % (probes, "" if probes == 1 else "s"))


if __name__ == "__main__":
    main()
```

Ketiganya mencetak:

```
[0] 025 The C++ Programming
[1] 034 Mastering Delphi 6
[2] 041 Professional C#
[3] 056 Pure JavaScript v2
[4] 063 Advanced JSP & Servlet
[5] 072 Calculus Make it Easy
[6] 088 Visual Basic 2005 Express
[7] 096 Artificial Life : Volume 1
search 088
  low=0 high=7: position = 0 + 6.21 -> 6, code[6] = 088
  found: Visual Basic 2005 Express (1 probe)
search 060
  low=0 high=7: position = 0 + 3.45 -> 3, code[3] = 056
  key 060 lies outside code[4..7] = 063..096
  not found (1 probe)
search 072
  low=0 high=7: position = 0 + 4.63 -> 4, code[4] = 063
  low=5 high=7: position = 5 + 0.00 -> 5, code[5] = 072
  found: Calculus Make it Easy (2 probes)
search 096
  low=0 high=7: position = 0 + 7.00 -> 7, code[7] = 096
  found: Artificial Life : Volume 1 (1 probe)
search 030
  low=0 high=7: position = 0 + 0.49 -> 0, code[0] = 025
  key 030 lies outside code[1..7] = 034..096
  not found (1 probe)
```

Jejak untuk 088 dan 060 adalah perhitungan tangan di atas. Kunci 072 memerlukan dua pemeriksaan: taksiran pertama, 4,63, mendarat di 063, dan taksiran kedua, pada interval $A[5..7]$, mendarat tepat di 072 karena kuncinya sama dengan $A[low]$. Kunci 030 ditaksir di 0,49, memeriksa 025, lalu ternyata lebih kecil daripada $A[1] = 034$: tidak ada, setelah satu pemeriksaan.

2.7.4 Cemerlang atau buruk: bergantung pada datanya

Bila kunci tersebar merata secara acak, analisis Perl, Itai, dan Avni menunjukkan bahwa pencarian interpolasi rata-rata melakukan sekitar $\lg\lg n$ pemeriksaan [6]. Fungsi $\lg\lg n$ tumbuh begitu lambat sehingga hampir konstan: untuk $n = 10^6$ nilainya sekitar 4,3, dan untuk $n = 10^{18}$

hanya sekitar 6. Subbab 2.10 mengukur 4,26 pemeriksaan pada sejuta kunci yang tersebar merata, dibandingkan dengan 18,8 untuk pencarian biner.

Namun, taksiran itu hanya sebaik asumsi garis lurusnya. Ketika kunci menumpuk — banyak nilai kecil dan sedikit nilai yang sangat besar — garis melalui $A[low]$ dan $A[high]$ menjadi pemandu yang buruk, taksiran mendarat jauh dari kunci, dan setiap pemeriksaan hanya menggeser batas interval beberapa tempat. Pada kasus terburuk pencarian interpolasi merosot menjadi $O(n)$ pemeriksaan, lebih buruk daripada pencarian biner dan tidak lebih baik daripada pencarian sekuensial. Subbab 2.10 memperlihatkan hal ini pada data nyata: pada pengenal (*identifier*) katalog buku sungguhan, yang sangat menumpuk, pencarian interpolasi memerlukan rata-rata 211 pemeriksaan, sedangkan pencarian biner memerlukan 12.

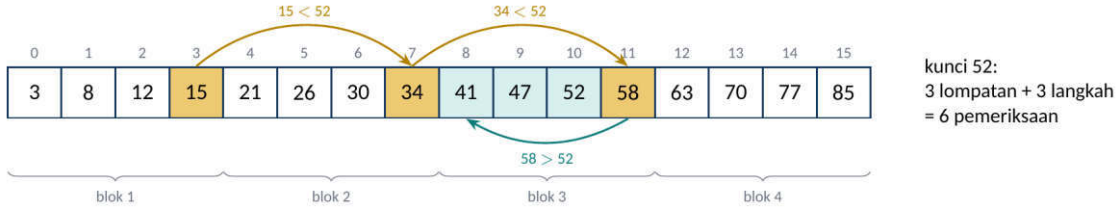
> **PERINGATAN:** Pencarian interpolasi memerlukan kunci yang dapat dikurangkan dan dibagi secara bermakna — bilangan, atau string yang sengaja Anda sandikan menjadi bilangan. Untuk kode seperti `B1001` atau untuk judul, pakailah pencarian biner. Dan setiap langkahnya memerlukan satu pembagian, yang pada prosesor masa kini beberapa kali lebih lambat daripada perbandingan, sehingga bahkan ketika pencarian interpolasi melakukan lebih sedikit pemeriksaan, ia tidak selalu lebih cepat.

2.8 Pencarian Lompat

2.8.1 Melompat maju per blok

Sebagian media penyimpanan hanya dapat dibaca dengan efisien ke satu arah — pita magnetik, berkas yang dibaca per blok besar, daftar yang dapat dilompati ke depan tetapi tidak ke belakang. Pada media seperti itu pencarian biner, yang melompat maju-mundur, terasa canggung, sedangkan pencarian sekuensial lambat. **Pencarian lompat** (*jump search*), yang dianalisis Shneiderman pada 1978 [7], adalah jalan tengahnya. Pada larik terurut ia melompat maju m tempat sekaligus, hanya melihat elemen terakhir dari setiap blok berukuran m, sampai tiba di blok yang elemen terakhirnya paling sedikit sebesar kunci. Kunci, jika ada, pasti berada di blok itu, dan penelusuran sekuensial yang pendek di dalam blok akan menemukannya. **Pseudokode 2.7** memberikan prosedurnya.

Gambar 2.7 memperlihatkan pencarian lompat untuk 52 di antara enam belas nomor panggil terurut dengan blok berukuran $m = 4$.



Gambar 2.7 Pencarian lompat untuk 52 dengan blok berisi empat. Lompatan memeriksa elemen terakhir setiap blok (15, 34, 58); karena $58 > 52$, kunci pasti berada di blok 3, dan penelusuran melalui 41, 47, 52 menemukannya.

Pseudokode 2.7 `JUMP-SEARCH`: lompati blok-blok utuh, lalu telusuri satu-satunya blok yang dapat memuat kunci.

```
JUMP-SEARCH(A, n, x, m):          // A[0..n-1] terurut, ukuran blok m
 1.  start ← 0
 2.  end ← min(m, n)              // blok saat ini adalah A[start..end-1]
 3.  while A[end - 1] < x         // lompati selama seluruh blok terlalu kecil
 4.      start ← end
 5.      end ← min(end + m, n)
 6.      if start ≥ n
 7.          return -1            // x lebih besar daripada semua elemen
 8.  for i ← start to end - 1     // telusuri satu blok itu
 9.      if A[i] = x
10.          return i
11.      if A[i] > x
12.          return -1            // sudah melewati tempat x seharusnya
13.  return -1
```

Program di bawah menyempurnakan baris 3 dan 8 sedikit: ketika elemen terakhir sebuah blok *sama dengan* kunci, pencarian langsung berhenti, dan penelusuran kemudian melewatkan elemen terakhir itu, yang sudah diperiksa. Dengan begitu hitungan pemeriksaan cocok persis dengan analisisnya.

2.8.2 Memilih ukuran blok

Pada kasus terburuk, pencarian lompat melakukan sekitar n/m lompatan (satu per blok) lalu menelusuri $m - 1$ elemen blok terakhir. Karena itu biaya kasus terburuknya adalah

$$f(m) = \frac{n}{m} + m - 1. \tag{2.8}$$

Blok besar berarti sedikit lompatan dan penelusuran panjang; blok kecil berarti banyak lompatan dan penelusuran pendek. Untuk mencari keseimbangan terbaik, perlakukan m sebagai bilangan real dan samakan turunan Persamaan (2.8) dengan nol: $f'(m) = -n/m^2 + 1 = 0$ memberikan $m^2 = n$, yaitu

$$m^* = \sqrt{n}, \qquad f(m^*) = 2\sqrt{n} - 1. \tag{2.9}$$

Turunan keduanya, $2n/m^3$, positif, sehingga titik ini adalah minimum. Anda juga dapat melihatnya tanpa kalkulus: hasil kali kedua suku n/m dan m selalu n, dan dua bilangan positif dengan hasil kali tetap memiliki jumlah terkecil ketika keduanya sama. Dengan $m = \lfloor\sqrt{n}\rfloor$, pencarian lompat memerlukan paling banyak sekitar $2\sqrt{n}$ pemeriksaan — 199 untuk $n = 10^4$ dan 1,999 untuk $n = 10^6$. Itu jauh lebih baik daripada n milik pencarian sekuensial tetapi jauh lebih buruk daripada $\lfloor \lg n \rfloor + 1$ milik pencarian biner; pencarian lompat baru pantas dipakai ketika lompatan mundur itu mahal.

2.8.3 Programnya

Kode 2.16 `jump_search.cpp`: pencarian lompat pada enam belas nomor panggil, dan kasus terburuk untuk beberapa n.

```cpp
// Pencarian lompat: melompat maju m tempat sekaligus, lalu menelusuri satu blok.
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

// Hasil: indeks atau -1; pemeriksaan lompat dan langkah dikembalikan terpisah.
int jumpSearch(const vector<int>& data, int key, int m, int& jumps, int& walks) {
    int n = (int)data.size();
    jumps = walks = 0;
    // blok saat ini adalah data[start..end-1]
    int start = 0, end = min(m, n);
    while (true) {
        jumps++;
        // elemen terakhir blok adalah kuncinya
        if (data[end - 1] == key) return end - 1;
        // kunci, jika ada, berada di blok ini
        if (data[end - 1] > key) break;
        start = end;
        end = min(end + m, n);
        // melompat melewati ujung: key > semua elemen
        if (start >= n) return -1;
    }
    for (int i = start; i < end - 1; i++) {  // data[end-1] sudah diperiksa
        walks++;
        if (data[i] == key) return i;
        if (data[i] > key) break;  // sudah melewati tempat kunci seharusnya berada
    }
    return -1;
}

int isqrt(int n) {  // m terbesar dengan m * m <= n
    int m = (int)sqrt((double)n);
    while (m * m > n) m--;
    while ((m + 1) * (m + 1) <= n) m++;
    return m;
}
```

Kode 2.16 (Lanjutan) *jump_search.cpp: pencarian lompat pada enam belas nomor panggil, dan kasus terburuk untuk beberapa n.*

```cpp
int main() {
    vector<int> calls = {3,  8,  12, 15, 21, 26, 30, 34,
                         41, 47, 52, 58, 63, 70, 77, 85};
    int m = isqrt((int)calls.size());
    printf("call numbers:");
    for (int x : calls) printf(" %d", x);
    printf("\nn = %zu, block size m = %d\n", calls.size(), m);
    for (int key : {52, 58, 60, 3, 90}) {
        int j, w;
        int i = jumpSearch(calls, key, m, j, w);
        string where = i >= 0 ? "index " + to_string(i) : "not found";
        printf("key %2d: %-9s  jump %d + walk %d = %d probes\n", key, where.c_str(),
               j, w, j + w);
    }
    printf("worst case n/m + m - 1 for the best m = floor(sqrt(n)):\n");
    for (int n : {16, 100, 10000, 1000000}) {
        int best = isqrt(n);
        printf("  n = %7d  m = %4d  worst = %4d probes\n", n, best,
               n / best + best - 1);
    }
    return 0;
}
```

Kode 2.17 `JumpSearch.java`: program yang sama dalam Java.

```java
// Pencarian lompat: melompat maju m tempat sekaligus, lalu menelusuri satu blok.
public class JumpSearch {
    // Mengembalikan {indeks atau -1, pemeriksaan lompat, pemeriksaan langkah}.
    static int[] jumpSearch(int[] data, int key, int m) {
        int n = data.length;
        int jumps = 0, walks = 0;
        // blok saat ini adalah data[start..end-1]
        int start = 0, end = Math.min(m, n);
        while (true) {
            jumps++;
            if (data[end - 1] == key)
                return new int[] {
                        end - 1, jumps, walks};  // elemen terakhir adalah kuncinya
            // kunci, jika ada, berada di blok ini
            if (data[end - 1] > key) break;
            start = end;
            end = Math.min(end + m, n);
            // key > semua elemen
            if (start >= n) return new int[] {-1, jumps, walks};
        }
        // data[end-1] sudah diperiksa
        for (int i = start; i < end - 1; i++) {
            walks++;
            if (data[i] == key) return new int[] {i, jumps, walks};
            // sudah melewati tempat kunci seharusnya berada
            if (data[i] > key) break;
        }
        return new int[] {-1, jumps, walks};
    }

    static int isqrt(int n) {  // m terbesar dengan m * m <= n
        int m = (int) Math.sqrt(n);
        while (m * m > n) m--;
        while ((m + 1) * (m + 1) <= n) m++;
        return m;
    }
```

Kode 2.17 (Lanjutan) *JumpSearch.java: program yang sama dalam Java.*

```java
    public static void main(String[] args) {
        int[] calls = {
                3, 8, 12, 15, 21, 26, 30, 34, 41, 47, 52, 58, 63, 70, 77, 85};
        int m = isqrt(calls.length);
        StringBuilder line = new StringBuilder("call numbers:");
        for (int x : calls) line.append(" ").append(x);
        System.out.println(line);
        System.out.printf("n = %d, block size m = %d%n", calls.length, m);
        for (int key : new int[] {52, 58, 60, 3, 90}) {
            int[] r = jumpSearch(calls, key, m);
            String where = r[0] >= 0 ? "index " + r[0] : "not found";
            System.out.printf("key %2d: %-9s  jump %d + walk %d = %d probes%n", key,
                    where, r[1], r[2], r[1] + r[2]);
        }
        System.out.println(
                "worst case n/m + m - 1 for the best m = floor(sqrt(n)):");
        for (int n : new int[] {16, 100, 10000, 1000000}) {
            int best = isqrt(n);
            System.out.printf("  n = %7d  m = %4d  worst = %4d probes%n", n, best,
                    n / best + best - 1);
        }
    }
}
```

Kode 2.18 `jump_search.py`: program yang sama dalam Python.

```
"""Pencarian lompat: melompat maju m tempat sekaligus, lalu menelusuri satu blok."""

import math


def jump_search(data, key, m):
    """Mengembalikan (indeks atau -1, pemeriksaan lompat, pemeriksaan langkah)."""
    n = len(data)
    jumps = walks = 0
    start, end = 0, min(m, n)  # blok saat ini adalah data[start..end-1]
    while True:
        jumps += 1
        if data[end - 1] == key:  # elemen terakhir blok adalah kuncinya
            return end - 1, jumps, walks
        if data[end - 1] > key:  # kunci, jika ada, berada di blok ini
            break
        start, end = end, min(end + m, n)
        if start >= n:  # melompat melewati ujung: key > semua elemen
            return -1, jumps, walks
    for i in range(start, end - 1):  # data[end-1] sudah diperiksa
        walks += 1
        if data[i] == key:
            return i, jumps, walks
        if data[i] > key:  # sudah melewati tempat kunci seharusnya berada
            break
    return -1, jumps, walks


def main():
    calls = [3, 8, 12, 15, 21, 26, 30, 34, 41, 47, 52, 58, 63, 70, 77, 85]
    m = math.isqrt(len(calls))
    print("call numbers:", " ".join(str(x) for x in calls))
    print("n = %d, block size m = %d" % (len(calls), m))
    for key in (52, 58, 60, 3, 90):
        i, j, w = jump_search(calls, key, m)
        where = "index %d" % i if i >= 0 else "not found"
        print(
            "key %2d: %-9s  jump %d + walk %d = %d probes"
            % (key, where, j, w, j + w)
        )
    print("worst case n/m + m - 1 for the best m = floor(sqrt(n)):")
    for n in (16, 100, 10000, 1000000):
        m = math.isqrt(n)
        print("  n = %7d  m = %4d  worst = %4d probes" % (n, m, n // m + m - 1))
```

Kode 2.18 (Lanjutan) *jump_search.py: program yang sama dalam Python.*

```
if __name__ == "__main__":
    main()
```

Kode 2.16, **Kode 2.17**, dan **Kode 2.18** semuanya mencetak:

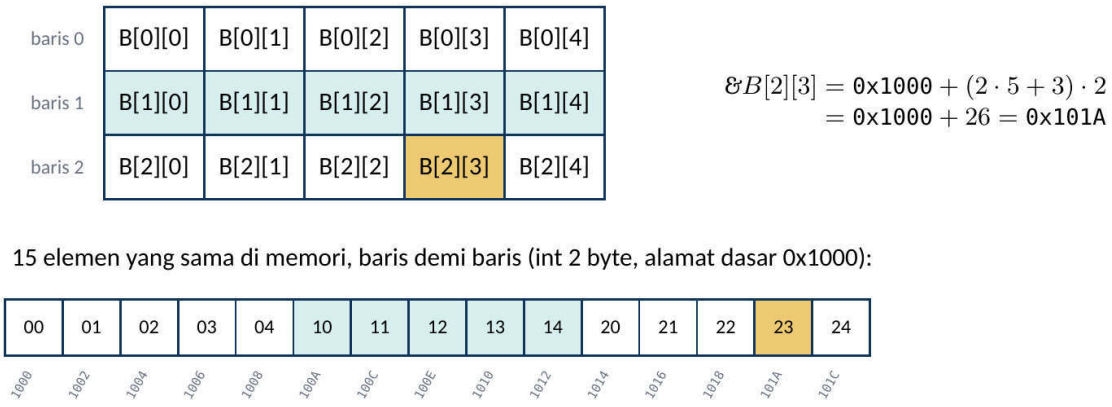
```
call numbers: 3 8 12 15 21 26 30 34 41 47 52 58 63 70 77 85
n = 16, block size m = 4
key 52: index 10   jump 3 + walk 3 = 6 probes
key 58: index 11   jump 3 + walk 0 = 3 probes
key 60: not found  jump 4 + walk 1 = 5 probes
key  3: index 0    jump 1 + walk 1 = 2 probes
key 90: not found  jump 4 + walk 0 = 4 probes
worst case n/m + m - 1 for the best m = floor(sqrt(n)):
  n =      16  m =    4  worst =    7 probes
  n =     100  m =   10  worst =   19 probes
  n =   10000  m =  100  worst =  199 probes
  n = 1000000  m = 1000  worst = 1999 probes
```

Pencarian 52 adalah **Gambar 2.7**: tiga lompatan, tiga langkah. Pencarian 58 berhenti pada lompatan ketiganya, karena 58 adalah elemen terakhir blok 3. Pencarian 60 melompat empat kali, melangkah sekali ke 63, lalu berhenti karena 63 sudah terlalu besar. Pencarian 90 melompat melewati ujung larik. Tabel di bagian bawah menghitung Persamaan (2.9).

2.9 Pencarian pada Larik Dua Dimensi

2.9.1 Letak larik dua dimensi di memori

Rak-rak Pustaka Ceria membentuk sebuah kisi: tiga rak, masing-masing lima slot. Di C++ itu adalah `int B[3][5]`, larik dua dimensi dengan 3 **baris** dan 5 **kolom**. Namun, memori itu satu dimensi: satu deretan panjang byte. C dan C++ menyimpan larik dua dimensi dalam **urutan baris-mayor** (*row-major order*): seluruh baris 0, lalu seluruh baris 1, lalu seluruh baris 2, seperti diperlihatkan **Gambar 2.8**.



Gambar 2.8 Larik 3 × 5 dan tata letak baris-mayornya di memori. Setiap sel memori diberi label baris dan kolom elemen yang dimuatnya; angka kecil adalah alamatnya bila sebuah `int` menempati 2 byte dan larik dimulai di 0x1000.

Untuk mencapai $B[i][j]$, mesin harus melewati i baris utuh yang masing-masing berisi C elemen, lalu j elemen lagi. Jadi elemen itu bernomor $i \cdot C + j$ dalam deretan, dan

$$address(B[i][j]) = base + (i \cdot C + j) \cdot w, \tag{2.10}$$

dengan C banyak kolom dan w lebar satu elemen. Persamaan (2.10) merampatkan Persamaan (2.1): indeks dua dimensi diubah menjadi indeks satu dimensi, lalu diperlakukan persis seperti sebelumnya. Fortran, MATLAB, dan R memakai konvensi lainnya, urutan **kolom-mayor**, yang menukar peran i dan j: $base + (j \cdot R + i) \cdot w$, dengan R banyak baris.

CONTOH: Misalkan `int A[10]` dimulai di alamat 0x1000 pada kompiler yang `int`-nya menempati 2 byte (seperti kompiler 16-bit tahun 1990-an). Maka $\&A[7] = 0\text{x}1000 + 7 \cdot 2 = 0\text{x}1000 + 14$. Empat belas adalah E dalam heksadesimal, jadi $\&A[7] = 0\text{x}100\text{E}$. Sekarang misalkan `int B[3][5]` dimulai di 0x1000. Untuk $B[2][3]$: lewati 2 baris berisi 5, lalu 3: elemen bernomor $2 \cdot 5 + 3 = 13$, pada selisih byte $13 \cdot 2 = 26 = 0\text{x}1\text{A}$, jadi $\&B[2][3] = 0\text{x}101\text{A}$. Pada kompiler masa kini sebuah `int` berukuran 4 byte, dan kedua alamat tadi menjadi $0\text{x}1000 + 28 = 0\text{x}101\text{C}$ dan $0\text{x}1000 + 52 = 0\text{x}1034$. Kekeliruan yang sering terjadi adalah menambahkan *banyak elemen* alih-alih selisih byte — menulis $0\text{x}1000 + 7$ — sehingga faktor w terlupa.

2.9.2 Tiga cara mencari di sebuah kisi

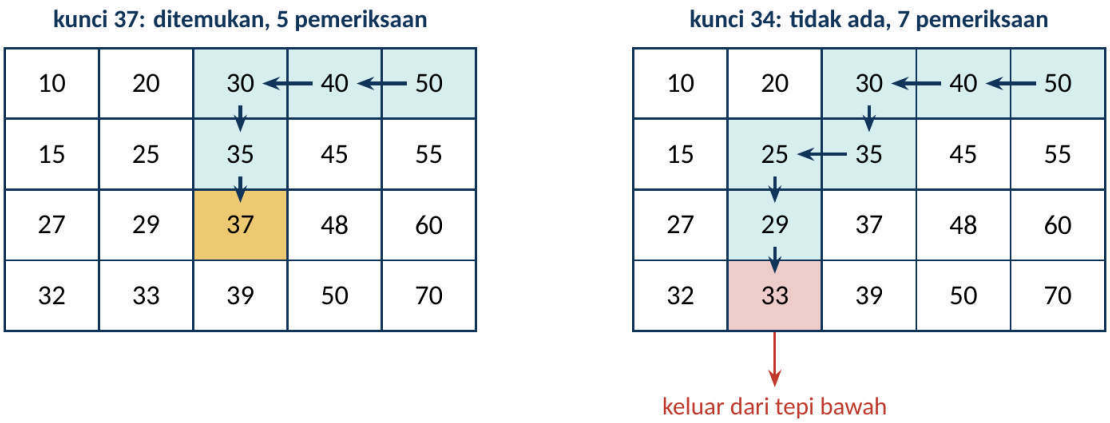
Kisi tidak terurut: telusuri. Jika kisi tidak berurutan, tidak ada yang dapat dimanfaatkan: kunjungi baris satu per satu dan slot setiap baris satu per satu. Ini adalah pencarian sekuensial pada $R \cdot C$ elemen, $O(RC)$ pemeriksaan. Menelusuri baris demi baris — gelung dalam berjalan atas kolom — mengunjungi memori menurut urutan tata letaknya pada **Gambar 2.8**, yang juga merupakan urutan tercepat bagi *cache* prosesor.

Baris dan kolom sama-sama terurut: metode tangga. Andaikan setiap baris naik dari kiri ke kanan *dan* setiap kolom naik dari atas ke bawah — tata letak sebuah tabel yang nomor rak dan nomor slotnya sama-sama mengikuti kode. Kisi seperti itu tidak terurut sebagai satu deret, sehingga pencarian biner tidak dapat langsung dipakai, tetapi ada metode $O(R + C)$ yang indah. Mulailah di pojok **kanan atas**. Jika elemen di sana sama dengan kunci, berhentilah. Jika ia *lebih besar* daripada kunci, kunci tidak mungkin berada di bawahnya pada kolom itu (semua yang di bawah lebih besar lagi), jadi bergeserlah satu langkah ke kiri. Jika ia *lebih kecil*, kunci tidak mungkin berada di kirinya pada baris itu (semua yang di kiri lebih kecil lagi), jadi turunlah satu langkah. Setiap langkah membuang satu baris utuh atau satu kolom utuh, sehingga setelah paling banyak $R + C - 1$ langkah pencarian menemukan kunci atau keluar dari kisi.

Pseudokode 2.8 `STAIRCASE-SEARCH`: pada kisi yang baris dan kolomnya sama-sama menaik, setiap pemeriksaan menyingkirkan satu baris utuh atau satu kolom utuh.

```
STAIRCASE-SEARCH(M, R, C, x):     // baris dan kolom sama-sama menaik
 1.  r ← 0
 2.  c ← C − 1                    // mulai di pojok kanan atas
 3.  while r < R and c ≥ 0
 4.      if M[r][c] = x
 5.          return (r, c)
 6.      else if M[r][c] > x
 7.          c ← c − 1            // sisa kolom c lebih besar lagi
 8.      else
 9.          r ← r + 1            // sisa baris r lebih kecil lagi
10.  return (−1, −1)
```

Gambar 2.9 menelusurinya untuk kunci yang ada dan kunci yang tidak ada.



Gambar 2.9 Metode tangga pada kisi 4×5 yang baris dan kolomnya menaik. Untuk 37, lintasan 50, 40, 30, 35, 37 menemukan kunci dalam lima pemeriksaan; untuk 34, lintasan bergeser ke kiri dan ke bawah sampai keluar dari kisi di bawah 33.

Seluruh kisi terurut dalam urutan baris-mayor: pencarian biner. Jika elemen terakhir setiap baris lebih kecil daripada elemen pertama baris berikutnya, kisi itu, dibaca dalam urutan baris-mayor, adalah satu deret terurut berisi $R \cdot C$ elemen. Carilah secara biner sebagai satu larik berindeks $0..RC-1$, dengan mengubah setiap indeks k kembali menjadi baris dan kolom melalui $r = \lfloor k/C \rfloor$ dan $c = k \bmod C$ — Persamaan (2.10) yang dijalankan terbalik. Biayanya $\lfloor \lg(RC) \rfloor + 1$ pemeriksaan.

2.9.3 Programnya

Kode 2.19 menghitung alamat-alamat pada contoh di atas, menelusuri kisi kode rak yang tidak terurut, menjalankan metode tangga untuk 37 dan 34, dan melakukan pencarian biner pada kisi yang terurut baris-mayor. **Kode 2.20** dan **Kode 2.21** adalah versi Java dan Python-nya.

Kode 2.19 `search_2d.cpp`: alamat pada larik dua dimensi, penelusuran, metode tangga, dan pencarian biner dalam urutan baris-mayor.

```cpp
// Pencarian pada larik dua dimensi: alamat, penelusuran biasa, metode tangga,
// dan pencarian biner pada matriks yang terurut dalam urutan baris-mayor.
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

using Matrix = vector<vector<int>>;

long address(long base, int width, int index) {
    return base + (long)index * width;
}

// Baris demi baris, slot demi slot: berlaku untuk kisi apa pun.
bool scan(const Matrix& grid, int key, int& row, int& col, int& probes) {
    probes = 0;
    for (size_t r = 0; r < grid.size(); r++)
        for (size_t c = 0; c < grid[r].size(); c++) {
            probes++;
            if (grid[r][c] == key) {
                row = (int)r;
                col = (int)c;
                return true;
            }
        }
    return false;
}

// Baris dan kolom menaik: mulai di kanan atas, geser ke kiri atau turun.
bool staircase(const Matrix& m, int key, int& row, int& col, vector<int>& path) {
    int r = 0, c = (int)m[0].size() - 1;
    while (r < (int)m.size() && c >= 0) {
        path.push_back(m[r][c]);
        if (m[r][c] == key) {
            row = r;
            col = c;
            return true;
        }
        if (m[r][c] > key)
            c--;  // semua yang di bawahnya pada kolom ini lebih besar lagi
        else
            r++;  // semua yang di kirinya pada baris ini lebih kecil lagi
    }
    return false;
}
```

Kode 2.19 (Lanjutan) *search_2d.cpp: alamat pada larik dua dimensi, penelusuran, metode tangga, dan pencarian biner dalam urutan baris-mayor.*

```
// Seluruh matriks adalah satu deret terurut dalam urutan baris-mayor.
bool binary2d(const Matrix& m, int key, int& row, int& col, int& probes) {
    int rows = (int)m.size(), cols = (int)m[0].size();
    int low = 0, high = rows * cols - 1;
    probes = 0;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        int r = mid / cols, c = mid % cols;  // indeks k -> (baris, kolom)
        probes++;
        if (m[r][c] == key) {
            row = r;
            col = c;
            return true;
        }
        if (m[r][c] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return false;
}

int main() {
    long base = 0x1000;
    for (int width : {2, 4}) {
        long a7 = address(base, width, 7);
        long b23 = address(base, width, 2 * 5 + 3);
        printf("int of %d bytes: &A[7] = 0x%lX, &B[2][3] = 0x%lX\n", width, a7,
               b23);
    }

    Matrix shelves = {
            {72, 25, 96, 41, 88}, {34, 17, 63, 56, 12}, {80, 45, 29, 91, 7}};
    int r, c, p;
    for (int key : {63, 50}) {
        if (scan(shelves, key, r, c, p))
            printf("scan for %d: shelf %d, slot %d after %d probes\n", key, r, c,
                   p);
        else
            printf("scan for %d: not found after %d probes\n", key, p);
    }
```

Kode 2.19 (Lanjutan) *search_2d.cpp: alamat pada larik dua dimensi, penelusuran, metode tangga, dan pencarian biner dalam urutan baris-mayor.*

```
    Matrix m = {{10, 20, 30, 40, 50},
                {15, 25, 35, 45, 55},
                {27, 29, 37, 48, 60},
                {32, 33, 39, 50, 70}};
    for (int key : {37, 34}) {
        vector<int> path;
        bool found = staircase(m, key, r, c, path);
        string where = found ? "at (" + to_string(r) + ", " + to_string(c) + ")"
                             : "not found";
        string steps;
        for (size_t k = 0; k < path.size(); k++)
            steps += (k ? " " : "") + to_string(path[k]);
        printf("staircase for %d: %s, path %s (%zu probes)\n", key, where.c_str(),
               steps.c_str(), path.size());
    }

    Matrix s = {{2, 5, 9, 14}, {17, 23, 31, 38}, {42, 47, 55, 61}};
    for (int key : {31, 40}) {
        bool found = binary2d(s, key, r, c, p);
        string where = found ? "at (" + to_string(r) + ", " + to_string(c) + ")"
                             : "not found";
        printf("row-major binary search for %d: %s after %d probes\n", key,
               where.c_str(), p);
    }
    return 0;
}
```

Kode 2.20 `Search2D.java`: program yang sama dalam Java.

```java
// Pencarian pada larik dua dimensi: alamat, penelusuran biasa, metode tangga,
// dan pencarian biner pada matriks yang terurut dalam urutan baris-mayor.
import java.util.ArrayList;
import java.util.List;
import java.util.StringJoiner;

public class Search2D {
    static long address(long base, int width, int index) {
        return base + (long) index * width;
    }

    // Baris demi baris: berlaku untuk kisi apa pun. Hasil: {baris, kolom, periksa}.
    static int[] scan(int[][] grid, int key) {
        int probes = 0;
        for (int r = 0; r < grid.length; r++)
            for (int c = 0; c < grid[r].length; c++) {
                probes++;
                if (grid[r][c] == key) return new int[] {r, c, probes};
            }
        return new int[] {-1, -1, probes};
    }

    // Baris dan kolom menaik: mulai di kanan atas, geser ke kiri atau turun.
    // Mengembalikan {baris, kolom}; nilai yang diperiksa ditambahkan ke path.
    static int[] staircase(int[][] m, int key, List<Integer> path) {
        int r = 0, c = m[0].length - 1;
        while (r < m.length && c >= 0) {
            path.add(m[r][c]);
            if (m[r][c] == key) return new int[] {r, c};
            if (m[r][c] > key)
                c--;  // semua yang di bawahnya pada kolom ini lebih besar lagi
            else
                r++;  // semua yang di kirinya pada baris ini lebih kecil lagi
        }
        return new int[] {-1, -1};
    }
```

Kode 2.20 (Lanjutan) *Search2D.java: program yang sama dalam Java.*

```
    // Matriks = satu deret terurut baris-mayor. Hasil: {baris, kolom, periksa}.
    static int[] binary2d(int[][] m, int key) {
        int rows = m.length, cols = m[0].length;
        int low = 0, high = rows * cols - 1, probes = 0;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            int r = mid / cols, c = mid % cols;  // indeks k -> (baris, kolom)
            probes++;
            if (m[r][c] == key) return new int[] {r, c, probes};
            if (m[r][c] < key)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return new int[] {-1, -1, probes};
    }

    public static void main(String[] args) {
        long base = 0x1000;
        for (int width : new int[] {2, 4}) {
            long a7 = address(base, width, 7);
            long b23 = address(base, width, 2 * 5 + 3);
            System.out.printf("int of %d bytes: &A[7] = 0x%X, &B[2][3] = 0x%X%n",
                    width, a7, b23);
        }

        int[][] shelves = {
                {72, 25, 96, 41, 88}, {34, 17, 63, 56, 12}, {80, 45, 29, 91, 7}};
        for (int key : new int[] {63, 50}) {
            int[] r = scan(shelves, key);
            if (r[0] >= 0)
                System.out.printf(
                        "scan for %d: shelf %d, slot %d after %d probes%n", key,
                        r[0], r[1], r[2]);
            else
                System.out.printf(
                        "scan for %d: not found after %d probes%n", key, r[2]);
        }
```

Kode 2.20 (Lanjutan) *Search2D.java: program yang sama dalam Java.*

```
        int[][] m = {{10, 20, 30, 40, 50}, {15, 25, 35, 45, 55},
                {27, 29, 37, 48, 60}, {32, 33, 39, 50, 70}};
        for (int key : new int[] {37, 34}) {
            List<Integer> path = new ArrayList<>();
            int[] r = staircase(m, key, path);
            String where =
                    r[0] >= 0 ? "at (" + r[0] + ", " + r[1] + ")" : "not found";
            StringJoiner steps = new StringJoiner(" ");
            for (int x : path) steps.add(String.valueOf(x));
            System.out.printf("staircase for %d: %s, path %s (%d probes)%n", key,
                    where, steps, path.size());
        }

        int[][] s = {{2, 5, 9, 14}, {17, 23, 31, 38}, {42, 47, 55, 61}};
        for (int key : new int[] {31, 40}) {
            int[] r = binary2d(s, key);
            String where =
                    r[0] >= 0 ? "at (" + r[0] + ", " + r[1] + ")" : "not found";
            System.out.printf(
                    "row-major binary search for %d: %s after %d probes%n", key,
                    where, r[2]);
        }
    }
}
```

Kode 2.21 `search_2d.py`: program yang sama dalam Python, dengan daftar berisi daftar sebagai kisinya.

```python
"""Pencarian pada larik dua dimensi: alamat, penelusuran biasa, metode tangga,
dan pencarian biner pada matriks yang terurut dalam urutan baris-mayor."""


def address(base, width, index):
    return base + index * width


def scan(grid, key):
    """Baris demi baris, untuk kisi apa pun. Hasil: (baris, kolom, periksa)."""
    probes = 0
    for r in range(len(grid)):
        for c in range(len(grid[r])):
            probes += 1
            if grid[r][c] == key:
                return r, c, probes
    return -1, -1, probes


def staircase(m, key):
    """Baris dan kolom menaik: mulai di kanan atas, geser ke kiri atau turun."""
    r, c = 0, len(m[0]) - 1
    path = []
    while r < len(m) and c >= 0:
        path.append(m[r][c])
        if m[r][c] == key:
            return r, c, path
        if m[r][c] > key:
            c -= 1  # semua yang di bawahnya pada kolom ini lebih besar lagi
        else:
            r += 1  # semua yang di kirinya pada baris ini lebih kecil lagi
    return -1, -1, path


def binary_2d(m, key):
    """Seluruh matriks adalah satu deret terurut dalam urutan baris-mayor."""
    rows, cols = len(m), len(m[0])
    low, high = 0, rows * cols - 1
    probes = 0
    while low <= high:
        mid = low + (high - low) // 2
        r, c = mid // cols, mid % cols  # indeks k -> (baris, kolom)
        probes += 1
        if m[r][c] == key:
            return r, c, probes
        if m[r][c] < key:
```

Kode 2.21 (Lanjutan) *search_2d.py: program yang sama dalam Python, dengan daftar berisi daftar sebagai kisinya.*

```
            low = mid + 1
        else:
            high = mid - 1
    return -1, -1, probes


def main():
    base = 0x1000
    for width in (2, 4):
        a7 = address(base, width, 7)
        b23 = address(base, width, 2 * 5 + 3)
        print("int of %d bytes: &A[7] = 0x%X, &B[2][3] = 0x%X" % (width, a7, b23))

    shelves = [[72, 25, 96, 41, 88], [34, 17, 63, 56, 12], [80, 45, 29, 91, 7]]
    for key in (63, 50):
        r, c, p = scan(shelves, key)
        if r >= 0:
            print("scan for %d: shelf %d, slot %d after %d probes" % (key, r, c, p))
        else:
            print("scan for %d: not found after %d probes" % (key, p))

    m = [
        [10, 20, 30, 40, 50],
        [15, 25, 35, 45, 55],
        [27, 29, 37, 48, 60],
        [32, 33, 39, 50, 70],
    ]
    for key in (37, 34):
        r, c, path = staircase(m, key)
        where = "at (%d, %d)" % (r, c) if r >= 0 else "not found"
        print(
            "staircase for %d: %s, path %s (%d probes)"
            % (key, where, " ".join(str(x) for x in path), len(path))
        )

    s = [[2, 5, 9, 14], [17, 23, 31, 38], [42, 47, 55, 61]]
    for key in (31, 40):
        r, c, p = binary_2d(s, key)
        where = "at (%d, %d)" % (r, c) if r >= 0 else "not found"
        print(
            "row-major binary search for %d: %s after %d probes" % (key, where, p)
        )
```

Kode 2.21 (Lanjutan) *search_2d.py: program yang sama dalam Python, dengan daftar berisi daftar sebagai kisinya.*

```
if __name__ == "__main__":
    main()
```

Ketiganya mencetak:

```
int of 2 bytes: &A[7] = 0x100E, &B[2][3] = 0x101A
int of 4 bytes: &A[7] = 0x101C, &B[2][3] = 0x1034
scan for 63: shelf 1, slot 2 after 8 probes
scan for 50: not found after 15 probes
staircase for 37: at (2, 2), path 50 40 30 35 37 (5 probes)
staircase for 34: not found, path 50 40 30 35 25 29 33 (7 probes)
row-major binary search for 31: at (1, 2) after 3 probes
row-major binary search for 40: not found after 4 probes
```

Penelusuran menemukan 63 di rak 1, slot 2, setelah 8 pemeriksaan — lima di baris 0 dan tiga di baris 1 — dan memerlukan kelima belasnya untuk memastikan 50 tidak ada. Lintasan metode tangga adalah lintasan pada **Gambar 2.9**. Pencarian biner untuk 31 memeriksa indeks 5, yaitu (1,1) yang memuat 23, lalu indeks 8, yaitu (2,0) yang memuat 42, lalu indeks 6, yaitu (1,2) yang memuat 31.

2.10 Mengukur pada Data Nyata

2.10.1 Sebuah katalog nyata

Menghitung pemeriksaan dengan tangan pada delapan buku mengajarkan mekanismenya; ia tidak memberi tahu kita apa yang terjadi pada skala besar. Karena itu kita sekarang meninggalkan rak komputer dan memberi Pustaka Ceria katalog sungguhan: metadata sepuluh ribu buku yang paling banyak diberi penilaian di situs Goodreads, yang diterbitkan sebagai himpunan data goodbooks-10k [8]. Dari himpunan itu kita menyimpan, untuk setiap buku, pengenal Goodreads-nya, tahun terbit pertamanya, banyak penilaiannya, judulnya, dan nama pengarangnya, dalam berkas berpemisah tab `catalog.tsv` yang disertakan bersama kode bab ini. Berkas itu mendaftar buku menurut urutan kepopulerannya, yang, dari sudut pandang pengenalnya, adalah urutan sembarang: ia memerankan urutan kedatangan.

Pengenal-pengenal itu nyata, dan data nyata punya bentuk. Nilainya berkisar dari 1 sampai 33.288.638, tetapi separuhnya di bawah 395.090 — buku-buku lama yang populer didaftarkan lebih awal, ketika nomornya masih kecil. Itu adalah sebaran yang sangat menumpuk, dan hal ini akan berpengaruh.

2.10.2 Percobaannya

Kode 2.22 memuat katalog, menyimpan pengenal menurut urutan berkas untuk kedua pencarian sekuensial, mengurutkan salinannya untuk tiga pencarian lainnya, dan menyiapkan dua himpunan kueri: 1.000 pengenal yang ada (setiap buku kesepuluh menurut urutan berkas) dan 905 pengenal yang tidak ada (satu lebih besar daripada setiap pengenal kesepuluh dalam urutan terurut, dengan melewatkan sedikit yang kebetulan sudah terpakai). Untuk masing-masing dari lima pencarian, program melaporkan rata-rata banyak pemeriksaan pada setiap himpunan dan waktu per pencarian, dengan mengambil hasil terbaik dari sembilan pengukuran agar gangguan dari program lain di komputer sesedikit mungkin memengaruhi hasilnya. Di bagian akhir, program memakai pencarian biner dan *lower bound* untuk dua pertanyaan sehari-hari.

Kode 2.22 `measure_catalog.cpp`: lima pencarian pada katalog sepuluh ribu buku sungguhan, dengan pemeriksaan dihitung dan waktu diukur.

```cpp
// Lima pencarian pada katalog nyata 10.000 buku: pemeriksaan dan waktu diukur.
#include <algorithm>
#include <chrono>
#include <cmath>
#include <cstdio>
#include <fstream>
#include <functional>
#include <numeric>
#include <sstream>
#include <string>
#include <unordered_set>
#include <vector>
using namespace std;

struct Book {
    long long id;
    int year;
    long long ratings;
    string title, writers;
};

struct Result {
    int index;
    int probes;
};

vector<Book> load(const string& path) {
    vector<Book> books;
    ifstream in(path);
    string line;
    getline(in, line);  // lewati baris judul kolom
    while (getline(in, line)) {
        if (!line.empty() && line.back() == '\r') line.pop_back();
        stringstream ss(line);
        string id, year, ratings;
        Book b;
        getline(ss, id, '\t');
        getline(ss, year, '\t');
        getline(ss, ratings, '\t');
        getline(ss, b.title, '\t');
        getline(ss, b.writers, '\t');
        b.id = stoll(id);
        b.year = stoi(year);
        b.ratings = stoll(ratings);
        books.push_back(b);
    }
```

Kode 2.22 (Lanjutan) *measure_catalog.cpp: lima pencarian pada katalog sepuluh ribu buku sungguhan, dengan pemeriksaan dihitung dan waktu diukur.*

```
    return books;
}

Result sequential(const vector<long long>& a, long long key) {
    for (size_t i = 0; i < a.size(); i++)
        if (a[i] == key) return {(int)i, (int)i + 1};
    return {-1, (int)a.size()};
}

// a punya satu sel cadangan a[n]
Result sentinel(vector<long long>& a, int n, long long key) {
    a[n] = key;
    int i = 0;
    while (a[i] != key) i++;
    return {i < n ? i : -1, i + 1};
}

Result binary(const vector<long long>& a, long long key) {
    int low = 0, high = (int)a.size() - 1, probes = 0;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        probes++;
        if (a[mid] == key) return {mid, probes};
        if (a[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return {-1, probes};
}

Result interpolation(const vector<long long>& a, long long key) {
    int low = 0, high = (int)a.size() - 1, probes = 0;
    while (low <= high && a[low] <= key && key <= a[high]) {
        int pos = low;
        if (a[high] != a[low])
            pos = low + (int)((double)(key - a[low]) / (double)(a[high] - a[low]) *
                              (high - low));
        probes++;
        if (a[pos] == key) return {pos, probes};
        if (a[pos] < key)
            low = pos + 1;
        else
            high = pos - 1;
    }
    return {-1, probes};
```

Kode 2.22 (Lanjutan) *measure_catalog.cpp: lima pencarian pada katalog sepuluh ribu buku sungguhan, dengan pemeriksaan dihitung dan waktu diukur.*

```
}

Result jump(const vector<long long>& a, long long key, int m) {
    int n = (int)a.size(), probes = 0;
    int start = 0, end = min(m, n);
    while (true) {
        probes++;
        if (a[end - 1] == key) return {end - 1, probes};
        if (a[end - 1] > key) break;
        start = end;
        end = min(end + m, n);
        if (start >= n) return {-1, probes};
    }
    for (int i = start; i < end - 1; i++) {
        probes++;
        if (a[i] == key) return {i, probes};
        if (a[i] > key) break;
    }
    return {-1, probes};
}

int lowerBound(const vector<int>& a, int key) {
    int low = 0, high = (int)a.size();
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (a[mid] < key)
            low = mid + 1;
        else
            high = mid;
    }
    return low;
}

int main() {
    vector<Book> books = load("catalog.tsv");
    int n = (int)books.size();
    // id menurut urutan di dalam berkas
    vector<long long> arrival;
    for (const Book& b : books) arrival.push_back(b.id);
    vector<long long> padded = arrival;
    // satu sel cadangan untuk sentinel
    padded.push_back(0);
    vector<int> byId(n);
    iota(byId.begin(), byId.end(), 0);
    sort(byId.begin(), byId.end(),
         [&](int x, int y) { return books[x].id < books[y].id; });
```

Kode 2.22 (Lanjutan) *measure_catalog.cpp: lima pencarian pada katalog sepuluh ribu buku sungguhan, dengan pemeriksaan dihitung dan waktu diukur.*

```
    vector<long long> ids;  // id dalam urutan menaik
    for (int i : byId) ids.push_back(books[i].id);
    vector<long long> present, absent;
    for (int i = 0; i < n; i += 10) present.push_back(arrival[i]);
    unordered_set<long long> known(arrival.begin(), arrival.end());
    for (int j = 5; j < n; j += 10)
        if (!known.count(ids[j] + 1)) absent.push_back(ids[j] + 1);
    int m = (int)sqrt((double)n);
    printf("loaded %d books; ids from %lld to %lld, median %lld\n", n, ids[0],
           ids[n - 1], ids[n / 2]);
    printf("queries: %zu present ids, %zu absent ids\n", present.size(),
           absent.size());

    vector<pair<string, function<Result(long long)>>> methods = {
            {"sequential (file order)",
             [&](long long k) { return sequential(arrival, k); }},
            {"sentinel (file order)",
             [&](long long k) { return sentinel(padded, n, k); }},
            {"binary (sorted ids)", [&](long long k) { return binary(ids, k); }},
            {"interpolation (sorted ids)",
             [&](long long k) { return interpolation(ids, k); }},
            {"jump, m = " + to_string(m) + " (sorted ids)",
             [&](long long k) { return jump(ids, k, m); }}};
    printf("%-28s %10s %10s %14s\n", "method", "present", "absent",
           "ns per search");
    // agar pengoptimal tidak membuang perhitungan
    volatile long long sink = 0;
    for (auto& [name, search] : methods) {
        double averages[2];
        const vector<long long>* sets[2] = {&present, &absent};
        for (int s = 0; s < 2; s++) {
            long long total = 0;
            for (long long k : *sets[s]) total += search(k).probes;
            averages[s] = (double)total / sets[s]->size();
        }
        long long best = -1;
        for (int run = 0; run < 9; run++) {  // terbaik dari sembilan pengukuran
            auto t0 = chrono::steady_clock::now();
            for (long long k : present) sink = sink + search(k).index;
            for (long long k : absent) sink = sink + search(k).index;
            long long elapsed = chrono::duration_cast<chrono::nanoseconds>(
                                    chrono::steady_clock::now() - t0)
                                    .count();
            best = (best < 0) ? elapsed : min(best, elapsed);
        }
        double per = (double)best / (present.size() + absent.size());
```

Kode 2.22 (Lanjutan) *measure_catalog.cpp: lima pencarian pada katalog sepuluh ribu buku sungguhan, dengan pemeriksaan dihitung dan waktu diukur.*

```
        printf("%-28s %10.1f %10.1f %14.0f\n", name.c_str(), averages[0],
               averages[1], per);
    }

    Result r = binary(ids, 3);
    printf("id 3 -> %s (%d probes)\n", books[byId[r.index]].title.c_str(),
           r.probes);
    vector<int> years;
    for (const Book& b : books) years.push_back(b.year);
    sort(years.begin(), years.end());
    int first = lowerBound(years, 2005), after = lowerBound(years, 2006);
    printf("first published in 2005: %d books (sorted positions %d..%d)\n",
           after - first, first, after - 1);
    printf("first published 1990-1999: %d books\n",
           lowerBound(years, 2000) - lowerBound(years, 1990));
    return 0;
}
```

Kode 2.23 `MeasureCatalog.java`: percobaan yang sama dalam Java.

```
// Lima pencarian pada katalog nyata 10.000 buku: pemeriksaan dan waktu diukur.
import java.io.BufferedReader;
import java.io.IOException;
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.function.LongFunction;

public class MeasureCatalog {
    record Book(long id, int year, long ratings, String title, String writers) {}

    static List<Book> load(String path) throws IOException {
        List<Book> books = new ArrayList<>();
        try (BufferedReader in = Files.newBufferedReader(
                    Path.of(path), StandardCharsets.UTF_8)) {
            in.readLine();  // lewati baris judul kolom
            String line;
            while ((line = in.readLine()) != null) {
                String[] f = line.split("\t", -1);
                books.add(new Book(Long.parseLong(f[0]), Integer.parseInt(f[1]),
                        Long.parseLong(f[2]), f[3], f[4]));
            }
        }
        return books;
    }

    // Bulatkan nilai biner eksak x, seperti printf C dan % Python.
    static String fixed(double x, int digits) {
        return new BigDecimal(x)
                .setScale(digits, RoundingMode.HALF_EVEN)
                .toPlainString();
    }
```

Kode 2.23 (Lanjutan) *MeasureCatalog.java: percobaan yang sama dalam Java.*

```
    // Setiap pencarian mengembalikan {indeks atau -1, pemeriksaan}.
    static int[] sequential(long[] a, long key) {
        for (int i = 0; i < a.length; i++)
            if (a[i] == key) return new int[] {i, i + 1};
        return new int[] {-1, a.length};
    }

    // a punya satu sel cadangan a[n]
    static int[] sentinel(long[] a, int n, long key) {
        a[n] = key;
        int i = 0;
        while (a[i] != key) i++;
        return new int[] {i < n ? i : -1, i + 1};
    }

    static int[] binary(long[] a, long key) {
        int low = 0, high = a.length - 1, probes = 0;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            probes++;
            if (a[mid] == key) return new int[] {mid, probes};
            if (a[mid] < key)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return new int[] {-1, probes};
    }

    static int[] interpolation(long[] a, long key) {
        int low = 0, high = a.length - 1, probes = 0;
        while (low <= high && a[low] <= key && key <= a[high]) {
            int pos = low;
            if (a[high] != a[low])
                pos = low +
                        (int) ((double) (key - a[low]) /
                                (double) (a[high] - a[low]) * (high - low));
            probes++;
            if (a[pos] == key) return new int[] {pos, probes};
            if (a[pos] < key)
                low = pos + 1;
            else
                high = pos - 1;
        }
        return new int[] {-1, probes};
    }
```

Kode 2.23 (Lanjutan) *MeasureCatalog.java: percobaan yang sama dalam Java.*

```
    static int[] jump(long[] a, long key, int m) {
        int n = a.length, probes = 0;
        int start = 0, end = Math.min(m, n);
        while (true) {
            probes++;
            if (a[end - 1] == key) return new int[] {end - 1, probes};
            if (a[end - 1] > key) break;
            start = end;
            end = Math.min(end + m, n);
            if (start >= n) return new int[] {-1, probes};
        }
        for (int i = start; i < end - 1; i++) {
            probes++;
            if (a[i] == key) return new int[] {i, probes};
            if (a[i] > key) break;
        }
        return new int[] {-1, probes};
    }

    static int lowerBound(int[] a, int key) {
        int low = 0, high = a.length;
        while (low < high) {
            int mid = low + (high - low) / 2;
            if (a[mid] < key)
                low = mid + 1;
            else
                high = mid;
        }
        return low;
    }

    public static void main(String[] args) throws IOException {
        List<Book> books = load("catalog.tsv");
        int n = books.size();
        // id menurut urutan di dalam berkas
        long[] arrival = new long[n];
        for (int i = 0; i < n; i++) arrival[i] = books.get(i).id();
        // satu sel cadangan untuk sentinel
        long[] padded = Arrays.copyOf(arrival, n + 1);
        Integer[] byId = new Integer[n];
        for (int i = 0; i < n; i++) byId[i] = i;
        Arrays.sort(byId, Comparator.comparingLong(i -> books.get(i).id()));
        // id dalam urutan menaik
        long[] ids = new long[n];
        for (int i = 0; i < n; i++) ids[i] = books.get(byId[i]).id();
```

Kode 2.23 (Lanjutan) *MeasureCatalog.java: percobaan yang sama dalam Java.*

```
        List<Long> presentList = new ArrayList<>(), absentList = new ArrayList<>();
        for (int i = 0; i < n; i += 10) presentList.add(arrival[i]);
        Set<Long> known = new HashSet<>();
        for (long id : arrival) known.add(id);
        for (int j = 5; j < n; j += 10)
            if (!known.contains(ids[j] + 1)) absentList.add(ids[j] + 1);
        long[] present = presentList.stream().mapToLong(Long::longValue).toArray();
        long[] absent = absentList.stream().mapToLong(Long::longValue).toArray();
        int m = (int) Math.sqrt(n);
        System.out.printf("loaded %d books; ids from %d to %d, median %d%n", n,
                ids[0], ids[n - 1], ids[n / 2]);
        System.out.printf("queries: %d present ids, %d absent ids%n",
                present.length, absent.length);

        String[] names = {"sequential (file order)", "sentinel (file order)",
                "binary (sorted ids)", "interpolation (sorted ids)",
                "jump, m = " + m + " (sorted ids)"};
        List<LongFunction<int[]>> methods = List.of(k
                -> sequential(arrival, k),
                k
                -> sentinel(padded, n, k),
                k
                -> binary(ids, k),
                k -> interpolation(ids, k), k -> jump(ids, k, m));
        System.out.printf("%-28s %10s %10s %14s%n", "method", "present", "absent",
                "ns per search");
        // agar kompiler JIT tidak membuang perhitungan
        long sink = 0;
        for (int t = 0; t < names.length; t++) {
            LongFunction<int[]> search = methods.get(t);
            double[] averages = new double[2];
            long[][] sets = {present, absent};
            for (int s = 0; s < 2; s++) {
                long total = 0;
                for (long k : sets[s]) total += search.apply(k)[1];
                averages[s] = (double) total / sets[s].length;
            }
            long best = -1;
            // terbaik dari sembilan pengukuran
            for (int run = 0; run < 9; run++) {
                long t0 = System.nanoTime();
                for (long k : present) sink += search.apply(k)[0];
                for (long k : absent) sink += search.apply(k)[0];
                long elapsed = System.nanoTime() - t0;
                best = best < 0 ? elapsed : Math.min(best, elapsed);
            }
```

Kode 2.23 (Lanjutan) *MeasureCatalog.java: percobaan yang sama dalam Java.*

```
            double per = (double) best / (present.length + absent.length);
            System.out.printf("%-28s %10s %10s %14s%n", names[t],
                    fixed(averages[0], 1), fixed(averages[1], 1), fixed(per, 0));
        }
        if (sink == Long.MIN_VALUE) System.out.println(sink);

        int[] r = binary(ids, 3);
        System.out.printf(
                "id 3 -> %s (%d probes)%n", books.get(byId[r[0]]).title(), r[1]);
        int[] years = new int[n];
        for (int i = 0; i < n; i++) years[i] = books.get(i).year();
        Arrays.sort(years);
        int first = lowerBound(years, 2005), after = lowerBound(years, 2006);
        System.out.printf(
                "first published in 2005: %d books (sorted positions %d..%d)%n",
                after - first, first, after - 1);
        System.out.printf("first published 1990-1999: %d books%n",
                lowerBound(years, 2000) - lowerBound(years, 1990));
    }
}
```

Kode 2.24 `measure_catalog.py`: percobaan yang sama dalam Python.

```
"""Lima pencarian pada katalog nyata 10.000 buku: pemeriksaan dan waktu diukur."""

import math
import time


def load(path):
    books = []
    with open(path, encoding="utf-8") as f:
        f.readline()  # lewati baris judul kolom
        for line in f:
            book_id, year, ratings, title, writers = line.rstrip("\n").split("\t")
            books.append((int(book_id), int(year), int(ratings), title, writers))
    return books


def sequential(a, key):
    for i in range(len(a)):
        if a[i] == key:
            return i, i + 1
    return -1, len(a)


def sentinel(a, n, key):  # a punya satu sel cadangan a[n]
    a[n] = key
    i = 0
    while a[i] != key:
        i += 1
    return (i if i < n else -1), i + 1


def binary(a, key):
    low, high, probes = 0, len(a) - 1, 0
    while low <= high:
        mid = low + (high - low) // 2
        probes += 1
        if a[mid] == key:
            return mid, probes
        if a[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1, probes
```

Kode 2.24 (Lanjutan) *measure_catalog.py: percobaan yang sama dalam Python.*

```
def interpolation(a, key):
    low, high, probes = 0, len(a) - 1, 0
    while low <= high and a[low] <= key <= a[high]:
        if a[high] == a[low]:
            pos = low
        else:
            pos = low + int(
                float(key - a[low]) / float(a[high] - a[low]) * (high - low)
            )
        probes += 1
        if a[pos] == key:
            return pos, probes
        if a[pos] < key:
            low = pos + 1
        else:
            high = pos - 1
    return -1, probes


def jump(a, key, m):
    n = len(a)
    probes = 0
    start, end = 0, min(m, n)
    while True:
        probes += 1
        if a[end - 1] == key:
            return end - 1, probes
        if a[end - 1] > key:
            break
        start, end = end, min(end + m, n)
        if start >= n:
            return -1, probes
    for i in range(start, end - 1):
        probes += 1
        if a[i] == key:
            return i, probes
        if a[i] > key:
            break
    return -1, probes
```

Kode 2.24 (Lanjutan) *measure_catalog.py: percobaan yang sama dalam Python.*

```
def lower_bound(a, key):
    low, high = 0, len(a)
    while low < high:
        mid = low + (high - low) // 2
        if a[mid] < key:
            low = mid + 1
        else:
            high = mid
    return low


def main():
    books = load("catalog.tsv")
    n = len(books)
    # id menurut urutan di dalam berkas
    arrival = [b[0] for b in books]
    # satu sel cadangan untuk sentinel
    padded = arrival + [0]
    by_id = sorted(range(n), key=lambda i: books[i][0])
    ids = [books[i][0] for i in by_id]  # id dalam urutan menaik
    present = [arrival[i] for i in range(0, n, 10)]
    known = set(arrival)
    absent = [ids[j] + 1 for j in range(5, n, 10) if ids[j] + 1 not in known]
    m = math.isqrt(n)
    print(
        "loaded %d books; ids from %d to %d, median %d"
        % (n, ids[0], ids[-1], ids[n // 2])
    )
    print("queries: %d present ids, %d absent ids" % (len(present), len(absent)))

    methods = [
        ("sequential (file order)", lambda k: sequential(arrival, k)),
        ("sentinel (file order)", lambda k: sentinel(padded, n, k)),
        ("binary (sorted ids)", lambda k: binary(ids, k)),
        ("interpolation (sorted ids)", lambda k: interpolation(ids, k)),
        ("jump, m = %d (sorted ids)" % m, lambda k: jump(ids, k, m)),
    ]
    print("%-28s %10s %10s %14s" % ("method", "present", "absent", "ns per search"))
    for name, search in methods:
        averages = []
        for queries in (present, absent):
            total = 0
            for k in queries:
                total += search(k)[1]
            averages.append(total / len(queries))
        best = None
```

Kode 2.24 (Lanjutan) *measure_catalog.py: percobaan yang sama dalam Python.*

```
        for run in range(9):  # terbaik dari sembilan pengukuran
            t0 = time.perf_counter_ns()
            for k in present:
                search(k)
            for k in absent:
                search(k)
            elapsed = time.perf_counter_ns() - t0
            best = elapsed if best is None else min(best, elapsed)
        per = best / (len(present) + len(absent))
        print("%-28s %10.1f %10.1f %14.0f" % (name, averages[0], averages[1], per))

    i, probes = binary(ids, 3)
    print("id 3 -> %s (%d probes)" % (books[by_id[i]][3], probes))
    years = sorted(b[1] for b in books)
    first, after = lower_bound(years, 2005), lower_bound(years, 2006)
    print(
        "first published in 2005: %d books (sorted positions %d..%d)"
        % (after - first, first, after - 1)
    )
    print(
        "first published 1990-1999: %d books"
        % (lower_bound(years, 2000) - lower_bound(years, 1990))
    )


if __name__ == "__main__":
    main()
```

Program C++, yang dikompilasi dengan `g++ -std=c++17 -O2`, mencetak:

```
loaded 10000 books; ids from 1 to 33288638, median 395090
queries: 1000 present ids, 905 absent ids
method                          present     absent  ns per search
sequential (file order)          4996.0    10000.0           2243
sentinel (file order)            4996.0    10001.0           2248
binary (sorted ids)                12.4       13.3             89
interpolation (sorted ids)        211.2      191.8           2612
jump, m = 100 (sorted ids)         99.2      105.6            130
id 3 -> Harry Potter and the Sorcerer's Stone (Harry Potter, #1) (13 probes)
first published in 2005: 326 books (sorted positions 5067..5392)
first published 1990-1999: 1360 books
```

Program Java mencetak hitungan pemeriksaan yang sama dan waktu yang berbeda:

```
loaded 10000 books; ids from 1 to 33288638, median 395090
queries: 1000 present ids, 905 absent ids
method                          present     absent  ns per search
sequential (file order)          4996.0    10000.0           1481
sentinel (file order)            4996.0    10001.0           1487
binary (sorted ids)                12.4       13.3             74
interpolation (sorted ids)        211.2      191.8           3414
jump, m = 100 (sorted ids)         99.2      105.6            214
id 3 -> Harry Potter and the Sorcerer's Stone (Harry Potter, #1) (13 probes)
first published in 2005: 326 books (sorted positions 5067..5392)
first published 1990-1999: 1360 books
```

Dan program Python:

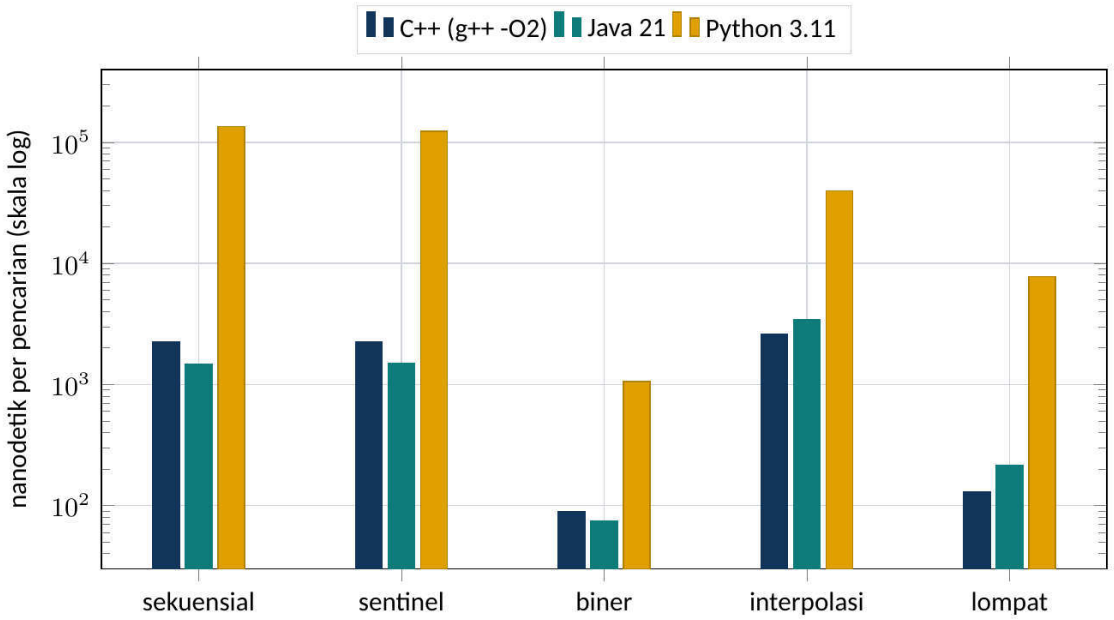
```
loaded 10000 books; ids from 1 to 33288638, median 395090
queries: 1000 present ids, 905 absent ids
method                          present     absent  ns per search
sequential (file order)          4996.0    10000.0         135376
sentinel (file order)            4996.0    10001.0         123270
binary (sorted ids)                12.4       13.3           1054
interpolation (sorted ids)        211.2      191.8          39667
jump, m = 100 (sorted ids)         99.2      105.6           7782
id 3 -> Harry Potter and the Sorcerer's Stone (Harry Potter, #1) (13 probes)
first published in 2005: 326 books (sorted positions 5067..5392)
first published 1990-1999: 1360 books
```

Ketiga pengukuran dijalankan berturut-turut di komputer yang sama, sebuah mesin virtual dengan dua inti prosesor Intel Xeon 2,1 GHz, pada September 2026, dengan g++ 13.3, OpenJDK 21, dan CPython 3.11. Di komputer Anda setiap waktu akan berbeda; hitungan pemeriksaannya akan persis sama. **Tabel 2.3** merangkum hasilnya.

Tabel 2.3 Biaya terukur kelima pencarian pada katalog 10.000 buku: rata-rata pemeriksaan per pencarian, dan nanodetik per pencarian dirata-ratakan atas seluruh 1.905 kueri (terbaik dari sembilan pengukuran). Angka desimal ditulis dengan titik, sama seperti keluaran program.

Pencarian	Pemeriksaan, ada	Pemeriksaan, tidak ada	Ns, C++	Ns, Java	Ns, Python
Sekuensial (urutan berkas)	4996.0	10000.0	2243	1481	135376
Sentinel (urutan berkas)	4996.0	10001.0	2248	1487	123270
Biner (terurut)	12.4	13.3	89	74	1054
Interpolasi (terurut)	211.2	191.8	2612	3414	39667
Lompat, $m = 100$ (terurut)	99.2	105.6	130	214	7782

Gambar 2.10 menggambar ketiga kolom waktu itu pada skala logaritmik, yang membuat rasio yang sama tampak sebagai jarak yang sama.



Gambar 2.10 Nanodetik per pencarian pada katalog 10.000 buku, menurut metode pencarian dan bahasa, pada skala logaritmik. Diukur oleh **Kode 2.22**, **Kode 2.23**, dan **Kode 2.24** pada September 2026.

2.10.3 Apa kata angka-angka itu

Hitungan pemeriksaan cocok dengan teori. Pencarian sekuensial rata-rata memerlukan 4.996 pemeriksaan pada kunci yang ada — Persamaan (2.2) meramalkan $(n + 1)/2 = 5{,}000.5$ untuk

kunci di posisi acak; kueri kita adalah setiap posisi kesepuluh, yang rata-ratanya tepat 4,996 — dan tepat $n = 10{,}000$ pada kunci yang tidak ada. Versi sentinel memerlukan 10.001 pada kunci yang tidak ada: pemeriksaan tambahan itu adalah sentinel itu sendiri. Pencarian biner rata-rata 12,4 pada kunci yang ada dan 13,3 pada yang tidak ada, tidak pernah lebih dari $\lfloor \lg 10^4 \rfloor + 1 = 14$ menurut Persamaan (2.5). Pencarian lompat dengan $m = \lfloor \sqrt{10^4} \rfloor = 100$ rata-rata sekitar 100, separuh kasus terburuk pada Persamaan (2.9), seperti diharapkan untuk kunci yang tersebar di semua blok.

Pencarian interpolasi gagal pada pengenal nyata. 211 pemeriksaan, sedangkan pencarian biner memerlukan 12. Lihat kembali angka-angka di Subbab 2.10.1: ketika pencarian dimulai, garis lurusnya membentang dari pengenal 1 ke pengenal 33.288.638, tetapi separuh buku berpengenal di bawah 395.090, sehingga buku yang pengenalnya, katakanlah, 400.000 berada di sekitar posisi 5.000, sedangkan garis itu meramalkan posisi $\lfloor 400{,}000/33{,}288{,}638 \cdot 9{,}999 \rfloor = 120$. Pemeriksaan mendarat ribuan tempat terlalu awal, interval hanya menyusut sedikit dari kiri, dan taksiran berikutnya kembali terlalu awal. Inilah kemerosotan yang diperingatkan Subbab 2.7.4, terjadi pada data nyata.

Pemeriksaan bukan waktu. Di C++, pencarian biner sekitar 25 kali lebih cepat daripada pencarian sekuensial, bukan hampir 600 kali seperti yang disiratkan hitungan pemeriksaan, dan pencarian interpolasi, dengan pemeriksaan sekitar 36 kali lebih sedikit daripada pencarian sekuensial, justru *lebih lambat*. Tiga efek menjelaskannya. Pertama, penelusuran sekuensial membaca memori secara berurutan, yang diambil lebih dahulu (*prefetch*) oleh prosesor, sehingga setiap pemeriksaan hanya memakan sepersekian nanodetik; pemeriksaan biner dan interpolasi melompat ke sana kemari di memori. Kedua, percabangan pada penelusuran sekuensial — "bukan yang ini, bukan yang ini…" — mudah diramalkan, sedangkan setiap percabangan pencarian biner ibarat lempar koin yang separuh waktunya ditebak salah oleh prosesor. Ketiga, setiap pemeriksaan interpolasi melakukan pembagian titik-mengambang, salah satu instruksi aritmetika paling lambat. Hitungan pemeriksaan memberi tahu bagaimana sebuah algoritma *berskala*; hanya jam yang memberi tahu seberapa cepat ia berjalan di mesin tertentu.

Sentinel tidak membantu di C++ maupun Java. 2.243 lawan 2.248 nanodetik di C++, 1.481 lawan 1.487 di Java: selisihnya lebih kecil daripada variasi antarpengukuran. Kompiler pengoptimal dan peramal percabangan prosesor sudah membuat uji batas itu nyaris gratis. Di Python, tempat setiap uji adalah instruksi yang ditafsirkan, sentinel menghemat sekitar 9%.

Bahasa berpengaruh sebesar faktor konstanta; algoritma berpengaruh sebesar orde besaran. Python sekitar 60 kali lebih lambat daripada C++ untuk pencarian sekuensial dan sekitar 12 kali lebih lambat untuk pencarian biner, karena Python membayar biaya penafsiran tetap pada setiap pemeriksaan. Namun, pencarian biner di Python (1.054 ns) mengalahkan pencarian sekuensial di C++ (2.243 ns). Algoritma yang lebih baik dalam bahasa yang lebih lambat berlari lebih cepat daripada algoritma yang lebih buruk dalam bahasa yang lebih cepat — dan selisihnya makin lebar ketika n tumbuh.

2.10.4 Bagaimana hitungan pemeriksaan tumbuh

Untuk melihat pertumbuhan, kita perlu mengubah-ubah n, sedangkan katalog nyata hanya punya satu ukuran. Karena itu **Kode 2.25** membangkitkan sendiri kunci-kunci terurut dengan pembangkit bilangan acak semu yang kecil dan terdefinisi lengkap, SplitMix64, dengan benih 2026, sehingga ketiga bahasa menghasilkan kunci yang sama dan mencetak tabel yang sama. Untuk setiap n dari 10^3 sampai 10^6 program melakukan 200 pencarian berhasil dengan setiap metode. Program juga membangun larik kedua berisi kunci **kubik**, $0,1,8,27,\ldots,i^3$, yang

berdesakan di ujung bawah dan merenggang di ujung atas, untuk memperlihatkan pencarian interpolasi pada data yang menumpuk. **Kode 2.26** dan **Kode 2.27** mengulang percobaan itu dalam Java dan Python.

Kode 2.25 `scaling.cpp`: rata-rata pemeriksaan ketika n tumbuh, pada kunci seragam yang dibangkitkan dan pada kunci kubik.

```cpp
// Pertumbuhan banyak pemeriksaan terhadap n, pada data terurut yang kita buat
// sendiri dengan benih tetap, agar C++, Java, dan Python mencetak tabel sama.
#include <algorithm>
#include <cmath>
#include <cstdint>
#include <cstdio>
#include <vector>
using namespace std;

// Pembangkit acak semu kecil yang terkenal, dengan aritmetika 64-bit.
struct SplitMix64 {
    uint64_t state;
    explicit SplitMix64(uint64_t seed) : state(seed) {}
    uint64_t next() {
        state += 0x9E3779B97F4A7C15ULL;
        uint64_t z = state;
        z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9ULL;
        z = (z ^ (z >> 27)) * 0x94D049BB133111EBULL;
        return z ^ (z >> 31);
    }
    uint64_t below(uint64_t bound) { return next() % bound; }
};

using Keys = vector<int64_t>;

int sequential(const Keys& a, int64_t key) {
    for (size_t i = 0; i < a.size(); i++)
        if (a[i] == key) return (int)i + 1;
    return (int)a.size();
}

int binary(const Keys& a, int64_t key) {
    int low = 0, high = (int)a.size() - 1, probes = 0;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        probes++;
        if (a[mid] == key) return probes;
        if (a[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return probes;
}
```

Kode 2.25 (Lanjutan) *scaling.cpp: rata-rata pemeriksaan ketika n tumbuh, pada kunci seragam yang dibangkitkan dan pada kunci kubik.*

```
int interpolation(const Keys& a, int64_t key) {
    int low = 0, high = (int)a.size() - 1, probes = 0;
    while (low <= high && a[low] <= key && key <= a[high]) {
        int pos = low;
        if (a[high] != a[low])
            pos = low + (int)((double)(key - a[low]) / (double)(a[high] - a[low]) *
                              (high - low));
        probes++;
        if (a[pos] == key) return probes;
        if (a[pos] < key)
            low = pos + 1;
        else
            high = pos - 1;
    }
    return probes;
}

int isqrt(int n) {
    int m = (int)sqrt((double)n);
    while (m * m > n) m--;
    while ((m + 1) * (m + 1) <= n) m++;
    return m;
}

int jump(const Keys& a, int64_t key) {
    int n = (int)a.size(), m = isqrt(n), probes = 0;
    int start = 0, end = min(m, n);
    while (true) {
        probes++;
        if (a[end - 1] == key) return probes;
        if (a[end - 1] > key) break;
        start = end;
        end = min(end + m, n);
        if (start >= n) return probes;
    }
    for (int i = start; i < end - 1; i++) {
        probes++;
        if (a[i] == key || a[i] > key) break;
    }
    return probes;
}
```

Kode 2.25 (Lanjutan) *scaling.cpp: rata-rata pemeriksaan ketika n tumbuh, pada kunci seragam yang dibangkitkan dan pada kunci kubik.*

```
int main() {
    const int queries = 200;
    printf("average probes over %d successful searches\n", queries);
    printf("%8s %10s %7s %6s %9s %9s %7s\n", "n", "sequential", "binary", "lg n",
           "interp-U", "interp-C", "jump");
    for (int n : {1000, 10000, 100000, 1000000}) {
        SplitMix64 rng(2026);
        Keys uniform(n), cubic(n);
        for (int i = 0; i < n; i++) uniform[i] = (int64_t)rng.below(10ULL * n);
        sort(uniform.begin(), uniform.end());
        // kunci berdesakan di ujung bawah
        for (int i = 0; i < n; i++) cubic[i] = (int64_t)i * i * i;
        vector<int> picks(queries);
        for (int& p : picks) p = (int)rng.below(n);
        long long seq = 0, bi = 0, iu = 0, ic = 0, ju = 0;
        for (int p : picks) {
            seq += sequential(uniform, uniform[p]);
            bi += binary(uniform, uniform[p]);
            iu += interpolation(uniform, uniform[p]);
            ic += interpolation(cubic, cubic[p]);
            ju += jump(uniform, uniform[p]);
        }
        printf("%8d %10.1f %7.2f %6.2f %9.2f %9.2f %7.1f\n", n,
               (double)seq / queries, (double)bi / queries, log2((double)n),
               (double)iu / queries, (double)ic / queries, (double)ju / queries);
    }
    return 0;
}
```

Kode 2.26 `Scaling.java`: percobaan yang sama dalam Java. Fungsi bantu `fixed` membulatkan nilai biner eksaknya, seperti yang dilakukan C dan Python, sehingga ketiga tabel sama sampai ke angka terakhir.

```java
// Pertumbuhan banyak pemeriksaan terhadap n, pada data terurut yang kita buat
// sendiri dengan benih tetap, agar C++, Java, dan Python mencetak tabel sama.
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.Arrays;

public class Scaling {
    // Pembangkit acak semu kecil yang terkenal, dengan aritmetika 64-bit.
    static final class SplitMix64 {
        private long state;
        SplitMix64(long seed) { state = seed; }
        long next() {
            state += 0x9E3779B97F4A7C15L;
            long z = state;
            z = (z ^ (z >>> 30)) * 0xBF58476D1CE4E5B9L;
            z = (z ^ (z >>> 27)) * 0x94D049BB133111EBL;
            return z ^ (z >>> 31);
        }
        long below(long bound) { return Long.remainderUnsigned(next(), bound); }
    }

    // Bulatkan nilai biner eksak x, seperti printf C dan % Python.
    static String fixed(double x, int digits) {
        return new BigDecimal(x)
                .setScale(digits, RoundingMode.HALF_EVEN)
                .toPlainString();
    }

    static int sequential(long[] a, long key) {
        for (int i = 0; i < a.length; i++)
            if (a[i] == key) return i + 1;
        return a.length;
    }

    static int binary(long[] a, long key) {
        int low = 0, high = a.length - 1, probes = 0;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            probes++;
            if (a[mid] == key) return probes;
            if (a[mid] < key)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return probes;
```

Kode 2.26 (Lanjutan) *Scaling.java: percobaan yang sama dalam Java. Fungsi bantu fixed membulatkan nilai biner eksaknya, seperti yang dilakukan C dan Python, sehingga ketiga tabel sama sampai ke angka terakhir.*

```
    }

    static int interpolation(long[] a, long key) {
        int low = 0, high = a.length - 1, probes = 0;
        while (low <= high && a[low] <= key && key <= a[high]) {
            int pos = low;
            if (a[high] != a[low])
                pos = low +
                        (int) ((double) (key - a[low]) /
                                (double) (a[high] - a[low]) * (high - low));
            probes++;
            if (a[pos] == key) return probes;
            if (a[pos] < key)
                low = pos + 1;
            else
                high = pos - 1;
        }
        return probes;
    }

    static int isqrt(int n) {
        int m = (int) Math.sqrt(n);
        while (m * m > n) m--;
        while ((m + 1) * (m + 1) <= n) m++;
        return m;
    }

    static int jump(long[] a, long key) {
        int n = a.length, m = isqrt(n), probes = 0;
        int start = 0, end = Math.min(m, n);
        while (true) {
            probes++;
            if (a[end - 1] == key) return probes;
            if (a[end - 1] > key) break;
            start = end;
            end = Math.min(end + m, n);
            if (start >= n) return probes;
        }
        for (int i = start; i < end - 1; i++) {
            probes++;
            if (a[i] == key || a[i] > key) break;
        }
        return probes;
    }
```

Kode 2.26 (Lanjutan) *Scaling.java: percobaan yang sama dalam Java. Fungsi bantu fixed membulatkan nilai biner eksaknya, seperti yang dilakukan C dan Python, sehingga ketiga tabel sama sampai ke angka terakhir.*

```
    public static void main(String[] args) {
        int queries = 200;
        System.out.printf("average probes over %d successful searches%n", queries);
        System.out.printf("%8s %10s %7s %6s %9s %9s %7s%n", "n", "sequential",
                "binary", "lg n", "interp-U", "interp-C", "jump");
        for (int n : new int[] {1000, 10000, 100000, 1000000}) {
            SplitMix64 rng = new SplitMix64(2026);
            long[] uniform = new long[n], cubic = new long[n];
            for (int i = 0; i < n; i++) uniform[i] = rng.below(10L * n);
            Arrays.sort(uniform);
            // kunci berdesakan di ujung bawah
            for (int i = 0; i < n; i++) cubic[i] = (long) i * i * i;
            int[] picks = new int[queries];
            for (int q = 0; q < queries; q++) picks[q] = (int) rng.below(n);
            long seq = 0, bi = 0, iu = 0, ic = 0, ju = 0;
            for (int p : picks) {
                seq += sequential(uniform, uniform[p]);
                bi += binary(uniform, uniform[p]);
                iu += interpolation(uniform, uniform[p]);
                ic += interpolation(cubic, cubic[p]);
                ju += jump(uniform, uniform[p]);
            }
            System.out.printf("%8d %10s %7s %6s %9s %9s %7s%n", n,
                    fixed((double) seq / queries, 1),
                    fixed((double) bi / queries, 2),
                    fixed(Math.log(n) / Math.log(2), 2),
                    fixed((double) iu / queries, 2),
                    fixed((double) ic / queries, 2),
                    fixed((double) ju / queries, 1));
        }
    }
}
```

Kode 2.27 `scaling.py`: percobaan yang sama dalam Python.

```
"""Pertumbuhan banyak pemeriksaan terhadap n, pada data terurut yang kita buat
sendiri dengan benih tetap, agar C++, Java, dan Python mencetak tabel sama."""

import math

MASK = (1 << 64) - 1


class SplitMix64:
    """Pembangkit acak semu kecil yang terkenal, dengan aritmetika 64-bit."""

    def __init__(self, seed):
        self.state = seed

    def next(self):
        self.state = (self.state + 0x9E3779B97F4A7C15) & MASK
        z = self.state
        z = ((z ^ (z >> 30)) * 0xBF58476D1CE4E5B9) & MASK
        z = ((z ^ (z >> 27)) * 0x94D049BB133111EB) & MASK
        return z ^ (z >> 31)

    def below(self, bound):
        return self.next() % bound


def sequential(a, key):
    for i in range(len(a)):
        if a[i] == key:
            return i + 1
    return len(a)


def binary(a, key):
    low, high, probes = 0, len(a) - 1, 0
    while low <= high:
        mid = low + (high - low) // 2
        probes += 1
        if a[mid] == key:
            return probes
        if a[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return probes
```

Kode 2.27 (Lanjutan) *scaling.py: percobaan yang sama dalam Python.*

```
def interpolation(a, key):
    low, high, probes = 0, len(a) - 1, 0
    while low <= high and a[low] <= key <= a[high]:
        if a[high] == a[low]:
            pos = low
        else:
            pos = low + int(
                float(key - a[low]) / float(a[high] - a[low]) * (high - low)
            )
        probes += 1
        if a[pos] == key:
            return probes
        if a[pos] < key:
            low = pos + 1
        else:
            high = pos - 1
    return probes


def jump(a, key):
    n = len(a)
    m = math.isqrt(n)
    probes = 0
    start, end = 0, min(m, n)
    while True:
        probes += 1
        if a[end - 1] == key:
            return probes
        if a[end - 1] > key:
            break
        start, end = end, min(end + m, n)
        if start >= n:
            return probes
    for i in range(start, end - 1):
        probes += 1
        if a[i] == key or a[i] > key:
            break
    return probes
```

Kode 2.27 (Lanjutan) *scaling.py: percobaan yang sama dalam Python.*

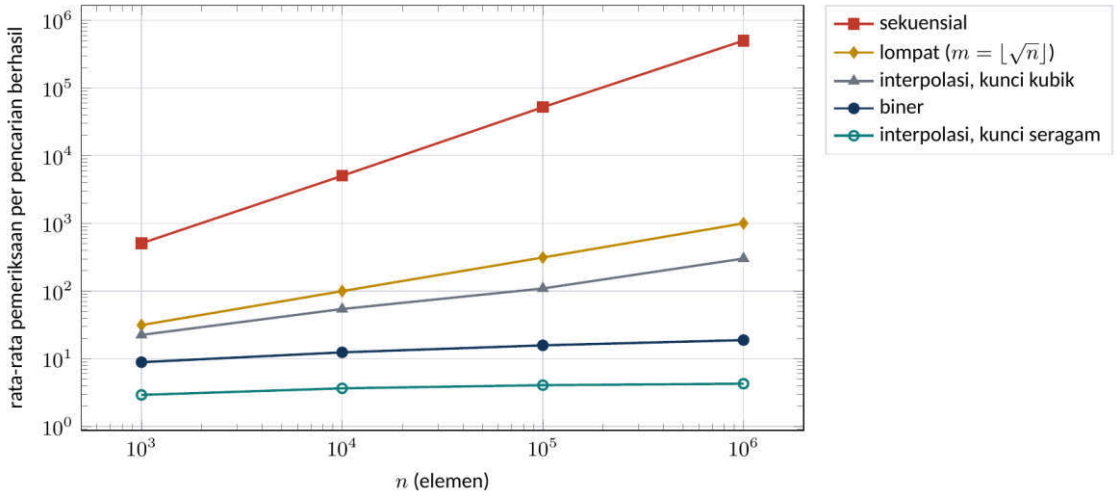
```
def main():
    queries = 200
    print("average probes over %d successful searches" % queries)
    print(
        "%8s %10s %7s %6s %9s %9s %7s"
        % ("n", "sequential", "binary", "lg n", "interp-U", "interp-C", "jump")
    )
    for n in (1000, 10000, 100000, 1000000):
        rng = SplitMix64(2026)
        uniform = sorted(rng.below(10 * n) for _ in range(n))
        # kunci berdesakan di ujung bawah
        cubic = [i * i * i for i in range(n)]
        picks = [rng.below(n) for _ in range(queries)]
        seq = sum(sequential(uniform, uniform[p]) for p in picks) / queries
        bi = sum(binary(uniform, uniform[p]) for p in picks) / queries
        iu = sum(interpolation(uniform, uniform[p]) for p in picks) / queries
        ic = sum(interpolation(cubic, cubic[p]) for p in picks) / queries
        ju = sum(jump(uniform, uniform[p]) for p in picks) / queries
        print(
            "%8d %10.1f %7.2f %6.2f %9.2f %9.2f %7.1f"
            % (n, seq, bi, math.log2(n), iu, ic, ju)
        )


if __name__ == "__main__":
    main()
```

Ketiganya mencetak:

```
average probes over 200 successful searches
       n sequential  binary   lg n  interp-U  interp-C    jump
    1000      506.2    8.86   9.97      2.91     22.41    31.3
   10000     5032.7   12.38  13.29      3.64     54.13    99.5
  100000    51919.3   15.72  16.61      4.05    108.86   312.3
 1000000   500932.9   18.79  19.93      4.26    301.42   999.3
```

Gambar 2.11 memetakan tabel itu pada sumbu logaritmik, tempat besaran yang sebanding dengan n^k tampak sebagai garis lurus berkemiringan k.



Gambar 2.11 Rata-rata pemeriksaan per pencarian berhasil ketika n tumbuh dari 10^3 sampai 10^6, pada sumbu logaritmik. Pencarian sekuensial naik dengan kemiringan 1, pencarian lompat dengan kemiringan $1/2$; pencarian biner naik sekitar 3,3 pemeriksaan untuk setiap kenaikan sepuluh kali lipat; pencarian interpolasi pada kunci seragam nyaris tidak naik, tetapi pada kunci kubik ia mendaki curam.

Setiap kurva berperilaku seperti yang diramalkan analisisnya. Pencarian sekuensial menjadi sepuluh kali lipat ketika n menjadi sepuluh kali lipat. Pencarian lompat menjadi sekitar $\sqrt{10} \approx 3.16$ kali lipat. Pencarian biner bertambah sekitar $\lg 10 \approx 3.32$ pemeriksaan per langkah sepuluh kali lipat dan tetap sedikit di bawah $\lg n$, kolom yang dicetak di sebelahnya. Pencarian interpolasi pada kunci seragam bergerak dari 2,91 ke 4,26 sementara n tumbuh seribu kali lipat — perilaku $\lg\lg n$ dari Subbab 2.7.4 — dan pada kunci kubik ia tumbuh dari 22 ke 301, sudah melampaui pencarian biner pada $n = 1{,}000$.

2.11 Pencarian dengan Pustaka Standar

Sekarang Anda tahu cara kerja pencarian-pencarian ini; dalam kode produksi, Anda hampir selalu sebaiknya memanggil versi di pustaka standar bahasa Anda, yang sudah diuji oleh jutaan program. Namun, Anda harus tahu persis apa yang dijanjikan masing-masing, karena mereka berbeda dalam hal-hal kecil yang menimbulkan bug besar. **Tabel 2.4** menjajarkannya.

Tabel 2.4 Pencarian dengan pustaka standar C++, Java, dan Python.

Pertanyaan	C++ (`<algorithm>`)	Java (`java.util`)	Python
Pencarian linear	`std::find` → iterator, `end()` bila tidak ada	`list.indexOf(x)` → indeks atau −1	`x in list`; `list.index(x)` melempar `ValueError`
Ada di data terurut?	`std::binary_search` → `bool`	`Arrays.binarySearch` → ≥ 0 bila ditemukan	`i = bisect_left(a, x)`, lalu `a[i] == x`
Posisi pertama $\geq x$	`std::lower_bound`	`-(r) - 1` bila $r < 0$ (lihat teks)	`bisect.bisect_left`
Posisi pertama $> x$	`std::upper_bound`	—	`bisect.bisect_right`
Semua salinan x	`std::equal_range`	—	`bisect_left` … `bisect_right`
Sisipkan, tetap terurut	`v.insert(upper_bound(…), x)`	`list.add(-(r) - 1, x)`	`bisect.insort(a, x)`
Cari rekaman menurut satu medan	lambda pembanding	`Collections.binarySearch(list, key, comparator)`	argumen `key=` (Python 3.10+)

2.11.1 C++

Header `<algorithm>` C++ menyediakan `std::find` untuk pencarian linear, dan empat pencarian biner yang semuanya menerima rentang terurut `[first, last)` — konvensi setengah terbuka dari Subbab 2.6. `std::binary_search` hanya menjawab ya atau tidak. `std::lower_bound` dan `std::upper_bound` mengembalikan iterator ke posisi yang didefinisikan di Subbab 2.6.3; kurangkan `begin()` untuk mendapatkan indeks. `std::equal_range` mengembalikan keduanya sekaligus. Standar C++ menjamin paling banyak $\log_2(last - first) + O(1)$ perbandingan untuk masing-masing [9]. Keempatnya menerima pembanding opsional, dan dengan cara itulah Anda mencari larik rekaman menurut satu medan: pembanding menerima sebuah elemen dan kunci, lalu menyatakan apakah elemen itu mendahului kunci.

Sejak C++20 masing-masing juga tersedia di *namespace* `std::ranges`, yang menerima seluruh wadah, `std::ranges::lower_bound(v, x)`, dan sebuah *projection* yang memilih medan yang dibandingkan. GCC 16, yang dirilis pada April 2026, secara bawaan mengompilasi C++20 [10]; program-program buku ini memakai C++17 dan bentuk klasik agar dapat dikompilasi di setiap kompiler yang masih dipakai. **Kode 2.28** mencoba semuanya, termasuk pencarian rekaman menurut kode dengan pembanding.

Kode 2.28 `library_search.cpp`: `std::find`, `std::binary_search`, `std::lower_bound`, `std::upper_bound`, dan `std::equal_range`, termasuk pencarian rekaman menurut kode.

```cpp
// Pencarian dengan pustaka standar C++: find, binary_search, lower_bound,
// upper_bound, dan equal_range dari <algorithm>.
#include <algorithm>
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

struct Book {
    int code;
    string title;
};

void print(const char* label, const vector<int>& v) {
    printf("%s", label);
    for (int x : v) printf(" %d", x);
    printf("\n");
}

int main() {
    vector<int> arrival = {72, 25, 96, 41, 88, 34, 63, 56};
    auto it = find(arrival.begin(), arrival.end(), 63);  // penelusuran linear
    printf("find(63): index %td\n", it - arrival.begin());
    it = find(arrival.begin(), arrival.end(), 60);
    printf("find(60): %s\n", it == arrival.end() ? "end() -- not found" : "found");

    vector<int> codes = arrival;
    sort(codes.begin(), codes.end());
    print("sorted codes:", codes);
    printf("binary_search(88): %s\n",
           binary_search(codes.begin(), codes.end(), 88) ? "true" : "false");
    printf("binary_search(60): %s\n",
           binary_search(codes.begin(), codes.end(), 60) ? "true" : "false");
    auto lb = lower_bound(codes.begin(), codes.end(), 60);
    printf("lower_bound(60): index %td, where 60 would be inserted\n",
           lb - codes.begin());
```

Kode 2.28 (Lanjutan) *library_search.cpp: std::find, std::binary_search, std::lower_bound, std::upper_bound, dan std::equal_range, termasuk pencarian rekaman menurut kode.*

```
    vector<int> years = {1998, 2001, 2001, 2005, 2005,
                         2005, 2009, 2012, 2012, 2020};
    auto lo = lower_bound(years.begin(), years.end(), 2005);
    auto hi = upper_bound(years.begin(), years.end(), 2005);
    printf("2005: lower_bound = %td, upper_bound = %td, count = %td\n",
           lo - years.begin(), hi - years.begin(), hi - lo);
    auto range = equal_range(years.begin(), years.end(), 2012);
    printf("equal_range(2012) = [%td, %td)\n", range.first - years.begin(),
           range.second - years.begin());
    // tetap terurut
    years.insert(upper_bound(years.begin(), years.end(), 2003), 2003);
    print("after inserting 2003:", years);

    vector<Book> shelf = {
            {25, "The C++ Programming"},       {34, "Mastering Delphi 6"},
            {41, "Professional C#"},           {56, "Pure JavaScript v2"},
            {63, "Advanced JSP & Servlet"},    {72, "Calculus Make it Easy"},
            {88, "Visual Basic 2005 Express"}, {96, "Artificial Life : Volume 1"}};
    auto rec = lower_bound(shelf.begin(), shelf.end(), 63,
                           [](const Book& b, int code) { return b.code < code; });
    if (rec != shelf.end() && rec->code == 63)
        printf("record with code 063: %s\n", rec->title.c_str());
    return 0;
}
```

```
find(63): index 6
find(60): end() -- not found
sorted codes: 25 34 41 56 63 72 88 96
binary_search(88): true
binary_search(60): false
lower_bound(60): index 4, where 60 would be inserted
2005: lower_bound = 3, upper_bound = 6, count = 3
equal_range(2012) = [7, 9)
after inserting 2003: 1998 2001 2001 2003 2005 2005 2005 2009 2012 2012 2020
record with code 063: Advanced JSP & Servlet
```

2.11.2 Java

Java menyediakan `List.indexOf` dan `List.contains` untuk pencarian linear, serta dua pencarian biner: `Arrays.binarySearch` untuk larik dan `Collections.binarySearch` untuk daftar. Nilai kembaliannya memadatkan dua jawaban ke dalam satu `int` [11], [12]. Jika kunci ditemukan, hasilnya adalah indeksnya, yang tak negatif. Jika tidak ditemukan, hasilnya $-(ip) - 1$, dengan *ip*, *titik sisip*, adalah indeks elemen pertama yang lebih besar daripada kunci — yaitu *lower bound* kita. "-1" itu menjamin bahwa kunci yang tidak ada tidak pernah mengembalikan 0, yang akan terlihat seperti "ditemukan di indeks 0". Jadi hasil negatif r berarti "tidak ada, dan akan diletakkan di $-r - 1$".

Dua peringatan dari dokumentasinya perlu diulang. Jika larik tidak terurut, hasilnya tidak terdefinisi. Dan jika kunci muncul beberapa kali, "tidak ada jaminan salinan mana yang akan ditemukan" [11]: `Arrays.binarySearch` adalah **Pseudokode 2.3**, bukan *lower bound*. Java tidak punya `upper_bound`; bila Anda memerlukannya, sepuluh baris **Pseudokode 2.5** dengan `<=` adalah jawabannya. **Kode 2.29** memperlihatkan metode-metode itu bekerja, termasuk nilai kembalian negatifnya.

Kode 2.29 `LibrarySearch.java`: `indexOf`, `Arrays.binarySearch`, dan `Collections.binarySearch`, termasuk nilai kembalian negatif dan pencarian rekaman dengan pembanding.

```java
// Pencarian dengan pustaka Java: List.indexOf, Arrays.binarySearch, dan
// Collections.binarySearch.
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class LibrarySearch {
    record Book(int code, String title) {}

    public static void main(String[] args) {
        List<Integer> arrival = List.of(72, 25, 96, 41, 88, 34, 63, 56);
        // penelusuran linear
        System.out.println("arrival.contains(63): " + arrival.contains(63));
        // penelusuran linear
        System.out.println("arrival.indexOf(63): " + arrival.indexOf(63));
        System.out.println("arrival.indexOf(60): " + arrival.indexOf(60));

        int[] codes = {72, 25, 96, 41, 88, 34, 63, 56};
        Arrays.sort(codes);
        System.out.println("sorted codes: " + Arrays.toString(codes));
        System.out.println("Arrays.binarySearch(codes, 88) = " +
                Arrays.binarySearch(codes, 88));
        int r = Arrays.binarySearch(codes, 60);
        System.out.println("Arrays.binarySearch(codes, 60) = " + r +
                ", so 60 would be inserted at " + (-r - 1));

        int[] years = {1998, 2001, 2001, 2005, 2005, 2005, 2009, 2012, 2012, 2020};
        System.out.println("Arrays.binarySearch(years, 2005) = " +
                Arrays.binarySearch(years, 2005) +
                " (one of the 2005s, no promise which)");
        List<Integer> list = new ArrayList<>();
        for (int y : years) list.add(y);
        int pos = Collections.binarySearch(list, 2003);
        // daftar tetap terurut
        list.add(-pos - 1, 2003);
        System.out.println("after inserting 2003: " + list);
```

Kode 2.29 (Lanjutan) *LibrarySearch.java: indexOf, Arrays.binarySearch, dan Collections.binarySearch, termasuk nilai kembalian negatif dan pencarian rekaman dengan pembanding.*

```java
        List<Book> shelf = List.of(new Book(25, "The C++ Programming"),
                new Book(34, "Mastering Delphi 6"), new Book(41, "Professional C#"),
                new Book(56, "Pure JavaScript v2"),
                new Book(63, "Advanced JSP & Servlet"),
                new Book(72, "Calculus Make it Easy"),
                new Book(88, "Visual Basic 2005 Express"),
                new Book(96, "Artificial Life : Volume 1"));
        int i = Collections.binarySearch(
                shelf, new Book(63, ""), Comparator.comparingInt(Book::code));
        System.out.println("record with code 063: " + shelf.get(i).title());
    }
}
```

```
arrival.contains(63): true
arrival.indexOf(63): 6
arrival.indexOf(60): -1
sorted codes: [25, 34, 41, 56, 63, 72, 88, 96]
Arrays.binarySearch(codes, 88) = 6
Arrays.binarySearch(codes, 60) = -5, so 60 would be inserted at 4
Arrays.binarySearch(years, 2005) = 4 (one of the 2005s, no promise which)
after inserting 2003: [1998, 2001, 2001, 2003, 2005, 2005, 2005, 2009, 2012, 2012, 2020]
record with code 063: Advanced JSP & Servlet
```

Panggilan untuk 60 mengembalikan -5, jadi 60 akan disisipkan di $-(-5)-1=4$, di antara 56 dan 63. Panggilan untuk 2005 kebetulan mengembalikan indeks 4, salinan tengah — bukan yang pertama.

Program Java di buku ini ditulis untuk Java 25, rilis dukungan jangka panjang (LTS) yang berlaku saat ini, dan diuji dengan javac 21; Java 27, rilis fitur terbaru, terbit pada 15 September 2026 [13]. Tidak satu pun metode yang dipakai di sini berubah di antara versi-versi itu.

2.11.3 Python

Operator `in` dan `list.index` di Python melakukan pencarian linear; `list.index` melempar `ValueError` alih-alih mengembalikan -1. Untuk daftar terurut, modul `bisect` menyediakan `bisect_left` (*lower bound*) dan `bisect_right` (*upper bound*, juga tersedia sebagai `bisect`), masing-masing dengan argumen opsional `lo` dan `hi` untuk mencari pada sebagian daftar [14]. Sejak Python 3.10 masing-masing juga menerima fungsi `key`, dan dengan cara itulah Anda mencari daftar rekaman menurut satu medan: fungsi itu diterapkan pada elemen-elemen, bukan pada kunci pencarian, sehingga Anda memberikan nilai kunci apa adanya. `insort` menyisipkan sambil menjaga daftar tetap terurut; dokumentasinya mengingatkan bahwa pencarian $O(\lg n)$ itu "didominasi oleh langkah penyisipan $O(n)$ yang lambat", karena setiap elemen sesudahnya harus bergeser satu tempat ke kanan [14]. **Kode 2.30** mencoba masing-masing.

Kode 2.30 `library_search.py`: `in`, `list.index`, dan modul `bisect`, termasuk pencarian rekaman dengan `key=`.

```
"""Pencarian dengan perangkat bawaan Python: in, list.index, dan modul bisect."""

import bisect


def main():
    arrival = [72, 25, 96, 41, 88, 34, 63, 56]
    print("63 in arrival:", 63 in arrival)  # penelusuran linear, True/False
    print("arrival.index(63):", arrival.index(63))  # penelusuran linear, posisi
    try:
        arrival.index(60)
    except ValueError as e:
        print("arrival.index(60): ValueError:", e)

    codes = sorted(arrival)
    print("sorted codes:", codes)
    i = bisect.bisect_left(codes, 88)
    print(
        "bisect_left(codes, 88) = %d, found: %s"
        % (i, i < len(codes) and codes[i] == 88)
    )
    i = bisect.bisect_left(codes, 60)
    print(
        "bisect_left(codes, 60) = %d, found: %s"
        % (i, i < len(codes) and codes[i] == 60)
    )

    years = [1998, 2001, 2001, 2005, 2005, 2005, 2009, 2012, 2012, 2020]
    lo, hi = bisect.bisect_left(years, 2005), bisect.bisect_right(years, 2005)
    print(
        "2005: bisect_left = %d, bisect_right = %d, count = %d" % (lo, hi, hi - lo)
    )
    bisect.insort(years, 2003)  # daftar tetap terurut
    print("after insort(2003):", years)

    shelf = [
        (25, "The C++ Programming"),
        (34, "Mastering Delphi 6"),
        (41, "Professional C#"),
        (56, "Pure JavaScript v2"),
        (63, "Advanced JSP & Servlet"),
        (72, "Calculus Make it Easy"),
        (88, "Visual Basic 2005 Express"),
        (96, "Artificial Life : Volume 1"),
    ]
    i = bisect.bisect_left(shelf, 63, key=lambda book: book[0])  # Python 3.10+
```

Kode 2.30 (Lanjutan) *library_search.py: in, list.index, dan modul bisect, termasuk pencarian rekaman dengan key=.*

```
    print("record with code 063:", shelf[i][1])


if __name__ == "__main__":
    main()
```

```
63 in arrival: True
arrival.index(63): 6
arrival.index(60): ValueError: 60 is not in list
sorted codes: [25, 34, 41, 56, 63, 72, 88, 96]
bisect_left(codes, 88) = 6, found: True
bisect_left(codes, 60) = 4, found: False
2005: bisect_left = 3, bisect_right = 6, count = 3
after insort(2003): [1998, 2001, 2001, 2003, 2005, 2005, 2005, 2009, 2012, 2012, 2020]
record with code 063: Advanced JSP & Servlet
```

Ketiga program di subbab ini mencetak baris yang sedikit berbeda, karena ketiga pustaka melaporkan jawabannya dengan cara berbeda; setiap jawaban menyatakan fakta yang sama tentang data yang sama.

2.12 Memilih Metode Pencarian, dan Batas Pencarian dengan Perbandingan

2.12.1 Gambaran utuhnya

Tabel 2.5 merangkum bab ini.

Tabel 2.5 Metode-metode pencarian di bab ini, apa yang diperlukan masing-masing, dan berapa biayanya.

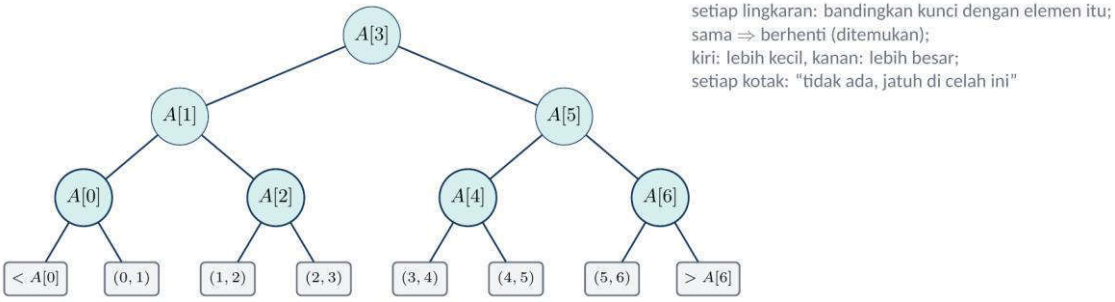
Metode	Memerlukan	Terbaik	Rata-rata	Terburuk	Ruang tambahan
Sekuensial	tidak ada	$O(1)$	$(n+1)/2$ pemeriksaan	n pemeriksaan	$O(1)$
Sentinel	satu sel cadangan yang dapat ditulisi	$O(1)$	$(n+1)/2$ pemeriksaan	$n+1$ pemeriksaan	$O(1)$
Biner	data terurut	$O(1)$	$\approx \lg n - 1$ pemeriksaan	$\lfloor \lg n \rfloor + 1$ pemeriksaan	$O(1)$ iteratif
Lower / upper bound	data terurut	$\approx \lg n$	$\approx \lg n$	$\lceil \lg(n+1) \rceil$ pemeriksaan	$O(1)$
Interpolasi	kunci numerik terurut yang tersebar merata	$O(1)$	$\approx \lg\lg n$ (seragam)	$O(n)$	$O(1)$
Lompat	data terurut	$O(1)$	$\approx \sqrt{n}$	$\approx 2\sqrt{n}$	$O(1)$
Tangga (2-D)	baris dan kolom terurut	$O(1)$	—	$R + C - 1$ pemeriksaan	$O(1)$

Memilih kemudian menjadi soal mengajukan pertanyaan secara berurutan. Apakah datanya terurut, atau dapat dijaga tetap terurut? Jika tidak, dan data itu hanya akan dicari beberapa kali, pakailah pencarian sekuensial; jika akan dicari berkali-kali, urutkan sekali (Bab 3) atau masukkan ke tabel *hash* (Bab 15). Jika terurut, pakailah pencarian biner — dalam bentuk *lower bound* jika duplikat atau rentang itu penting. Pertimbangkan pencarian interpolasi hanya jika kuncinya bilangan, Anda sudah *mengukur* bahwa kunci itu tersebar merata, dan pemeriksaan itu mahal (misalnya, setiap pemeriksaan adalah pembacaan cakram). Pertimbangkan pencarian lompat hanya bila bergerak mundur itu mahal.

2.12.2 Mengapa tidak ada algoritma berbasis perbandingan yang dapat mengalahkan pencarian biner

Mungkinkah algoritma berbasis perbandingan yang lebih cerdik mencari di larik terurut dengan kurang dari $\lfloor \lg n \rfloor + 1$ pemeriksaan pada kasus terburuk? Tidak — dan alasannya adalah argumen pencacahan yang akan kita jumpai lagi untuk pengurutan di Bab 3.

Setiap algoritma yang mempelajari larik hanya dengan membandingkan kunci dengan elemen-elemen dapat digambar sebagai **pohon keputusan** (*decision tree*). Setiap simpul dalam adalah perbandingan kunci dengan satu elemen, dengan tiga kemungkinan hasil: sama (berhenti, ditemukan), lebih kecil (ke kiri), lebih besar (ke kanan). Setiap daun adalah jawaban akhir "tidak ada, dan kunci jatuh di celah ini". **Gambar 2.12** memperlihatkan pohon pencarian biner untuk $n = 7$.



Gambar 2.12 Pohon keputusan pencarian biner pada tujuh elemen terurut. Setiap lingkaran membandingkan kunci dengan satu elemen; setiap kotak adalah satu dari delapan celah tempat kunci yang tidak ada dapat jatuh. Setiap lintasan dari akar ke kotak memuat tiga perbandingan.

Kunci yang tidak ada dapat jatuh di salah satu dari $n + 1$ celah: sebelum $A[0]$, di antara setiap pasangan bertetangga, dan sesudah $A[n-1]$. Algoritma yang benar harus mampu membedakan semuanya, sehingga pohonnya punya paling sedikit $n + 1$ daun. Pohon biner setinggi h punya paling banyak 2^h daun, jadi $2^h \geq n + 1$, dan dengan mengambil logaritmanya,

$$h \geq \lceil \lg(n+1) \rceil. \tag{2.11}$$

Tinggi pohon adalah banyak perbandingan pada lintasan terpanjang, yaitu kasus terburuknya. Dan $\lceil \lg(n+1) \rceil$ sama dengan $\lfloor \lg n \rfloor + 1$ untuk setiap $n \geq 1$ (keduanya adalah banyak digit biner n), sehingga Persamaan (2.11) menyatakan bahwa kasus terburuk pencarian biner, Persamaan (2.5), adalah yang terbaik yang mungkin. Pencarian biner tidak sekadar bagus; di antara algoritma yang hanya membandingkan, ia optimal.

Argumen ini punya celah, dan celah itu mendidik. Ia berlaku untuk algoritma yang belajar *hanya dengan membandingkan*. Pencarian interpolasi melakukan aritmetika pada nilai kunci, dan itulah sebabnya ia dapat mengalahkan lg$n$ pada data yang bersahabat. Tabel *hash* (Bab 15) menghitung posisi langsung dari kunci dan, secara rata-rata, menemukannya dalam $O(1)$ pemeriksaan. Untuk mengalahkan pencarian biner, Anda harus mengetahui sesuatu tentang kunci Anda di luar urutannya.

2.13 Profitina dalam Praktik

2.13.1 Di mana isi bab ini muncul dalam produk nyata

Profitina adalah platform visibilitas AI: ia memindai situs web sebuah merek, menilai seberapa siap situs itu dikutip oleh mesin penjawab AI, menjelaskan setiap sinyalnya, dan membantu pemiliknya memperbaiki apa yang ditemukannya. Platform ini berjalan di lingkungan produksi di `app.profitina.com`, sebagai aplikasi web Python di atas PostgreSQL dengan model Qwen2.5 yang dijalankan secara lokal untuk mesin konten dan bantuan teknisnya [15]. Pencarian, dalam bentuk-bentuk yang dibahas di bab ini, mendasari beberapa fiturnya.

Deret waktu itu terurut, sehingga rentang cukup dijawab dengan dua pencarian batas. Dasbor analitik historis memperlihatkan skor visibilitas AI setiap merek pekan demi pekan, dan mengekspornya sebagai CSV untuk periode yang dipilih. Riwayat sebuah merek secara alami tersimpan menurut urutan waktu, karena hasil pindaian ditambahkan ketika terjadi. "Pindaian antara 1 Juli dan 30 September" kemudian tepat berupa $A[lower(\text{1 Juli})..lower(\text{1 Oktober}) - 1]$ — dua pencarian *lower bound*, $O(\lg n)$, betapapun panjangnya riwayat itu. Di basis data, gagasan

yang sama muncul sebagai indeks pada kolom stempel waktu, yang merupakan struktur terurut yang dicari dengan cara yang sama.

"N baru" adalah *upper bound*. Dasbor memperlihatkan, untuk setiap kanal komunitas yang diikuti, berapa banyak unggahan tingkat atas yang baru sejak pengguna terakhir melihat. Unggahan disimpan menurut urutan waktu, sehingga hitungannya adalah banyak unggahan yang lebih baru daripada waktu baca terakhir t: $n - upper(t)$. Tidak perlu menelusuri seluruh kanal.

Peringkat adalah titik sisip. Papan peringkat adalah daftar skor yang terurut. Peringkat seorang pengguna adalah banyak skor di atas skornya, yaitu satu pencarian batas pada daftar terurut, bukan satu kali penelusuran atas semua pengguna.

Penyaringan memerlukan uji keanggotaan yang cepat. Setiap tautan yang diunggah di komunitas disaring oleh lapisan aturan dasar yang menggabungkan beberapa sinyal sebelum pendapat AI lokal yang bersifat opsional dimintai. Penyaring semacam ini harus menjawab pertanyaan berbentuk "apakah kita pernah melihat ini?" berkali-kali setiap detik, dan perbedaan antara menelusuri sebuah daftar dan mencari di daftar yang terurut — n pemeriksaan lawan $\lfloor \lg n \rfloor + 1$ — adalah perbedaan antara penyaring yang mampu mengikuti laju komunitas dan yang tidak. (Bab 15 memperlihatkan tabel *hash*, yang menjawab pertanyaan keanggotaan dalam rata-rata $O(1)$ pemeriksaan.)

Pola yang perlu dibawa pulang bersifat umum: **simpanlah data dalam urutan yang cocok dengan pertanyaan yang akan Anda ajukan kepadanya, dan setiap pertanyaan menjadi pencarian batas.** Rincian implementasi Profitina yang sebenarnya adalah miliknya sendiri; *bentuk* persoalan-persoalan ini — riwayat terurut, hitungan sejak waktu tertentu, peringkat, keanggotaan — berlaku di mana saja.

2.13.2 Studi kasus yang dikerjakan: Profitina Laundry

Studi kasus kita yang dapat dijalankan adalah **Profitina Laundry**, layanan binatu kecil yang mencatat pesanan setiap hari: nomor pesanan, jam pesanan diterima, nama depan pelanggan, dan berat cucian dalam kilogram. Pesanan datang menurut urutan waktu dan menerima nomor berurutan, sehingga daftar hari itu terurut menurut waktu *dan* menurut nomor tanpa biaya apa pun — datanya terurut karena cara ia dihasilkan. **Kode 2.31** menjawab empat pertanyaan sehari-hari dengannya: pesanan yang diterima antara pukul 10:00 dan 12:00 beserta total beratnya (dua *lower bound*), pencarian pesanan menurut nomor (satu *lower bound* dan satu uji kesamaan), pesanan yang paling dekat dengan pukul 13:00 (satu *lower bound* dan satu perbandingan dengan tetangga kirinya), dan pesanan pertama dari pelanggan tertentu — menurut nama, yang *tidak* terurut, sehingga memerlukan penelusuran sekuensial. **Kode 2.32** dan **Kode 2.33** adalah versi Java dan Python-nya.

Kode 2.31 `laundry_orders.cpp`: pesanan Profitina Laundry dicari menurut rentang waktu, nomor, waktu terdekat, dan nama.

```cpp
// Profitina Laundry: pesanan satu hari, disimpan menurut urutan kedatangan.
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

struct Order {
    int number;
    string time;  // "HH:MM"
    string customer;
    double kg;
};

const vector<Order> ORDERS = {
        {1001, "07:45", "Sari", 3.5},  {1002, "08:10", "Budi", 5.0},
        {1003, "08:55", "Dewi", 2.0},  {1004, "09:30", "Andi", 7.5},
        {1005, "10:05", "Rina", 4.0},  {1006, "10:40", "Joko", 6.0},
        {1007, "11:15", "Maya", 3.0},  {1008, "11:50", "Tono", 8.0},
        {1009, "12:20", "Lina", 2.5},  {1010, "13:05", "Agus", 5.5},
        {1011, "14:30", "Fitri", 4.5}, {1012, "15:10", "Hadi", 3.0}};

int minutes(const string& hhmm) {
    return stoi(hhmm.substr(0, 2)) * 60 + stoi(hhmm.substr(3));
}

// Indeks pertama yang nilainya >= key; a terurut menaik.
int lowerBound(const vector<int>& a, int key) {
    int low = 0, high = (int)a.size();
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (a[mid] < key)
            low = mid + 1;
        else
            high = mid;
    }
    return low;
}
```

Kode 2.31 (Lanjutan) *laundry_orders.cpp: pesanan Profitina Laundry dicari menurut rentang waktu, nomor, waktu terdekat, dan nama.*

```
int main() {
    vector<int> times, numbers;
    for (const Order& o : ORDERS) {
        // menaik: pesanan datang menurut urutan waktu
        times.push_back(minutes(o.time));
        // juga menaik: nomor diberikan berurutan
        numbers.push_back(o.number);
    }

    int first = lowerBound(times, minutes("10:00")),
        stop = lowerBound(times, minutes("12:00"));
    double total = 0;
    for (int i = first; i < stop; i++) total += ORDERS[i].kg;
    printf("orders from 10:00 to before 12:00: %d orders, %.1f kg\n", stop - first,
           total);
    for (int i = first; i < stop; i++)
        printf("  #%d %s %-5s %.1f kg\n", ORDERS[i].number, ORDERS[i].time.c_str(),
               ORDERS[i].customer.c_str(), ORDERS[i].kg);

    for (int number : {1009, 1020}) {
        int i = lowerBound(numbers, number);
        if (i < (int)numbers.size() && numbers[i] == number)
            printf("order #%d: %s at %s, %.1f kg\n", number,
                   ORDERS[i].customer.c_str(), ORDERS[i].time.c_str(),
                   ORDERS[i].kg);
        else
            printf("order #%d: no such order today\n", number);
    }

    int t = minutes("13:00");
    int i = lowerBound(times, t);  // pesanan pertama pada atau setelah 13:00
    if (i > 0 && (i == (int)times.size() || t - times[i - 1] <= times[i] - t))
        i--;  // pesanan tepat sebelumnya lebih dekat
    printf("order closest to 13:00: #%d at %s (%s)\n", ORDERS[i].number,
           ORDERS[i].time.c_str(), ORDERS[i].customer.c_str());

    string name = "Maya";  // nama tidak terurut: telusuri satu per satu
    for (const Order& o : ORDERS)
        if (o.customer == name) {
            printf("first order by %s: #%d at %s\n", name.c_str(), o.number,
                   o.time.c_str());
            break;
        }
    return 0;
}
```

Kode 2.32 `LaundryOrders.java`: program yang sama dalam Java.

```java
// Profitina Laundry: pesanan satu hari, disimpan menurut urutan kedatangan.
import java.util.List;
import java.util.Locale;

public class LaundryOrders {
    record Order(int number, String time, String customer, double kg) {}

    static final List<Order> ORDERS = List.of(new Order(1001, "07:45", "Sari", 3.5),
            new Order(1002, "08:10", "Budi", 5.0),
            new Order(1003, "08:55", "Dewi", 2.0),
            new Order(1004, "09:30", "Andi", 7.5),
            new Order(1005, "10:05", "Rina", 4.0),
            new Order(1006, "10:40", "Joko", 6.0),
            new Order(1007, "11:15", "Maya", 3.0),
            new Order(1008, "11:50", "Tono", 8.0),
            new Order(1009, "12:20", "Lina", 2.5),
            new Order(1010, "13:05", "Agus", 5.5),
            new Order(1011, "14:30", "Fitri", 4.5),
            new Order(1012, "15:10", "Hadi", 3.0));

    static int minutes(String hhmm) {
        return Integer.parseInt(hhmm.substring(0, 2)) * 60 +
                Integer.parseInt(hhmm.substring(3));
    }

    // Indeks pertama yang nilainya >= key; a terurut menaik.
    static int lowerBound(int[] a, int key) {
        int low = 0, high = a.length;
        while (low < high) {
            int mid = low + (high - low) / 2;
            if (a[mid] < key)
                low = mid + 1;
            else
                high = mid;
        }
        return low;
    }
```

Kode 2.32 (Lanjutan) *LaundryOrders.java: program yang sama dalam Java.*

```
    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);  // cetak 21.0, jangan 21,0
        int n = ORDERS.size();
        int[] times = new int[n], numbers = new int[n];
        for (int i = 0; i < n; i++) {
            // menaik: pesanan datang menurut urutan waktu
            times[i] = minutes(ORDERS.get(i).time());
            // juga menaik: nomor diberikan berurutan
            numbers[i] = ORDERS.get(i).number();
        }

        int first = lowerBound(times, minutes("10:00")),
            stop = lowerBound(times, minutes("12:00"));
        double total = 0;
        for (int i = first; i < stop; i++) total += ORDERS.get(i).kg();
        System.out.printf("orders from 10:00 to before 12:00: %d orders, %.1f kg%n",
                stop - first, total);
        for (int i = first; i < stop; i++) {
            Order o = ORDERS.get(i);
            System.out.printf("  #%d %s %-5s %.1f kg%n", o.number(), o.time(),
                    o.customer(), o.kg());
        }

        for (int number : new int[] {1009, 1020}) {
            int i = lowerBound(numbers, number);
            if (i < n && numbers[i] == number) {
                Order o = ORDERS.get(i);
                System.out.printf("order #%d: %s at %s, %.1f kg%n", number,
                        o.customer(), o.time(), o.kg());
            } else {
                System.out.printf("order #%d: no such order today%n", number);
            }
        }

        int t = minutes("13:00");
        int i = lowerBound(times, t);  // pesanan pertama pada atau setelah 13:00
        if (i > 0 && (i == n || t - times[i - 1] <= times[i] - t))
            i--;  // pesanan tepat sebelumnya lebih dekat
        Order c = ORDERS.get(i);
        System.out.printf("order closest to 13:00: #%d at %s (%s)%n", c.number(),
                c.time(), c.customer());
```

Kode 2.32 (Lanjutan) *LaundryOrders.java: program yang sama dalam Java.*

```
        // nama tidak terurut: telusuri satu per satu
        String name = "Maya";
        for (Order o : ORDERS)
            if (o.customer().equals(name)) {
                System.out.printf("first order by %s: #%d at %s%n", name,
                        o.number(), o.time());
                break;
            }
    }
}
```

Kode 2.33 `laundry_orders.py`: program yang sama dalam Python.

```
"""Profitina Laundry: pesanan satu hari, disimpan menurut urutan kedatangan."""

ORDERS = [
    (1001, "07:45", "Sari", 3.5),
    (1002, "08:10", "Budi", 5.0),
    (1003, "08:55", "Dewi", 2.0),
    (1004, "09:30", "Andi", 7.5),
    (1005, "10:05", "Rina", 4.0),
    (1006, "10:40", "Joko", 6.0),
    (1007, "11:15", "Maya", 3.0),
    (1008, "11:50", "Tono", 8.0),
    (1009, "12:20", "Lina", 2.5),
    (1010, "13:05", "Agus", 5.5),
    (1011, "14:30", "Fitri", 4.5),
    (1012, "15:10", "Hadi", 3.0),
]


def minutes(hhmm):
    return int(hhmm[:2]) * 60 + int(hhmm[3:])


def lower_bound(a, key):
    """Indeks pertama yang nilainya >= key; a terurut menaik."""
    low, high = 0, len(a)
    while low < high:
        mid = low + (high - low) // 2
        if a[mid] < key:
            low = mid + 1
        else:
            high = mid
    return low


def main():
    # menaik: pesanan datang menurut urutan waktu
    times = [minutes(o[1]) for o in ORDERS]
    # juga menaik: nomor diberikan berurutan
    numbers = [o[0] for o in ORDERS]
```

Kode 2.33 (Lanjutan) *laundry_orders.py: program yang sama dalam Python.*

```
    first, stop = lower_bound(times, minutes("10:00")), lower_bound(
        times, minutes("12:00")
    )
    total = sum(ORDERS[i][3] for i in range(first, stop))
    print(
        "orders from 10:00 to before 12:00: %d orders, %.1f kg"
        % (stop - first, total)
    )
    for i in range(first, stop):
        print("  #%d %s %-5s %.1f kg" % ORDERS[i])

    for number in (1009, 1020):
        i = lower_bound(numbers, number)
        if i < len(numbers) and numbers[i] == number:
            print(
                "order #%d: %s at %s, %.1f kg"
                % (number, ORDERS[i][2], ORDERS[i][1], ORDERS[i][3])
            )
        else:
            print("order #%d: no such order today" % number)

    t = minutes("13:00")
    i = lower_bound(times, t)  # pesanan pertama pada atau setelah 13:00
    if i > 0 and (i == len(times) or t - times[i - 1] <= times[i] - t):
        i -= 1  # pesanan tepat sebelumnya lebih dekat
    print(
        "order closest to 13:00: #%d at %s (%s)"
        % (ORDERS[i][0], ORDERS[i][1], ORDERS[i][2])
    )

    name = "Maya"  # nama tidak terurut: telusuri satu per satu
    for o in ORDERS:
        if o[2] == name:
            print("first order by %s: #%d at %s" % (name, o[0], o[1]))
            break


if __name__ == "__main__":
    main()
```

Ketiganya mencetak:

```
orders from 10:00 to before 12:00: 4 orders, 21.0 kg
  #1005 10:05 Rina  4.0 kg
  #1006 10:40 Joko  6.0 kg
  #1007 11:15 Maya  3.0 kg
  #1008 11:50 Tono  8.0 kg
order #1009: Lina at 12:20, 2.5 kg
order #1020: no such order today
order closest to 13:00: #1010 at 13:05 (Agus)
first order by Maya: #1007 at 11:15
```

Dua pilihan rancangan dalam program ini layak ditiru. Yang pertama: jam diubah menjadi menit sejak tengah malam, yaitu sebuah bilangan, sebelum pencarian, karena membandingkan `"9:30"` dengan `"10:05"` sebagai teks memberikan urutan yang salah (karakter `9` datang sesudah `1`); dengan nol di depan, `"09:30"`, perbandingan teks kebetulan benar, dan program ini mempertahankan nol di depan itu, tetapi mengubahnya menjadi bilangan menghilangkan keraguan. Yang kedua: pelanggan di sini dikenali hanya dari nama depan, dan pencarian menurut nama adalah penelusuran biasa atas pesanan satu hari. Sistem sungguhan akan mencari menurut pengenal pelanggan, bukan menurut medan pribadi, dan sesuai undang-undang pelindungan data

pribadi Indonesia, UU PDP 27/2022, akan menjauhkan medan pribadi dari kunci pengurutan, log, dan *cache*, sehingga data seorang pelanggan dapat ditemukan dan dihapus atas permintaannya.

2.14 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. **Pencarian biner pada data tidak terurut.** Ia mengembalikan jawaban yang salah tanpa suara (Subbab 2.5.6). Jika Anda tidak mengurutkan datanya sendiri, periksalah, atau carilah secara sekuensial.
2. **Mencampur konvensi interval.** Inisialisasi interval tertutup dengan syarat gelung setengah terbuka, atau sebaliknya. Pilih satu kolom **Tabel 2.2** dan ambil setiap barisnya dari kolom itu.
3. **Memperbarui batas menjadi `mid` pada interval tertutup.** `low = mid` atau `high = mid` dapat membiarkan interval tidak berubah dan berputar selamanya. Pada interval tertutup, kedua pembaruan harus melewati `mid`.
4. **Menghitung `(low + high) / 2`.** Meluap untuk indeks besar di C, C++, dan Java. Tulislah `low + (high - low) / 2` di mana-mana.
5. **Mengharapkan salinan pertama dari pencarian biner biasa.** `Arrays.binarySearch` dan **Pseudokode 2.3** mengembalikan *salah satu* salinan. Pakailah *lower bound* untuk salinan pertama, dan *upper bound* dikurangi satu untuk salinan terakhir.
6. **Salah membaca nilai kembalian negatif Java.** Hasil negatif tidak berarti "−1 artinya tidak ada"; ia menyandikan titik sisip sebagai $-(ip) - 1$. Ujilah `r >= 0` untuk keberadaan dan hitunglah `-r - 1` untuk posisinya.
7. **Mengembalikan penanda, bukan posisi.** Pencarian yang hanya mencetak FOUND memaksa pemanggil mencari ulang. Kembalikan indeksnya, dan -1 bila tidak ada.
8. **Berhenti lebih awal ketika menghitung.** Untuk menghitung salinan atau mendaftar semuanya, gelung sekuensial harus berjalan sampai ujung — atau, pada data terurut, pakailah *upper bound* dikurangi *lower bound*.
9. **Lupa mengembalikan sel sentinel, atau memakai sentinel pada data bersama atau data hanya-baca.** Simpan sel cadangan, kembalikan, dan jangan pernah menanam sentinel di larik yang mungkin sedang dibaca *thread* lain.
10. **Pencarian interpolasi tanpa pengamannya.** Tanpa pemeriksaan rentang $A[low] \le x \le A[high]$ taksirannya dapat menunjuk ke luar larik; tanpa pemeriksaan $A[low] = A[high]$ ia membagi dengan nol; dan pada kunci yang besar hasil kali $(x - A[low])(high - low)$ dapat meluapkan `int`. Program C asli yang mengilhami Subbab 2.7 memakai gelung `do … while` yang memeriksa elemen sebelum memeriksa rentang, dan memerlukan perbaikan-perbaikan ini.
11. **Memercayai pencarian interpolasi pada data yang belum diukur.** Pada katalog nyata ia 17 kali lebih buruk daripada pencarian biner. Ukurlah sebaran kunci Anda sebelum mengandalkannya.
12. **Menambahkan banyak elemen ke sebuah alamat.** Alamat $A[7]$ adalah $base + 7w$, bukan $base + 7$. Kalikan dengan lebar elemen.
13. **Menelusuri larik dua dimensi kolom demi kolom di C++, Java, atau Python.** Jawabannya sama, tetapi urutan memori pada **Gambar 2.8** membuat gelung baris demi baris lebih cepat. Letakkan gelung kolom di bagian dalam.
14. **Membandingkan jam atau bilangan sebagai teks.** `"9:30" > "10:05"` sebagai string. Ubahlah menjadi bilangan lebih dahulu, atau beri nol di depan secara konsisten.
15. **Mengira hitungan pemeriksaan sama dengan waktu.** Hitungan itu meramalkan pertumbuhan, bukan kecepatan. Ukurlah, seperti yang dilakukan Subbab 2.10.

Bab 4 mengerjakan soal-soal persis seperti ini, beberapa di antaranya dibangun di sekitar pencarian di katalog Pustaka Ceria.

2.15 Ringkasan Bab

- **Masalah pencarian** meminta *posisi* sebuah kunci dalam koleksi, atau pernyataan "tidak ada" yang jelas (-1). Kita mengukur biaya dalam **pemeriksaan**, yaitu kunjungan ke satu elemen.
- **Pencarian sekuensial** bekerja pada data apa pun dan memerlukan rata-rata $(n+1)/2$ pemeriksaan untuk kunci yang ada dan n untuk kunci yang tidak ada: $O(n)$. Ia satu-satunya pilihan untuk data tidak terurut dan data yang dibaca dari depan ke belakang. Untuk menghitung salinan, ia tidak boleh berhenti lebih awal.
- **Sentinel** — salinan kunci di sel cadangan sesudah data — menyingkirkan uji batas dari gelung, memangkas separuh uji tetapi tidak mengurangi pemeriksaan. Pada kompiler dan prosesor modern, keuntungan terukurnya nihil di C++ dan Java dan sekitar 9% di Python.
- **Pencarian biner** pada data terurut membagi dua interval pada setiap pemeriksaan. Invariannya — semua yang di kiri *low* lebih kecil, semua yang di kanan *high* lebih besar — membuktikan kebenarannya, dan $T(n) = \lfloor \lg n \rfloor + 1$ membatasi banyak pemeriksaannya: 14 untuk sepuluh ribu elemen, 30 untuk semiliar. Ia punya bentuk iteratif dan rekursif.
- Pencarian biner mudah keliru: pakailah satu konvensi interval secara konsisten, pastikan setiap pembaruan mengecilkan interval, hitung titik tengah sebagai $low + \lfloor (high - low)/2 \rfloor$, dan jangan menganggap Anda tahu salinan mana dari kunci berduplikat yang didapat.
- ***Lower bound*** dan ***upper bound*** menemukan elemen pertama $\geq x$ dan elemen pertama $>x$. Berdua mereka menjawab keberadaan, salinan pertama dan terakhir, banyak salinan, titik sisip, dan hitungan dalam rentang.
- **Pencarian interpolasi** menaksir posisi dari nilai kunci. Pada kunci yang tersebar merata ia rata-rata memerlukan sekitar $\lg\lg n$ pemeriksaan; pada kunci yang menumpuk ia merosot ke arah $O(n)$ — pada katalog nyata ia memerlukan 211 pemeriksaan, sedangkan pencarian biner 12.
- **Pencarian lompat** melompat $\sqrt{n}$ tempat sekaligus lalu menelusuri satu blok: paling banyak sekitar $2\sqrt{n}$ pemeriksaan, untuk media penyimpanan yang gerakan mundurnya mahal.
- Larik dua dimensi disimpan dalam urutan **baris-mayor** di C dan C++: $address(B[i][j]) = base + (iC + j)w$. Kisi tidak terurut ditelusuri; kisi yang baris dan kolomnya terurut dicari dengan **metode tangga** $O(R + C)$; kisi yang terurut dalam urutan baris-mayor dicari secara biner sebagai satu deret.
- **Ukurlah**: hitungan pemeriksaan meramalkan pertumbuhan, tetapi *cache*, peramalan percabangan, dan mahalnya pembagian menentukan kecepatan. Pencarian biner di Python tetap mengalahkan pencarian sekuensial di C++.
- Pustaka standar menyediakan semua ini: `std::find`, `std::binary_search`, `std::lower_bound`, `std::upper_bound`, `std::equal_range` di C++; `indexOf`, `Arrays.binarySearch`, dan `Collections.binarySearch` (hasil negatif $= -(ip) - 1$) di Java; `in`, `list.index`, `bisect_left`, `bisect_right`, dan `insort` di Python.
- Tidak ada algoritma yang hanya membandingkan dapat mengalahkan $\lceil \lg(n+1) \rceil$ pemeriksaan pada kasus terburuk, sehingga pencarian biner optimal di antara pencarian berbasis perbandingan; mengalahkannya memerlukan aritmetika pada kunci (interpolasi) atau *hashing* (Bab 15).

2.15.1 Menatap ke depan

Pencarian biner, pencarian interpolasi, dan pencarian lompat semuanya berasumsi bahwa seseorang sudah mengurutkan datanya. Bab 3 membahas "seseorang" itu: ia membangun algoritma pengurutan klasik, dari *bubble sort* sampai *quicksort* dan *merge sort*, mengukurnya pada katalog yang sama, dan memperlihatkan bahwa mengurutkan sekali agar dapat mencari secara biner berkali-kali adalah salah satu tawaran terbaik dalam komputasi.

2.16 Soal Latihan

2.1 Telusurilah versi penanda dan versi berhenti-awal pencarian sekuensial pada $[8,10,6,-2,11,7,1,100]$ untuk kunci 11. Berapa perbandingan yang dilakukan masing-masing? Kemudian telusurilah pencarian kemunculan *terakhir* 041 dalam log peminjaman [041,088,025,041,063,041,096,088] dengan menelusuri dari belakang. Berapa perbandingan yang dilakukannya, dan mengapa menelusuri dari belakang adalah gagasan yang tepat di sini?

2.2 Sebuah katalog memuat $n = 2{,}000$ rekaman tanpa urutan tertentu, dan 70% pencarian adalah untuk kunci yang ada. Pakailah Persamaan (2.3) untuk menghitung rata-rata banyak pemeriksaan pencarian sekuensial. Seberapa besar p harus agar rata-ratanya turun di bawah 1,200?

2.3 Pada **Pseudokode 2.2**, apa yang salah jika baris 6 (mengembalikan sel cadangan) dihilangkan dan larik yang sama kemudian dicari dengan **Pseudokode 2.1** biasa untuk $n + 1$ elemen? Berikan contoh konkret dengan larik **Gambar 2.2**. Sebutkan dua situasi ketika teknik sentinel sama sekali tidak boleh dipakai.

2.4 Telusurilah **Pseudokode 2.3** pada $A = [3,9,11,12,15,17,23,31,35]$ untuk kunci 31 dan 4. Untuk setiap pemeriksaan, berikan low, $high$, mid, dan $A[mid]$. Di mana low berakhir untuk kunci 4, dan apa arti indeks itu?

2.5 Dengan Persamaan (2.5), carilah banyak pemeriksaan kasus terburuk pencarian biner untuk $n = 1, 2, 7, 8, 1{,}000, 1{,}000{,}000$, dan 10^9. Untuk nilai n yang mana kasus terburuk bertambah satu?

2.6 Fungsi C++ berikut dimaksudkan sebagai pencarian biner pada interval tertutup. Temukan setiap bug-nya, berikan masukan yang memperlihatkan masing-masing, dan perbaiki fungsinya.

```
int search(const std::vector<int>& a, int x) {
    int low = 0, high = a.size();
    while (low < high) {
        int mid = (low + high) / 2;
        if (a[mid] == x) return mid;
        if (a[mid] < x) low = mid;
        else high = mid - 1;
    }
    return -1;
}
```

2.7 Dengan hanya memakai fungsi *lower bound*, tuliskan ungkapan untuk (a) indeks salinan terakhir x, (b) banyak elemen dalam $[a, b]$ (kedua ujung termasuk) untuk kunci bilangan bulat, dan (c) banyak elemen yang lebih besar secara tegas daripada x. Periksalah masing-masing pada tahun-tahun di **Gambar 2.5** dengan $x = 2012$, $a = 2001$, $b = 2009$.

2.8 Telusurilah **Pseudokode 2.6** pada rak **Tabel 2.1** untuk kunci 041 dan 090. Berikan setiap taksiran sampai dua angka desimal.

2.9 Tunjukkan bahwa Persamaan (2.7) selalu memberikan $low \le pos \le high$ bila $A[low] \le x \le A[high]$ dan $A[low] < A[high]$. Mengapa jaminan ini hilang jika pemeriksaan rentang di baris 3 **Pseudokode 2.6** dihapus?

2.10 Untuk $n = 10{,}000$, hitunglah Persamaan (2.8) untuk $m = 10$, 50, 100, 200, dan 1,000, lalu pastikan bahwa $m = 100$ adalah yang terbaik. Salah satu varian pencarian lompat mengganti penelusuran sekuensial di dalam blok dengan pencarian biner. Berapa kasus terburuknya, dan berapa ukuran blok terbaik untuknya?

2.11 Sebuah larik dideklarasikan `int A[4][6]` dan dimulai di alamat 0x2000; sebuah `int` menempati 4 byte. Hitunglah alamat $A[3][2]$ dalam urutan baris-mayor, dan alamat yang akan dimilikinya jika larik disimpan dalam urutan kolom-mayor.

2.12 Telusurilah **Pseudokode 2.8** pada kisi **Gambar 2.9** untuk kunci 48 dan 26. Daftarlah elemen-elemen yang diperiksa. Mengapa pencarian harus dimulai di pojok kanan atas (atau kiri bawah), dan bukan di kiri atas?

2.13 *Pencarian Fibonacci dan pencarian eksponensial.* (a) Pencarian Fibonacci [16] membagi interval pada bilangan Fibonacci, bukan paruhan, sehingga ia hanya memerlukan penjumlahan dan pengurangan untuk menghitung posisi pemeriksaannya. (b) Pencarian eksponensial [17] memeriksa posisi 1,2,4,8, … sampai melewati kunci, lalu melakukan pencarian biner pada rentang terakhir; ia cocok untuk larik yang panjangnya tidak diketahui atau tidak terbatas. Implementasikan keduanya dalam C++, Java, dan Python, lalu jalankan pada [3,9,11,12,15,17,23,31,35] untuk kunci 17, 35, 3, dan 10 sambil menghitung pemeriksaan. Tunjukkan bahwa pencarian eksponensial menemukan kunci di posisi i dengan $O(\lg i)$ pemeriksaan.

2.14 Profitina Laundry ingin tahu berapa pesanan yang datang *sebelum* pukul 09:00 dan berapa yang datang *pada atau setelah* pukul 14:00. Tuliskan masing-masing sebagai ungkapan dalam *lower bound* atau *upper bound* pada larik `times` di **Kode 2.31**, lalu hitung keduanya dengan tangan. Apa yang dikembalikan `Arrays.binarySearch(times, 600)` di Java, dan apa arti nilai itu?

2.15 Pada **Tabel 2.3**, mengapa rata-rata sentinel pada kunci yang tidak ada adalah 10.001, bukan 10.000? Mengapa sentinel di Python menghemat waktu sedangkan di C++ tidak? Dengan memakai Subbab 2.10.1, jelaskan dengan kata-kata Anda sendiri mengapa pencarian interpolasi memerlukan 211 pemeriksaan pada pengenal nyata.

2.16 Mesin pencari web menjawab sebuah kueri atas miliaran halaman dalam sepersekian detik. Jelaskan mengapa ia tidak dapat menjalankan pencarian sekuensial atas halaman-halaman itu untuk setiap kueri, dan uraikan bagaimana sebuah **indeks terbalik** (*inverted index*) — untuk setiap kata, daftar terurut halaman-halaman yang memuatnya — mengubah kueri dua kata menjadi operasi-operasi dari bab ini.

2.17 Penyelesaian Soal Latihan

2.1 Versi penanda selalu melakukan 8 perbandingan dan berakhir dengan penanda bernilai 1. Versi berhenti-awal membandingkan 8, 10, 6, −2, 11 lalu berhenti: 5 perbandingan, indeks 4. Untuk kemunculan terakhir 041, telusuri dari indeks 7 ke bawah: 088 (indeks 7), 096 (6), 041 (5) — ditemukan setelah 3 perbandingan, indeks 5. Menelusuri dari belakang itu tepat karena kecocokan *pertama* yang dijumpai ketika bergerak mundur adalah kemunculan *terakhir*, sehingga gelung boleh berhenti lebih awal; penelusuran maju harus berjalan sampai ujung sambil mengingat kecocokan terbaru.

2.2 Menurut Persamaan (2.3) dengan $n = 2{,}000$ dan $p = 0.7$: $C = 0.7 \cdot 2{,}001/2 + 0.3 \cdot 2{,}000 = 700.35 + 600 = 1{,}300.35$ pemeriksaan. Agar $C < 1{,}200$, kita perlu $p \cdot 1{,}000.5 + (1-p) \cdot 2{,}000 < 1{,}200$, yaitu $2{,}000 - 999.5p < 1{,}200$, yaitu $p > 800/999.5 \approx 0.8004$. Lebih dari sekitar 80% pencarian harus berhasil.

2.3 Jika sel cadangan tidak dikembalikan, sel itu masih memuat kunci yang terakhir dicari. Setelah pencarian sentinel untuk 5 pada [3,12,9, −4,21,6], sel $A[6]$ memuat 5; pencarian biasa berikutnya atas $n + 1 = 7$ sel untuk 5 lalu "menemukan" 5 di indeks 6, yang bukan data sesungguhnya. Kerusakannya tidak terlihat sampai seseorang mencari tepat kunci itu. Teknik ini tidak boleh dipakai pada larik hanya-baca (konstan), yang tidak dapat menerima sentinel, dan tidak pada larik yang mungkin sedang dibaca atau dicari *thread* lain pada saat yang sama, karena untuk sesaat larik memuat nilai yang bukan bagian dari data. Teknik ini juga memerlukan sel cadangan, sehingga tidak dapat diterapkan pada larik yang tepat penuh.

2.4 Kunci 31: $(low, high, mid, A[mid]) = (0,8,4,15)$, $31 > 15$, jadi $low = 5$; lalu $(5,8,6,23)$, $31 > 23$, jadi $low = 7$; lalu $(7,8,7,31)$: ditemukan di indeks 7 setelah 3 pemeriksaan. Kunci 4: $(0,8,4,15)$, $4 < 15$, $high = 3$; $(0,3,1,9)$, $4 < 9$, $high = 0$; $(0,0,0,3)$, $4 > 3$, $low = 1$; kini $low = 1 > high = 0$, tidak ada, setelah 3 pemeriksaan. $low = 1$ adalah titik sisipnya: 4 seharusnya berada di antara 3 dan 9.

2.5 $\lfloor \lg n \rfloor + 1$: $n = 1$ memberi 1; $n = 2$ memberi 2; $n = 7$ memberi 3; $n = 8$ memberi 4; $n = 1{,}000$ memberi $9 + 1 = 10$; $n = 10^6$ memberi $19 + 1 = 20$; $n = 10^9$ memberi $29 + 1 = 30$. Kasus terburuk bertambah satu tepat ketika n mencapai perpangkatan dua: pada $n = 2,4,8,16, \ldots$

2.6 Ada empat bug. (1) Inisialisasi `high = a.size()` dan syarat gelung `low < high` bersifat setengah terbuka, tetapi pembaruan `high = mid - 1` bersifat tertutup, sehingga sebuah elemen dapat terlewat. Pada [1,2] dengan kunci 1: $mid = 1$, $A[1] = 2 > 1$, $high = 0$, dan gelung berhenti dengan $low = high = 0$ tanpa pernah memeriksa $A[0]$; fungsi mengembalikan -1 padahal 1 ada di indeks 0. (2) Pembaruan `low = mid` dapat membiarkan interval tidak berubah. Pada [1,2,3] dengan kunci 4: $(low, high) = (0,3)$, $mid = 1$, $low = 1$; lalu $mid = 2$, $low = 2$; lalu $mid = 2$ lagi, $low = 2$ lagi, selamanya. (3) `(low + high) / 2` meluap untuk larik berisi lebih dari sekitar 2^{30} elemen. (4) `a.size()` tak bertanda; mengubahnya diam-diam menjadi `int` menyembunyikan konversi itu dan memunculkan peringatan pada `-Wextra`. Versi interval tertutup yang benar:

```
int search(const std::vector<int>& a, int x) {
    int low = 0, high = (int)a.size() - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (a[mid] == x) return mid;
        if (a[mid] < x) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

2.7 Tulis $L(v)$ untuk *lower bound* dari v. (a) Salinan terakhir x ada di $L(x+1) - 1$ untuk kunci bilangan bulat (secara umum, *upper bound* dikurangi satu), asalkan $L(x+1) > L(x)$. (b) Untuk kunci bilangan bulat, elemen dalam $[a, b]$ adalah yang $\geq a$ dan $< b + 1$: $L(b+1) - L(a)$. (c) Elemen yang lebih besar daripada x: $n - L(x+1)$. Pada tahun-tahun itu: (a) $L(2013) - 1 = 9 - 1 = 8$, dan memang $A[8] = 2012$ adalah 2012 terakhir; (b) $L(2010) - L(2001) = 7 - 1 = 6$ (2001, 2001, 2005, 2005, 2005, 2009); (c) $10 - L(2013) = 10 - 9 = 1$ (hanya 2020).

2.8 Kunci 041: $pos = 0 + \lfloor (16/71) \cdot 7 \rfloor = \lfloor 1.58 \rfloor = 1$, $A[1] = 034 < 041$, $low = 2$; lalu $pos = 2 + \lfloor (0/55) \cdot 5 \rfloor = 2 + \lfloor 0.00 \rfloor = 2$, $A[2] = 041$: ditemukan setelah 2 pemeriksaan. Kunci 090: $pos =$

$\lfloor (65/71) \cdot 7 \rfloor = \lfloor 6.41 \rfloor = 6$, $A[6] = 088 < 090$, $low = 7$; kini $090 < A[7] = 096$, sehingga pemeriksaan rentang gagal: tidak ada, setelah 1 pemeriksaan.

2.9 Tulis $f = (x - A[low])/(A[high] - A[low])$. Penyebutnya positif. Dari $A[low] \le x$ pembilangnya ≥ 0, jadi $f \ge 0$; dari $x \le A[high]$ pembilangnya paling besar sama dengan penyebut, jadi $f \le 1$. Karena itu $0 \le f(high - low) \le high - low$, dan pembulatan ke bawah mempertahankannya di $[0, high - low]$, sehingga $low \le pos \le high$. Tanpa pemeriksaan rentang, x bisa berada di bawah $A[low]$ (maka $f < 0$ dan $pos < low$, mungkin negatif) atau di atas $A[high]$ (maka $f > 1$ dan $pos > high$, mungkin melewati ujung): program membaca di luar larik.

2.10 $f(m) = 10{,}000/m + m - 1$: $f(10) = 1{,}000 + 9 = 1{,}009$; $f(50) = 200 + 49 = 249$; $f(100) = 100 + 99 = 199$; $f(200) = 50 + 199 = 249$; $f(1{,}000) = 10 + 999 = 1{,}009$. Minimumnya di $m = 100 = \sqrt{10{,}000}$, dan fungsinya simetris terhadap titik itu dalam arti bahwa m dan n/m memberikan biaya yang sama. Dengan pencarian biner di dalam blok, biayanya sekitar $n/m + \lg m$. Biaya ini menurun ketika m membesar, sampai $m = n$, ketika ia menjadi satu lompatan ditambah pencarian biner atas seluruh larik — yaitu pencarian biner biasa. Melompat baru sepadan ketika gerakan mundur itu mahal, dan dalam hal itu penelusuran di dalam blok memang sekuensial.

2.11 Baris-mayor: elemen bernomor $3 \cdot 6 + 2 = 20$, selisih $20 \cdot 4 = 80 = 0\text{x}50$, alamat $0\text{x}2050$. Kolom-mayor (4 baris): elemen bernomor $2 \cdot 4 + 3 = 11$, selisih $11 \cdot 4 = 44 = 0\text{x}2\text{C}$, alamat $0\text{x}202\text{C}$.

2.12 Kunci 48: periksa 50 (lebih besar, ke kiri), 40 (lebih kecil, ke bawah), 45 (lebih kecil, ke bawah), 48 — ditemukan di (2,3) setelah 4 pemeriksaan. Kunci 26: 50 (kiri), 40 (kiri), 30 (kiri), 20 (lebih kecil, bawah), 25 (lebih kecil, bawah), 29 (lebih besar, kiri), 27 (lebih besar, kiri); indeks kolom menjadi -1: tidak ada, setelah 7 pemeriksaan. Di pojok kiri atas kedua gerakan, ke kanan dan ke bawah, menuju elemen yang *lebih besar*, sehingga perbandingan dengan kunci yang lebih besar tidak memberi tahu arah mana yang harus dipilih. Di pojok kanan atas, satu gerakan menuju elemen yang lebih kecil dan yang lain menuju elemen yang lebih besar, sehingga setiap perbandingan menentukan arah dan menyingkirkan satu baris atau satu kolom; pojok kiri bawah punya sifat yang sama.

2.13 **Kode 2.34**, **Kode 2.35**, dan **Kode 2.36** mengimplementasikan kedua pencarian itu.

Kode 2.34 `fibonacci_exponential.cpp`: pencarian Fibonacci dan pencarian eksponensial.

```cpp
// Dua pencarian lagi pada data terurut: Fibonacci dan eksponensial.
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

// Membagi pada bilangan Fibonacci, bukan paruhan. Hasil: indeks atau -1.
int fibonacciSearch(const vector<int>& a, int key, int& probes) {
    int n = (int)a.size();
    int fib2 = 0, fib1 = 1;  // F(k-2), F(k-1)
    int fib = fib1 + fib2;   // F(k): perbesar sampai F(k) >= n
    while (fib < n) {
        fib2 = fib1;
        fib1 = fib;
        fib = fib1 + fib2;
    }
    int offset = -1;  // semua elemen sampai offset < key
    probes = 0;
    while (fib > 1) {
        int i = min(offset + fib2, n - 1);
        probes++;
        if (a[i] < key) {  // buang bagian depan sepanjang F(k-2)
            fib = fib1;
            fib1 = fib2;
            fib2 = fib - fib1;
            offset = i;
        } else if (a[i] > key) {  // simpan hanya bagian depan sepanjang F(k-2)
            fib = fib2;
            fib1 = fib1 - fib2;
            fib2 = fib - fib1;
        } else {
            return i;
        }
    }
    if (fib1 == 1 && offset + 1 < n) {
        probes++;
        if (a[offset + 1] == key) return offset + 1;
    }
    return -1;
}
```

Kode 2.34 (Lanjutan) *fibonacci_exponential.cpp: pencarian Fibonacci dan pencarian eksponensial.*

```
// Gandakan batas sampai melewati key, lalu cari biner pada rentang terakhir.
int exponentialSearch(const vector<int>& a, int key, int& probes) {
    int n = (int)a.size();
    probes = 1;
    if (a[0] == key) return 0;
    int bound = 1;
    while (bound < n) {
        probes++;
        if (a[bound] >= key) break;
        bound *= 2;
    }
    // a[bound / 2] < key sudah pasti
    int low = bound / 2 + 1, high = min(bound, n - 1);
    while (low <= high) {
        int mid = low + (high - low) / 2;
        probes++;
        if (a[mid] == key) return mid;
        if (a[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int main() {
    vector<int> data = {3, 9, 11, 12, 15, 17, 23, 31, 35};
    printf("data:");
    for (int x : data) printf(" %d", x);
    printf("\n");
    for (int key : {17, 35, 3, 10}) {
        int p, q;
        int i = fibonacciSearch(data, key, p);
        int j = exponentialSearch(data, key, q);
        printf("key %2d: Fibonacci -> %2d (probes: %d), "
               "exponential -> %2d (probes: %d)\n", key, i, p, j, q);
    }
    return 0;
}
```

Kode 2.35 `FibonacciExponential.java`: kedua pencarian yang sama dalam Java.

```java
// Dua pencarian lagi pada data terurut: Fibonacci dan eksponensial.
public class FibonacciExponential {
    // Membagi pada bilangan Fibonacci. Hasil: {indeks atau -1, pemeriksaan}.
    static int[] fibonacciSearch(int[] a, int key) {
        int n = a.length;
        int fib2 = 0, fib1 = 1;  // F(k-2), F(k-1)
        int fib = fib1 + fib2;  // F(k): perbesar sampai F(k) >= n
        while (fib < n) {
            fib2 = fib1;
            fib1 = fib;
            fib = fib1 + fib2;
        }
        int offset = -1;  // semua elemen sampai offset < key
        int probes = 0;
        while (fib > 1) {
            int i = Math.min(offset + fib2, n - 1);
            probes++;
            if (a[i] < key) {  // buang bagian depan sepanjang F(k-2)
                fib = fib1;
                fib1 = fib2;
                fib2 = fib - fib1;
                offset = i;
            } else if (a[i] > key) {  // simpan hanya bagian depan sepanjang F(k-2)
                fib = fib2;
                fib1 = fib1 - fib2;
                fib2 = fib - fib1;
            } else {
                return new int[] {i, probes};
            }
        }
        if (fib1 == 1 && offset + 1 < n) {
            probes++;
            if (a[offset + 1] == key) return new int[] {offset + 1, probes};
        }
        return new int[] {-1, probes};
    }
```

Kode 2.35 (Lanjutan) *FibonacciExponential.java: kedua pencarian yang sama dalam Java.*

```
    // Gandakan batas sampai melewati key, lalu cari biner pada rentang terakhir.
    static int[] exponentialSearch(int[] a, int key) {
        int n = a.length;
        int probes = 1;
        if (a[0] == key) return new int[] {0, probes};
        int bound = 1;
        while (bound < n) {
            probes++;
            if (a[bound] >= key) break;
            bound *= 2;
        }
        // a[bound / 2] < key sudah pasti
        int low = bound / 2 + 1, high = Math.min(bound, n - 1);
        while (low <= high) {
            int mid = low + (high - low) / 2;
            probes++;
            if (a[mid] == key) return new int[] {mid, probes};
            if (a[mid] < key)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return new int[] {-1, probes};
    }

    public static void main(String[] args) {
        int[] data = {3, 9, 11, 12, 15, 17, 23, 31, 35};
        StringBuilder line = new StringBuilder("data:");
        for (int x : data) line.append(" ").append(x);
        System.out.println(line);
        for (int key : new int[] {17, 35, 3, 10}) {
            int[] f = fibonacciSearch(data, key);
            int[] e = exponentialSearch(data, key);
            System.out.printf("key %2d: Fibonacci -> %2d (probes: %d), " +
                              "exponential -> %2d (probes: %d)%n",
                    key, f[0], f[1], e[0], e[1]);
        }
    }
}
```

Kode 2.36 `fibonacci_exponential.py`: kedua pencarian yang sama dalam Python.

```
"""Dua pencarian lagi pada data terurut: Fibonacci dan eksponensial."""


def fibonacci_search(a, key):
    """Membagi pada bilangan Fibonacci. Hasil: (indeks atau -1, pemeriksaan)."""
    n = len(a)
    fib2, fib1 = 0, 1  # F(k-2), F(k-1)
    fib = fib1 + fib2  # F(k): perbesar sampai F(k) >= n
    while fib < n:
        fib2, fib1 = fib1, fib
        fib = fib1 + fib2
    offset = -1  # semua elemen sampai offset < key
    probes = 0
    while fib > 1:
        i = min(offset + fib2, n - 1)
        probes += 1
        if a[i] < key:  # buang bagian depan sepanjang F(k-2)
            fib, fib1 = fib1, fib2
            fib2 = fib - fib1
            offset = i
        elif a[i] > key:  # simpan hanya bagian depan sepanjang F(k-2)
            fib, fib1 = fib2, fib1 - fib2
            fib2 = fib - fib1
        else:
            return i, probes
    if fib1 == 1 and offset + 1 < n:
        probes += 1
        if a[offset + 1] == key:
            return offset + 1, probes
    return -1, probes


def exponential_search(a, key):
    """Gandakan batas sampai melewati key, lalu cari biner pada rentang terakhir."""
    n = len(a)
    probes = 1
    if a[0] == key:
        return 0, probes
    bound = 1
    while bound < n:
        probes += 1
        if a[bound] >= key:
            break
        bound *= 2
    low, high = bound // 2 + 1, min(bound, n - 1)  # a[bound // 2] < key sudah pasti
    while low <= high:
```

Kode 2.36 (Lanjutan) *fibonacci_exponential.py: kedua pencarian yang sama dalam Python.*

```
        mid = low + (high - low) // 2
        probes += 1
        if a[mid] == key:
            return mid, probes
        if a[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1, probes


def main():
    data = [3, 9, 11, 12, 15, 17, 23, 31, 35]
    print("data:", " ".join(str(x) for x in data))
    for key in (17, 35, 3, 10):
        i, p = fibonacci_search(data, key)
        j, q = exponential_search(data, key)
        print(
            "key %2d: Fibonacci -> %2d (probes: %d), "
            "exponential -> %2d (probes: %d)" % (key, i, p, j, q)
        )


if __name__ == "__main__":
    main()
```

Ketiganya mencetak:

```
data: 3 9 11 12 15 17 23 31 35
key 17: Fibonacci ->  5 (probes: 3), exponential ->  5 (probes: 7)
key 35: Fibonacci ->  8 (probes: 3), exponential ->  8 (probes: 8)
key  3: Fibonacci ->  0 (probes: 3), exponential ->  0 (probes: 1)
key 10: Fibonacci -> -1 (probes: 4), exponential -> -1 (probes: 4)
```

Pencarian Fibonacci mencari bilangan Fibonacci terkecil $F_k \geq n$ dan memeriksa pada selisih F_{k-2}; setelah setiap perbandingan interval menyusut menjadi F_{k-1} atau F_{k-2} elemen, rasio sekitar $1/\varphi \approx 0.618$ per pemeriksaan alih-alih $1/2$, sehingga ia memerlukan sekitar $1.44 \lg n$ pemeriksaan pada kasus terburuk — sedikit lebih banyak daripada pencarian biner, tetapi dihitung hanya dengan penjumlahan. Pencarian eksponensial memeriksa $1, 2, 4, \ldots, 2^j$ dan berhenti pada 2^j pertama dengan $A[2^j] \geq x$. Jika kunci berada di posisi i, maka $2^{j-1} < i$, jadi $j \leq \lfloor \lg i \rfloor + 1$ pemeriksaan penggandaan, diikuti pencarian biner atas kurang dari $2^{j-1} \leq i$ elemen, yang memakan paling banyak $\lfloor \lg i \rfloor + 1$ pemeriksaan lagi. Totalnya $O(\lg i)$, tidak bergantung pada n — itulah sebabnya metode ini bekerja bahkan ketika n tidak diketahui, seperti pada pencarian tak terbatas Bentley dan Yao [17]. Keluaran memperlihatkan efeknya: kunci 3 di posisi 0 memerlukan satu pemeriksaan, sedangkan 35 di ujung jauh memerlukan delapan.

2.14 Dalam menit, 09:00 adalah 540 dan 14:00 adalah 840. Pesanan sebelum 09:00: $L(540) = 3$ (07:45, 08:10, 08:55). Pesanan pada atau setelah 14:00: $n - L(840) = 12 - 10 = 2$ (14:30 dan 15:10). `Arrays.binarySearch(times, 600)` mencari 10:00 = 600, yang bukan jam pesanan mana pun; titik sisipnya 4 (di antara 09:30 dan 10:05), sehingga panggilan itu mengembalikan $-(4) - 1 = -5$. Artinya "tidak ada pesanan tepat pukul 10:00; pesanan pertama sesudah 10:00 berada di indeks 4".

2.15 Untuk kunci yang tidak ada, gelung sentinel memeriksa seluruh 10.000 elemen sesungguhnya lalu sentinel itu sendiri, yang merupakan pemeriksaan ke-10.001. Di C++ dan Java, kompiler pengoptimal dan peramal percabangan prosesor membuat uji batas `i < n` nyaris gratis, sehingga menyingkirkannya tidak mengubah apa pun yang terukur; di Python setiap uji adalah

operasi tafsiran tersendiri yang benar-benar berbiaya, sehingga menyingkirkan separuhnya membantu. Pencarian interpolasi berasumsi bahwa pengenal naik merata dari yang terkecil ke yang terbesar. Dalam katalog nyata, separuh pengenal terletak di 1,2% pertama rentang itu, sehingga garis lurus melalui ujung-ujung interval sangat meremehkan posisi kunci yang umum; setiap pemeriksaan mendarat jauh di kiri, menggeser *low* hanya sedikit, dan taksiran berikutnya keliru dengan cara yang sama.

2.16 Pencarian sekuensial atas miliaran halaman akan membaca setiap halaman untuk setiap kueri — petabyte teks per detik untuk semua pengguna — dan itu mustahil. Indeks terbalik dibangun sekali, di muka: untuk setiap kata, daftar terurut pengenal halaman yang memuatnya. Kueri satu kata lalu menjadi pencarian kata itu dalam kosakata yang terurut (pencarian biner, atau *hashing*) diikuti pembacaan daftarnya. Kueri dua kata adalah irisan dua daftar terurut: telusuri kedua daftar bersama-sama seperti langkah penggabungan pada *merge sort* (Bab 3), atau, bila satu daftar jauh lebih pendek, ambil setiap pengenal dari daftar pendek dan carilah dengan pencarian biner atau eksponensial di daftar panjang, dengan biaya sekitar $s \lg \ell$ untuk daftar berpanjang $s \ll \ell$. Menentukan halaman mana yang ditampilkan lebih dahulu adalah persoalan lain, tetapi menemukannya adalah pencarian bab ini dalam skala raksasa.

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998, secs. 6.1–6.2.

[3] J. Bentley, *Programming Pearls*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2000, column 4.

[4] J. Bloch, "Extra, extra — read all about it: Nearly all binary searches and mergesorts are broken," *Google Research Blog*, Jun. 2, 2006. [Online]. Available: https://research.google/blog/extra-extra-read-all-about-it-nearly-all-binary-searches-and-mergesorts-are-broken/. [Accessed: 24-Sep-2026].

[5] W. W. Peterson, "Addressing for random-access storage," *IBM Journal of Research and Development*, vol. 1, no. 2, pp. 130–146, Apr. 1957, doi: 10.1147/rd.12.0130.

[6] Y. Perl, A. Itai, and H. Avni, "Interpolation search — a log log N search," *Communications of the ACM*, vol. 21, no. 7, pp. 550–553, Jul. 1978, doi: 10.1145/359545.359557.

[7] B. Shneiderman, "Jump searching: A fast sequential search technique," *Communications of the ACM*, vol. 21, no. 10, pp. 831–834, Oct. 1978, doi: 10.1145/359619.359623.

[8] Z. Zajac, "goodbooks-10k: A new dataset for book recommendations," 2017. [Online]. Available: https://github.com/zygmuntz/goodbooks-10k. License CC BY-SA 4.0. [Accessed: 24-Sep-2026].

[9] cppreference.com, "std::lower_bound," *C++ reference*. [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/lower_bound. [Accessed: 24-Sep-2026].

[10] Standard C++ Foundation, "GCC 16.1 released: C++26 reflection / contracts / safety hardening, C++20 by default, and more!," isocpp.org, Apr. 2026. [Online]. Available: https://isocpp.org/blog/2026/04/gcc-16.1. [Accessed: 24-Sep-2026].

[11] Oracle, "Class Arrays," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Arrays.html. [Accessed: 24-Sep-2026].

[12] Oracle, "Class Collections," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Collections.html. [Accessed: 24-Sep-2026].

[13] Oracle, "The arrival of Java 27," *Inside.java*, Sep. 15, 2026. [Online]. Available: https://inside.java/2026/09/15/jdk-27-available/. [Accessed: 24-Sep-2026].

[14] Python Software Foundation, “bisect — Array bisection algorithm,” *The Python Standard Library*, Python 3.14.7 documentation. [Online]. Available: https://docs.python.org/3/library/bisect.html. [Accessed: 24-Sep-2026].

[15] Profitina, “Profitina — AI Visibility (GEO) platform.” [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

[16] D. E. Ferguson, “Fibonaccian searching,” *Communications of the ACM*, vol. 3, no. 12, p. 648, Dec. 1960, doi: 10.1145/367487.367496.

[17] J. L. Bentley and A. C.-C. Yao, “An almost optimal algorithm for unbounded searching,” *Information Processing Letters*, vol. 5, no. 3, pp. 82–87, Aug. 1976, doi: 10.1016/0020-0190(76)90071-5.

Bab 3
Pengurutan

Setelah menuntaskan bab ini, Anda akan mampu menyatakan dengan tepat apa artinya data terurut dan mengapa begitu banyak bagian komputasi bergantung padanya; menukar dua nilai dengan aman, dalam C++ melalui pointer dan dalam Java serta Python dengan cara yang diizinkan kedua bahasa itu; melacak dengan tangan, di atas kertas, *bubble sort* (dengan keluar dininya), *exchange sort*, *selection sort*, *insertion sort*, *Shell sort*, *merge sort*, *quicksort* dengan partisi Lomuto maupun Hoare, *counting sort*, *radix sort*, dan versi pertama *heap sort*, semuanya pada enam bilangan yang sama; menulis masing-masing dari ingatan dalam C++, Java, dan Python; menghitung perbandingan dan pertukaran yang dilakukan setiap algoritma dan menurunkan hitungan itu sebagai rumus dalam n; menjelaskan arti *stabil*, *di tempat* (*in-place*), dan *adaptif*, serta algoritma mana yang memiliki sifat mana; membaca tabel kasus terbaik, rata-rata, dan terburuk serta mempertahankan setiap isinya; mengukur pengurutan sungguhan pada seribu hingga sejuta elemen dan membaca grafik hasilnya; dan memilih, untuk pekerjaan nyata, antara menulis pengurutan sendiri dan memanggil pengurutan yang sudah disediakan bahasa Anda, dengan mengetahui apa yang sebenarnya dikerjakan `std::sort`, `std::stable_sort`, `Arrays.sort` milik Java, dan `list.sort` milik Python pada tahun 2026.

3.1 Titik Berangkat

Bab 2 mengajukan satu pertanyaan kepada katalog Pustaka Ceria, “apakah buku ini ada di sini, dan di mana?”, lalu menjawabnya dengan beberapa cara. Dua di antaranya, pencarian sekuensial dan pencarian dengan *sentinel*, menyusuri rak dari satu ujung ke ujung lain dan sama sekali tidak memerlukan persiapan. Dua lainnya, pencarian biner dan pencarian interpolasi, jauh lebih cepat: pencarian biner menemukan satu buku di antara sejuta hanya dengan sekitar dua puluh kali melihat, bukan hingga sejuta kali. Namun keduanya membawa satu syarat yang mudah terlewat saat membaca. Keduanya hanya bekerja pada data yang **sudah terurut**. Pencarian biner memutuskan membuang seluruh separuh kanan rak karena kode buku di tengah lebih besar daripada kode yang dicari; pada rak yang tidak terurut, kesimpulan itu keliru, dan pencarian diam-diam mengembalikan jawaban yang salah.

Jadi Bab 2 meninggalkan utang. Program pencarian binernya mengurutkan larik dengan pustaka standar sebelum mencari, dan berjanji bahwa bab berikutnya akan menjelaskan bagaimana pengurutan sebenarnya dikerjakan. Bab ini melunasi utang itu, dan melunasinya penuh: sebelas algoritma, dari yang paling sederhana yang biasa ditemukan sendiri oleh siapa pun sampai yang bekerja di dalam pustaka yang Anda pakai setiap hari.

Mengapa sebelas, padahal satu pengurutan yang baik sudah cukup? Karena masing-masing mengajarkan gagasan yang akan muncul lagi di bagian lain buku ini, dan karena tidak ada satu pengurutan pun yang terbaik untuk semua pekerjaan. *Bubble sort* mengajarkan pertukaran bertetangga dan gagasan **keluar dini** (*early exit*). *Selection sort* mengajarkan bahwa banyaknya perbandingan dan banyaknya pemindahan adalah dua biaya yang berbeda. *Insertion sort* mengajarkan **adaptivitas**: bekerja lebih sedikit bila data sudah hampir terurut. *Shell sort* memperlihatkan bahwa perubahan kecil pada algoritma yang lambat dapat membuatnya cepat. *Merge sort* dan *quicksort* memperkenalkan *divide and conquer* (bagi dan kuasai), gagasan

rancangan di balik rekursi pada Bab 10. *Counting sort* dan *radix sort* memperlihatkan bahwa batas $n\log n$ yang terkenal itu hanya mengikat algoritma yang membandingkan, dan *heap sort* memberi pandangan pertama tentang *heap*, struktur yang pada Bab 14 dijadikan antrean prioritas.

Sepanjang bab ini setiap algoritma mengurutkan enam bilangan yang sama,

$$A = [22,10,15,3,8,2],$$

agar Anda dapat meletakkan jejak-jejaknya berdampingan dan melihat betapa berbedanya sebelas prosedur mencapai jawaban yang sama. Posisi kita beri nomor mulai dari 0, seperti C++, Java, dan Python, sehingga $A[0] = 22$ dan $A[5] = 2$.

3.2 Mengapa Mengurutkan, dan Istilah yang Kita Perlukan

3.2.1 Apa itu pengurutan

Mengurutkan sekumpulan butir berarti menata ulang butir-butir itu sehingga mengikuti suatu aturan urutan. Untuk bilangan, aturan yang lazim adalah urutan **menaik** (*ascending*), yang terkecil lebih dahulu, dan urutan **menurun** (*descending*), yang terbesar lebih dahulu:

```
data acak:    5  6  8  1  3 25 10
menaik:       1  3  5  6  8 10 25
menurun:     25 10  8  6  5  3  1
```

Tidak ada yang ditambahkan dan tidak ada yang dibuang; ketujuh bilangan yang sama muncul di setiap baris. Pengurutan hanya mengubah *di mana* setiap butir berada.

Aturan urutan tidak harus "bilangan yang lebih kecil lebih dahulu". Judul buku diurutkan secara alfabetis, tanggal secara kronologis, pesanan laundry menurut berat, hasil pencarian menurut relevansi. Semuanya bermuara pada satu pertanyaan yang diajukan pengurutan berulang kali tentang dua butir x dan y: *haruskah x datang sebelum y?* Bagian dari setiap butir yang diperiksa pertanyaan itu disebut **kunci** (*key*). Sebuah buku bisa diurutkan menurut kunci "judul" hari ini dan menurut kunci "kategori" esok hari; rekamannya sama, kuncinya berbeda.

> **DEFINISI.** Diberikan barisan $a_0, a_1, \ldots, a_{n-1}$ yang terdiri atas n butir dan suatu urutan pada kunci-kuncinya, **pengurutan** menghasilkan penataan ulang $a'_0, a'_1, \ldots, a'_{n-1}$ dari butir-butir yang sama sedemikian sehingga

$$a'_0 \le a'_1 \le \cdots \le a'_{n-1} \tag{3.1}$$

> dengan $\le$ membandingkan kunci. Urutan menurun adalah definisi yang sama dengan $\ge$ menggantikan $\le$.

Persamaan (3.1) menyatakan dua hal sekaligus, dan keduanya penting. Keluarannya harus *terurut*, dan keluarannya harus merupakan *penataan ulang*: sebuah "pengurutan" yang mengembalikan enam salinan bilangan terkecil memenuhi syarat pertama tetapi gagal pada syarat kedua.

3.2.2 Mengapa sepadan dengan usahanya

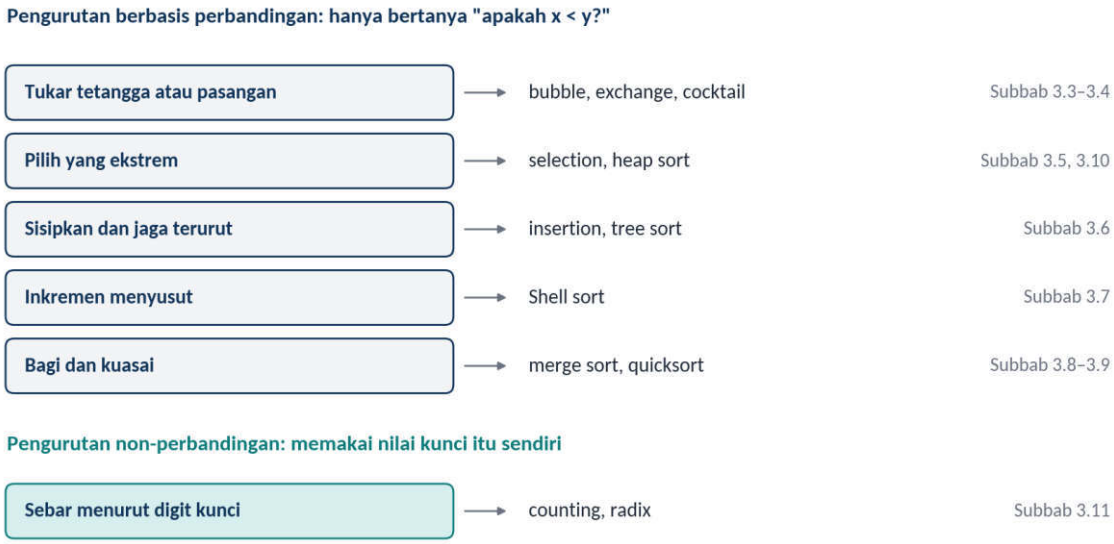
Pengurutan layak dipelajari karena data yang terurut membuat begitu banyak pekerjaan lain menjadi murah.

- **Pencarian.** Pencarian biner pada Bab 2 memerlukan sekitar $\log_2 n$ langkah pada data terurut dan sama sekali tidak dapat dijalankan pada data yang tidak terurut.
- **Menemukan duplikat.** Dalam daftar terurut, butir-butir yang sama berdiri bersebelahan, sehingga satu lintasan yang membandingkan tetangga menemukan semua duplikat. Tanpa pengurutan, Anda harus membandingkan setiap butir dengan setiap butir lainnya.
- **Pengelompokan dan laporan.** Laporan perpustakaan "banyak buku per kategori" menjadi mudah begitu katalog terurut menurut kategori: setiap kategori merupakan satu deretan yang tidak terputus.
- **Penggabungan.** Dua daftar terurut dapat digabung menjadi satu daftar terurut dalam satu lintasan, dan itulah jantung *merge sort* (Subbab 3.8) serta setiap basis data yang menggabungkan dua tabel.
- **Peringkat dan k teratas.** Papan peringkat, sepuluh pesanan laundry terberat, harga median: semuanya pertanyaan tentang urutan.
- **Manusia membaca daftar yang terurut.** Buku telepon, kamus, daftar peserta kelas, dan rekening koran semuanya terurut karena orang tidak sanggup memindai daftar seribu entri yang tidak beraturan.

Karena pengurutan ada di mana-mana, ia juga salah satu masalah yang paling banyak dikaji dalam komputasi. Knuth mencurahkan sebagian besar dari satu jilid setebal 780 halaman untuknya [1], dan algoritma-algoritma di bab ini telah disempurnakan selama lebih dari tujuh puluh tahun. Sejarah panjang itu kabar baik bagi kita: gagasannya bersih, dipahami dengan baik, dan terdokumentasi dengan baik.

3.2.3 Enam keluarga metode pengurutan

Algoritma-algoritma pengurutan terbagi menjadi beberapa keluarga menurut gagasan yang menggerakkannya. **Gambar 3.1** mengelompokkannya; nomor subbab menunjukkan di mana masing-masing diajarkan.



Gambar 3.1 Keluarga metode pengurutan dalam bab ini. Lima yang pertama membandingkan kunci; yang terakhir langsung memakai digit-digit kunci.

1. Metode **penukaran** (*exchange*) membandingkan dua butir dan menukarnya bila urutannya salah: *bubble sort* dan *exchange sort*.

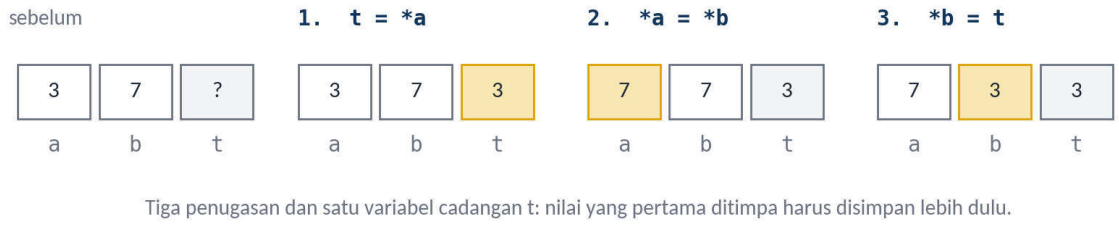
2. Metode **seleksi** (*selection*) berulang kali memilih butir terkecil (atau terbesar) yang tersisa lalu meletakkannya di tempatnya: *selection sort*, dan *heap sort*, yang memakai pohon agar pemilihan berlangsung cepat.
3. Metode **penyisipan** (*insertion*) mengambil butir satu per satu dan menyisipkan masing-masing ke tempatnya di antara butir yang sudah terurut: *insertion sort*, dan *tree sort*, yang menyisipkan ke pohon pencarian biner (Bab 11).
4. **Inkremen menyusut** (*diminishing increment*): *Shell sort*, yang menjalankan *insertion sort* lebih dahulu pada butir-butir yang berjauhan, lalu pada butir yang makin berdekatan.
5. **Bagi dan kuasai** (*divide and conquer*): *merge sort* dan *quicksort* memecah masalah menjadi masalah-masalah lebih kecil yang sejenis.
6. Metode **distribusi** sama sekali tidak membandingkan kunci; butir diletakkan menurut digit atau nilai kuncinya: *counting sort* dan *radix sort*.

3.2.4 Satu gerakan yang diperlukan setiap pengurutan: menukar dua nilai

Hampir setiap algoritma dalam bab ini dibangun dari satu operasi kecil: menukar nilai di dua tempat. Kedengarannya terlalu sederhana untuk dibahas, padahal justru di sinilah pemula paling dulu keliru. Misalkan `a` berisi 3 dan `b` berisi 7, lalu kita menulis

```
a = b;     // a sekarang 7
b = a;     // b juga menjadi 7: angka 3 hilang
```

Penugasan pertama menghancurkan satu-satunya salinan angka 3. Perbaikannya adalah variabel cadangan, lazimnya bernama `t` untuk "sementara" (*temporary*), yang mengamankan nilai pertama selama nilai itu ditimpa. **Gambar 3.2** memperlihatkan ketiga penugasan tersebut.



Gambar 3.2 Menukar dua nilai dengan variabel cadangan. Setiap kotak yang disorot adalah kotak yang berubah pada langkah itu.

Dalam C++ kita menginginkan sebuah *fungsi* yang menukar dua variabel milik pemanggilnya. Fungsi yang menerima dua `int` menerima salinan, dan menukar salinan tidak mengubah apa pun di luar fungsi. Maka fungsi itu menerima **alamat** kedua variabel, sebagai pointer, dan bekerja melaluinya: `*a` berarti "variabel yang alamatnya tersimpan di `a`". Bab 1 memperkenalkan operator alamat `&` dan gagasan pointer; Bab 6 membahas pointer secara lengkap. **Kode 3.1** adalah fungsi penukar yang dapat dipakai setiap pengurutan C++, lengkap dengan uji kecilnya.

Kode 3.1 `swap_demo.cpp`: menukar dua `int` melalui pointer, dan menukar dua elemen larik.

```cpp
// swap_demo.cpp - menukar dua nilai, gerakan dasar setiap pengurutan
#include <cstdio>

void swapInts(int* a, int* b) {        // menerima dua alamat, bukan dua nilai
    int t = *a;                         // amankan nilai pertama
    *a = *b;
    *b = t;
}

void printArray(const char* label, const int data[], int n) {
    std::printf("%s", label);
    for (int i = 0; i < n; ++i) std::printf(" %d", data[i]);
    std::printf("\n");
}

int main() {
    int a = 3, b = 7;
    std::printf("before: a = %d, b = %d\n", a, b);
    swapInts(&a, &b);                   // teruskan alamat a dan b
    std::printf("after:  a = %d, b = %d\n", a, b);

    int data[] = {22, 10, 15, 3, 8, 2};
    int n = sizeof(data) / sizeof(data[0]);
    printArray("data:", data, n);
    // tukar elemen pertama dan terakhir
    swapInts(&data[0], &data[n - 1]);
    printArray("after exchanging data[0] and data[5]:", data, n);
    return 0;
}
```

Pemanggilan `swapInts(&a, &b)` meneruskan alamat `a` dan `b`; di dalam fungsi, `*a` dan `*b` adalah `a` dan `b` itu sendiri. Fungsi yang sama juga menukar dua elemen larik, karena `&data[0]` hanyalah alamat elemen pertama. Dalam C++ yang sesungguhnya Anda akan memanggil `std::swap(x, y)` dari pustaka standar, yang menerima argumennya melalui referensi dan mengerjakan tiga penugasan yang persis sama; mulai Subbab 3.3 program C++ kita memakainya.

Java tidak mempunyai pointer ke variabel lokal: setiap argumen diteruskan menurut nilai, sehingga metode yang menerima dua `int` tidak akan pernah dapat menukar variabel pemanggilnya. Yang *dapat* dilakukan sebuah metode adalah mengubah isi larik yang diterimanya, karena larik adalah objek dan metode menerima referensi kepadanya. **Kode 3.2** memperlihatkan kedua fakta itu: `swapInts` tidak berpengaruh, sedangkan `swap(data, i, j)` bekerja.

Kode 3.2 `SwapDemo.java`: metode tidak dapat menukar variabel `int` milik pemanggilnya, tetapi dapat menukar dua slot larik.

```java
// SwapDemo.java - Java meneruskan nilai, jadi swap menukar dua slot larik
public class SwapDemo {
    static void swapInts(int a, int b) {  // menerima salinan: perubahan hilang
        int t = a;
        a = b;
        b = t;
    }

    // menerima larik: perubahan bertahan
    static void swap(int[] data, int i, int j) {
        int t = data[i];
        data[i] = data[j];
        data[j] = t;
    }

    static void printArray(String label, int[] data) {
        StringBuilder sb = new StringBuilder(label);
        for (int x : data) sb.append(' ').append(x);
        System.out.println(sb);
    }

    public static void main(String[] args) {
        int a = 3, b = 7;
        System.out.println("before: a = " + a + ", b = " + b);
        // tidak berpengaruh pada a dan b
        swapInts(a, b);
        int t = a;  // jadi kita menukarnya di sini
        a = b;
        b = t;
        System.out.println("after:  a = " + a + ", b = " + b);

        int[] data = {22, 10, 15, 3, 8, 2};
        printArray("data:", data);
        // tukar elemen pertama dan terakhir
        swap(data, 0, data.length - 1);
        printArray("after exchanging data[0] and data[5]:", data);
    }
}
```

Python tidak mempunyai pointer maupun variabel primitif; sebuah nama merujuk ke sebuah objek. Penugasan tuple-nya, `a, b = b, a`, menghitung seluruh ruas kanan lebih dahulu sebelum menugaskan, sehingga tidak diperlukan variabel cadangan, dan `data[i], data[j] = data[j], data[i]` menukar dua elemen list. **Kode 3.3** adalah versi Python-nya.

Kode 3.3 `swap_demo.py`: menukar dengan penugasan tuple, untuk dua nama dan untuk dua elemen list.

```python
# swap_demo.py - Python menukar dengan penugasan tuple
def swap_ints(a, b):
    """Menerima dua referensi; mengikat ulang tak menyentuh pemanggil."""
    a, b = b, a


def swap(data, i, j):
    """Menerima list itu sendiri: perubahan slotnya terlihat oleh pemanggil."""
    data[i], data[j] = data[j], data[i]


def print_array(label, data):
    print(label, *data)


a, b = 3, 7
print(f"before: a = {a}, b = {b}")
swap_ints(a, b)                          # tidak berpengaruh pada a dan b
a, b = b, a                              # idiom Python: tukar di sini
print(f"after:  a = {a}, b = {b}")

data = [22, 10, 15, 3, 8, 2]
print_array("data:", data)
swap(data, 0, len(data) - 1)  # tukar elemen pertama dan terakhir
print_array("after exchanging data[0] and data[5]:", data)
```

Ketiga program mencetak baris yang sama:

```
before: a = 3, b = 7
after:  a = 7, b = 3
data: 22 10 15 3 8 2
after exchanging data[0] and data[5]: 2 10 15 3 8 22
```

3.2.5 Menghitung kerja: perbandingan dan pertukaran

Untuk membandingkan algoritma pengurutan kita perlu mengukurnya, dan "waktu dalam milidetik" adalah ukuran pertama yang buruk: nilainya bergantung pada komputer, kompilator, bahasa, dan apa pun yang sedang dikerjakan mesin. Sebagai gantinya kita menghitung dua operasi yang membentuk sebuah pengurutan.

- **Perbandingan** (*comparison*) menanyakan "apakah x sebelum y?" tentang dua kunci.
- **Pertukaran** (*swap*) menukar dua butir; **pemindahan** (*move*) atau **pergeseran** (*shift*) menyalin satu butir ke tempat lain. Satu pertukaran kira-kira setara dengan tiga pemindahan.

Hitungan ini tidak bergantung pada mesin, dan pertumbuhannya terhadap n dapat diramalkan dengan aljabar. Bab 2 memperkenalkan **notasi big-O** untuk keperluan ini: kita katakan sebuah algoritma adalah $O(f(n))$ bila banyak langkahnya paling banyak suatu konstanta kali $f(n)$ untuk semua n yang besar. Dalam bab ini kita bertemu dua laju pertumbuhan yang belum Anda perlukan sebelumnya. Sebuah algoritma disebut **kuadratik**, $O(n^2)$, bila menggandakan n kira-kira melipatempatkan kerjanya; disebut **linearitmik**, $O(n \log n)$, bila menggandakan n membuat kerjanya sedikit lebih dari dua kali lipat. **Tabel 3.1** memperlihatkan mengapa perbedaan itu menentukan segalanya begitu n besar. Di sini $\lg n$ berarti $\log_2 n$, yaitu berapa kali n dapat dibagi dua sebelum mencapai 1.

Tabel 3.1 Pertumbuhan $n\lg n$ dan $n^2/2$. Pada satu miliar butir, komputer yang mengerjakan 10^9 langkah per detik memerlukan 30 detik untuk $n\lg n$ langkah dan 16 tahun untuk $n^2/2$ langkah.

n	$n\lg n$	$n^2/2$	Rasio
10	33	50	1.5
1,000	9,966	500,000	50
1,000,000	19,931,569	5×10^{11}	25,000
10^9	2.99×10^{10}	5×10^{17}	1.7×10^7

Ada satu hitungan lagi yang mengalir di sepanjang bab. Sebuah **inversi** dalam barisan adalah sepasang posisi yang urutannya satu terhadap yang lain salah.

> **DEFINISI.** Sebuah **inversi** pada $a_0, \ldots, a_{n-1}$ adalah pasangan indeks (i, j) dengan $i < j$ dan $a_i > a_j$. Banyaknya inversi adalah

$$\text{inv}(a) = \left|\{(i,j) : 0 \le i < j \le n-1,\ a_i > a_j\}\right|. \tag{3.2}$$

Barisan terurut mempunyai 0 inversi. Barisan yang urutannya terbalik sempurna mempunyai setiap pasangan terbalik, yaitu $n(n-1)/2$ inversi. Larik berjalan kita mempunyai

$$\underbrace{5}_{22} + \underbrace{3}_{10:\ 3,8,2} + \underbrace{3}_{15:\ 3,8,2} + \underbrace{1}_{3:\ 2} + \underbrace{1}_{8:\ 2} + \underbrace{0}_{2} = 13$$

inversi: 22 lebih besar daripada kelima bilangan sesudahnya, 10 lebih besar daripada tiga dari empat bilangan sesudahnya, dan seterusnya. Ingatlah angka 13 ini; ia akan muncul lagi sebagai banyaknya pertukaran yang tepat dilakukan *bubble sort* dan *exchange sort* serta banyaknya pergeseran yang tepat dilakukan *insertion sort*, dan Soal 3.14 menghitung inversi dalam waktu $O(n\lg n)$ dengan sedikit perubahan pada *merge sort*.

3.2.6 Tiga sifat: di tempat, stabil, adaptif

Tiga istilah lagi menggambarkan *bagaimana* sebuah pengurutan berperilaku, di luar seberapa cepat ia bekerja.

> **DEFINISI.** Sebuah pengurutan disebut **di tempat** (*in-place*) bila ia hanya memakai memori tambahan sebesar konstanta di luar larik itu sendiri: beberapa variabel indeks dan satu variabel sementara, seberapa pun besar n. Ia disebut **stabil** (*stable*) bila butir-butir dengan kunci yang sama keluar dalam urutan relatif yang sama seperti saat masuk. Ia disebut **adaptif** (*adaptive*) bila ia bekerja lebih sedikit pada masukan yang sudah hampir terurut.

Sifat di tempat penting bila memori terbatas atau datanya sangat besar. Kestabilan penting setiap kali sebuah rekaman diurutkan menurut satu kunci setelah sebelumnya ditata menurut kunci lain: bila katalog Pustaka Ceria tersusun menurut id buku lalu kita urutkan menurut kategori, pengurutan yang stabil mempertahankan buku-buku setiap kategori dalam urutan id, sedangkan pengurutan yang tidak stabil bisa mengacaknya. Subbab 3.12 memperlihatkan persis hal itu terjadi. Adaptivitas penting karena data nyata sangat sering *hampir* terurut: daftar terurut kemarin ditambah beberapa entri baru.

3.3 Bubble Sort

3.3.1 Gagasannya

Bubble sort adalah metode yang ditemukan sendiri oleh kebanyakan orang ketika diminta mengurutkan sederet bilangan dengan aturan "Anda hanya boleh melihat dua tetangga sekaligus". Susuri deretan dari kiri ke kanan. Pada setiap langkah bandingkan elemen sekarang dengan elemen berikutnya; bila urutannya salah, tukar keduanya. Setelah sampai di ujung, mulai lagi dari kiri.

Mengapa cara ini berhasil? Ikuti bilangan terbesar. Saat penyusuran pertama kali mencapainya, bilangan itu lebih besar daripada tetangga kanannya, sehingga ia ditukar satu tempat ke kanan; di posisi barunya ia kembali lebih besar daripada tetangga kanannya, sehingga ditukar lagi; begitu seterusnya sampai ia mencapai posisi terakhir. Satu penyusuran lengkap, yang disebut **lintasan** (*pass*), dengan demikian membawa elemen terbesar sampai ke ujung, tempat ia seharusnya berada. Namanya berasal dari gambaran ini: nilai-nilai besar naik melalui larik selangkah demi selangkah seperti gelembung yang naik di dalam gelas minuman bersoda. Astrachan menelusuri asal nama itu dan sejarah algoritmanya yang ternyata panjang dalam [2].

Setelah lintasan pertama elemen terbesar sudah di tempatnya, sehingga lintasan kedua dapat berhenti satu posisi lebih awal; lintasan itu membawa elemen terbesar kedua ke posisi kedua dari akhir. Secara umum, setelah lintasan p, p posisi terakhir memuat p elemen terbesar dalam keadaan terurut. Kalimat itu adalah **invarian gelung** (*loop invariant*) *bubble sort*, fakta yang benar setelah setiap lintasan dan yang membuat algoritmanya benar: setelah $n-1$ lintasan, $n-1$ posisi terakhir sudah benar, dan satu elemen yang tersisa, di posisi 0, pastilah yang terkecil.

3.3.2 Keluar dini

Masih ada satu pengamatan lagi, dan inilah yang membedakan *bubble sort* yang cermat dari yang sembarangan. Bila satu lintasan penuh **sama sekali tidak menukar apa pun**, setiap elemen sudah tidak lebih besar daripada tetangga kanannya, dan menurut Persamaan (3.1) larik itu sudah terurut. Tidak ada alasan menjalankan lintasan-lintasan sisanya. Sebuah penanda, `swapped`, mencatat apakah lintasan sekarang menukar sesuatu, dan algoritma berhenti begitu sebuah lintasan berakhir dengan penanda yang masih bernilai salah. **Pseudokode 3.1** menuliskan seluruh prosedurnya; indeks berjalan dari 0 sampai $n-1$.

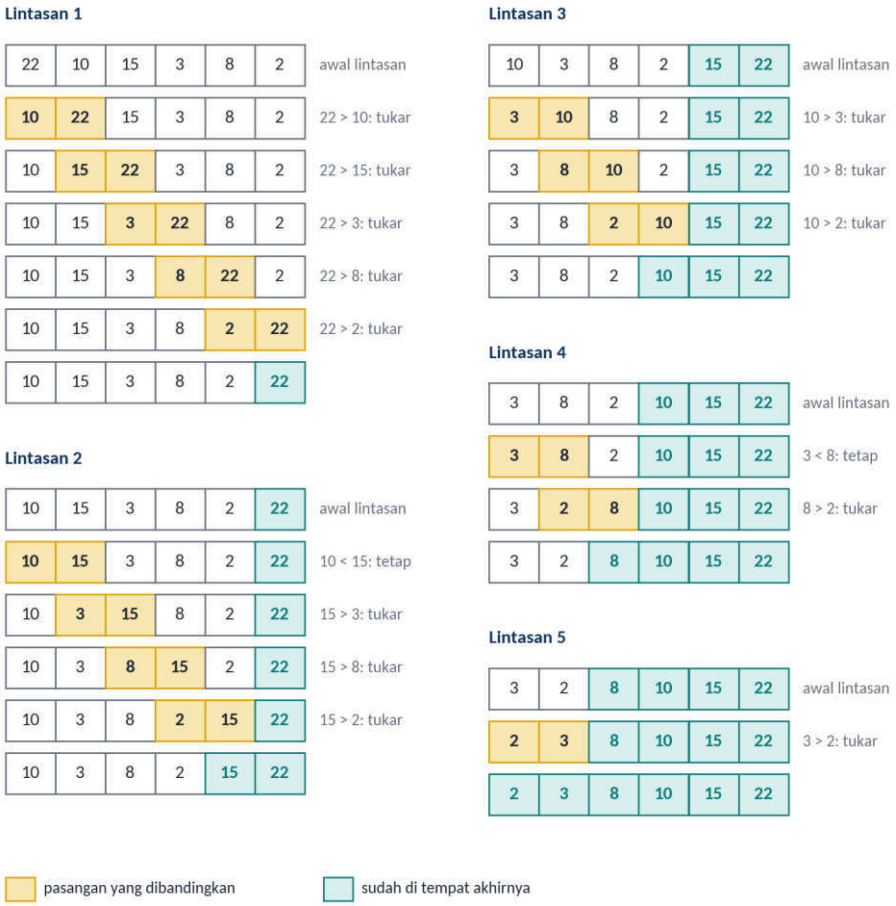
Pseudokode 3.1 *Bubble sort* dengan keluar dini: setiap lintasan membawa elemen terbesar yang tersisa ke ujung, dan lintasan tanpa pertukaran mengakhiri pengurutan.

```
BUBBLE-SORT(A, n)
 1  for pass = 1 to n - 1
 2      swapped = FALSE
 3      for j = 0 to n - pass - 1          // ekornya sudah di tempatnya
 4          if A[j] > A[j+1]
 5              exchange A[j] with A[j+1]
 6              swapped = TRUE
 7      if swapped == FALSE                  // lintasan tanpa tukar: terurut
 8          return
```

3.3.3 Jejak lengkap

Gambar 3.3 menjalankan **Pseudokode 3.1** pada $A = [22,10,15,3,8,2]$. Setiap baris memperlihatkan larik *sesudah* satu perbandingan; dua sel berwarna kuning tua adalah pasangan

yang baru dibandingkan, dan catatannya menyebutkan apakah keduanya ditukar. Sel berwarna hijau toska sudah mencapai posisi akhirnya.



Gambar 3.3 *Bubble sort* pada $A = [22,10,15,3,8,2]$, satu baris per perbandingan. Lintasan 1 membawa 22 ke ujung, lintasan 2 membawa 15, dan seterusnya; lima lintasan dan lima belas perbandingan seluruhnya.

Bacalah lintasan 1 perlahan-lahan. $22 > 10$, tukar: [10,22,15,3,8,2]. Sekarang 22 berada di indeks 1 dan bertemu 15: tukar. Ia bertemu 3, 8, dan 2 bergantian dan selalu ditukar, dan setelah lima perbandingan ia berada di indeks 5. Lintasan 2 mulai lagi dari kiri: $10 < 15$, jadi tidak terjadi apa-apa; lalu 15 bergerak ke kanan melewati 3, 8, dan 2 dan berhenti di sebelah 22. Lintasan 3 membawa 10 ke indeks 3; lintasan 4 membandingkan 3 dengan 8 (tanpa tukar) lalu memindahkan 8 melewati 2; lintasan 5 akhirnya menukar 3 dan 2. Perhatikan bahwa bilangan kecil 2, yang mulai dari ujung paling kanan, hanya bergerak *satu* langkah ke kiri pada setiap lintasan. Bilangan besar melaju cepat ke kanan, sedangkan bilangan kecil merangkak ke kiri; Soal 3.4 membangun varian yang memperbaiki ketimpangan ini.

3.3.4 Programnya

Kode 3.4 mengimplementasikan **Pseudokode 3.1** dalam C++. Parameter `ascending` memilih arah: untuk urutan menurun hanya perbandingannya yang berubah, dari `a[j] > a[j+1]` menjadi `a[j] < a[j+1]`. Fungsi itu menghitung perbandingan dan pertukarannya serta dapat mencetak larik setelah setiap lintasan. `main` menjalankannya pada larik berjalan, lalu pada hasil yang sudah terurut (untuk melihat keluar dini), pada larik terbalik (kasus terburuk), dan akhirnya dalam urutan menurun.

Kode 3.4 `bubble_sort.cpp`: *bubble sort* dengan keluar dini, menaik atau menurun, dengan hitungan perbandingan dan pertukaran.

```cpp
// bubble_sort.cpp - bubble sort dengan keluar dini, dilacak per lintasan
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

// Mengurutkan a secara menaik (atau menurun bila ascending == false).
void bubbleSort(vector<int>& a, bool ascending, bool trace) {
    long long comparisons = 0, swaps = 0;
    int n = static_cast<int>(a.size());
    for (int pass = 1; pass < n; ++pass) {
        bool swapped = false;
        // pass-1 slot terakhir sudah final
        for (int j = 0; j < n - pass; ++j) {
            ++comparisons;
            bool outOfOrder = ascending ? a[j] > a[j + 1] : a[j] < a[j + 1];
            if (outOfOrder) {
                swap(a[j], a[j + 1]);
                ++swaps;
                swapped = true;
            }
        }
        if (trace)
            cout << "  pass " << pass << ": " << show(a)
                 << (swapped ? "" : "  no swap, stop") << "\n";
        if (!swapped) break;  // keluar dini: tak ada yang bergerak
    }
    cout << "  result " << show(a) << ", comparisons = " << comparisons
         << ", swaps = " << swaps << "\n";
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Bubble sort, ascending, on " << show(data) << "\n";
    bubbleSort(data, true, true);
```

Kode 3.4 (Lanjutan) *bubble_sort.cpp: bubble sort dengan keluar dini, menaik atau menurun, dengan hitungan perbandingan dan pertukaran.*

```cpp
    cout << "Already sorted input " << show(data) << "\n";
    // kasus terbaik: satu lintasan
    bubbleSort(data, true, true);

    vector<int> reversed = {22, 15, 10, 8, 3, 2};
    cout << "Reversed input " << show(reversed) << "\n";
    // kasus terburuk: setiap pasangan ditukar
    bubbleSort(reversed, true, false);

    vector<int> again = {22, 10, 15, 3, 8, 2};
    cout << "Bubble sort, descending, on " << show(again) << "\n";
    bubbleSort(again, false, false);
    return 0;
}
```

Kode 3.5 adalah program yang sama dalam Java. Java tidak mempunyai `std::swap` untuk elemen larik, sehingga pertukarannya ditulis lengkap dengan variabel sementara `t`, persis seperti pada **Gambar 3.2**.

Kode 3.5 `BubbleSort.java`: *bubble sort* yang sama dalam Java.

```java
// BubbleSort.java - bubble sort dengan keluar dini, dilacak per lintasan
public class BubbleSort {
    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    // Mengurutkan a secara menaik (atau menurun bila ascending == false).
    static void bubbleSort(int[] a, boolean ascending, boolean trace) {
        long comparisons = 0, swaps = 0;
        int n = a.length;
        for (int pass = 1; pass < n; pass++) {
            boolean swapped = false;
            // pass-1 slot terakhir sudah final
            for (int j = 0; j < n - pass; j++) {
                comparisons++;
                boolean outOfOrder =
                        ascending ? a[j] > a[j + 1] : a[j] < a[j + 1];
                if (outOfOrder) {
                    int t = a[j]; a[j] = a[j + 1]; a[j + 1] = t;
                    swaps++;
                    swapped = true;
                }
            }
            if (trace)
                System.out.println("  pass " + pass + ": " + show(a)
                        + (swapped ? "" : "  no swap, stop"));
            if (!swapped) break;  // keluar dini: tak ada yang bergerak
        }
        System.out.println("  result " + show(a) + ", comparisons = "
                + comparisons
                + ", swaps = " + swaps);
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println("Bubble sort, ascending, on " + show(data));
        bubbleSort(data, true, true);

        System.out.println("Already sorted input " + show(data));
        // kasus terbaik: satu lintasan
        bubbleSort(data, true, true);
```

Kode 3.5 (Lanjutan) *BubbleSort.java: bubble sort yang sama dalam Java.*

```java
        int[] reversed = {22, 15, 10, 8, 3, 2};
        System.out.println("Reversed input " + show(reversed));
        // kasus terburuk: setiap pasangan ditukar
        bubbleSort(reversed, true, false);

        int[] again = {22, 10, 15, 3, 8, 2};
        System.out.println("Bubble sort, descending, on " + show(again));
        bubbleSort(again, false, false);
    }
}
```

Kode 3.6 adalah versi Python-nya. `pass` adalah kata kunci dalam Python, sehingga pencacah lintasannya bernama `pass_no`.

Kode 3.6 `bubble_sort.py`: *bubble sort* yang sama dalam Python.

```python
# bubble_sort.py - bubble sort dengan keluar dini, dilacak per lintasan
def show(a):
    return "[" + " ".join(map(str, a)) + "]"


def bubble_sort(a, ascending=True, trace=False):
    """Mengurutkan list a di tempat, menaik (atau menurun)."""
    comparisons = swaps = 0
    n = len(a)
    for pass_no in range(1, n):
        swapped = False
        for j in range(n - pass_no):     # pass_no-1 slot terakhir sudah final
            comparisons += 1
            out_of_order = a[j] > a[j + 1] if ascending else a[j] < a[j + 1]
            if out_of_order:
                a[j], a[j + 1] = a[j + 1], a[j]
                swaps += 1
                swapped = True
        if trace:
            note = "" if swapped else "  no swap, stop"
            print(f"  pass {pass_no}: {show(a)}{note}")
        if not swapped:                  # keluar dini: tak ada yang bergerak
            break
    print(f"  result {show(a)}, comparisons = {comparisons}, swaps = {swaps}")


data = [22, 10, 15, 3, 8, 2]
print(f"Bubble sort, ascending, on {show(data)}")
bubble_sort(data, True, True)

print(f"Already sorted input {show(data)}")
bubble_sort(data, True, True)            # kasus terbaik: satu lintasan

reversed_data = [22, 15, 10, 8, 3, 2]
print(f"Reversed input {show(reversed_data)}")
# kasus terburuk: setiap pasangan ditukar
bubble_sort(reversed_data, True, False)

again = [22, 10, 15, 3, 8, 2]
print(f"Bubble sort, descending, on {show(again)}")
bubble_sort(again, False, False)
```

Ketiga program mencetak keluaran yang identik:

```
Bubble sort, ascending, on [22 10 15 3 8 2]
  pass 1: [10 15 3 8 2 22]
  pass 2: [10 3 8 2 15 22]
  pass 3: [3 8 2 10 15 22]
  pass 4: [3 2 8 10 15 22]
  pass 5: [2 3 8 10 15 22]
  result [2 3 8 10 15 22], comparisons = 15, swaps = 13
Already sorted input [2 3 8 10 15 22]
  pass 1: [2 3 8 10 15 22]  no swap, stop
  result [2 3 8 10 15 22], comparisons = 5, swaps = 0
Reversed input [22 15 10 8 3 2]
  result [2 3 8 10 15 22], comparisons = 15, swaps = 15
Bubble sort, descending, on [22 10 15 3 8 2]
  result [22 15 10 8 3 2], comparisons = 9, swaps = 2
```

Baris-baris jejaknya cocok dengan **Gambar 3.3**, baris demi baris, pada akhir setiap lintasan. Jalannya yang kedua memperlihatkan keluar dini bekerja: larik yang sudah terurut hanya menghabiskan satu lintasan berisi $n - 1 = 5$ perbandingan tanpa pertukaran. Larik terbalik menghabiskan semua perbandingan, dan setiap perbandingannya berujung pertukaran.

3.3.5 Tepatnya berapa banyak kerja

Mari kita menghitung, karena hitungan adalah inti bab ini.

Perbandingan pada kasus terburuk. Lintasan p membandingkan posisi 0 dan 1, 1 dan 2, ..., sampai $n-p-1$ dan $n-p$: itu $n-p$ perbandingan. Bila tidak ada lintasan yang berakhir dini, totalnya

$$C_{\text{worst}}(n) = \sum_{p=1}^{n-1} (n-p) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}. \tag{3.3}$$

Langkah terakhir adalah jumlah klasik $n-1$ bilangan asli pertama: tulis jumlah itu maju dan mundur, jumlahkan kedua baris kolom demi kolom, dan setiap kolom menghasilkan n; ada $n-1$ kolom, dan jumlahnya terhitung dua kali. Untuk $n=6$ hasilnya $6 \cdot 5/2 = 15$, bilangan yang dicetak program. Untuk $n = 100{,}000$ hasilnya hampir tepat lima miliar.

Perbandingan pada kasus terbaik. Pada masukan terurut, lintasan pertama melakukan $n-1$ perbandingan, tidak menukar apa pun, dan algoritma berhenti. Jadi *bubble sort dengan keluar dini* bersifat adaptif, dengan kasus terbaik $n-1$ perbandingan, $O(n)$. Tanpa penanda itu ia akan melakukan $n(n-1)/2$ perbandingan bahkan pada masukan terurut.

Pertukaran, pada setiap kasus. Inilah fakta yang indah. Menukar dua elemen *bertetangga* $a_j > a_{j+1}$ hanya mengubah urutan relatif tepat satu pasangan, yaitu pasangan itu sendiri; setiap elemen lain tetap berada di sisi yang sama terhadap keduanya. Jadi setiap pertukaran yang dilakukan *bubble sort* menghapus tepat satu inversi, dan ia berhenti ketika tidak ada lagi yang tersisa. Karena itu

$$\text{pertukaran oleh bubble sort} = \text{inv}(A). \tag{3.4}$$

Untuk larik kita hasilnya 13, sesuai yang dicetak; untuk larik terbalik hasilnya $15 = 6 \cdot 5/2$, yaitu setiap pasangan.

Kasus rata-rata. Bila semua $n!$ urutan dari n kunci yang berbeda sama mungkinnya, setiap pasangan dari $n(n-1)/2$ pasangan itu terbalik pada tepat separuh urutan (membalik suatu urutan membalik tepat pasangan-pasangan yang tadinya tidak terbalik), sehingga rata-rata banyaknya inversi, dan menurut Persamaan (3.4) rata-rata banyaknya pertukaran, adalah

$$\overline{\text{inv}}(n) = \frac{1}{2} \cdot \frac{n(n-1)}{2} = \frac{n(n-1)}{4}. \tag{3.5}$$

Separuh kasus terburuk tetap kuadratik: *bubble sort* adalah $\Theta(n^2)$ pada kasus rata-rata. (Lambang Θ berarti "tumbuh persis seperti", dibatasi dari atas *dan* dari bawah oleh kelipatan konstanta.)

> **WAWASAN KUNCI:** Setiap algoritma yang hanya pernah menukar elemen *bertetangga* menghapus paling banyak satu inversi per pertukaran, sehingga menurut Persamaan (3.5) ia memerlukan rata-rata $n(n-1)/4$ pertukaran. Secerdik apa pun di dalam aturan itu, ia tidak dapat lebih cepat daripada kuadratik. Untuk mengalahkan n^2, algoritma harus memindahkan elemen sejauh-jauhnya dalam satu langkah. *Shell sort*, *merge sort*, dan *quicksort* semuanya melakukan hal itu.

3.3.6 Bentuk-bentuk lain yang akan Anda jumpai

Bubble sort kadang ditulis dengan gelung dalam yang berjalan dari kanan ke kiri, `for j = n - 1 down to pass`, membandingkan `A[j]` dengan `A[j-1]`. Maka setiap lintasan membawa elemen *terkecil* yang tersisa ke *depan*, alih-alih membawa yang terbesar ke belakang. Kedua bentuk mengerjakan jumlah kerja yang persis sama. Bentuk mana pun mengurutkan secara menurun bila perbandingannya dibalik, dan program di atas mengerjakan persis hal itu melalui parameter `ascending`. *Bubble sort* bersifat di tempat (ia hanya memerlukan variabel sementara untuk pertukaran) dan stabil (ia menukar hanya pada `>` yang tegas, sehingga dua tetangga yang sama tidak pernah ditukar).

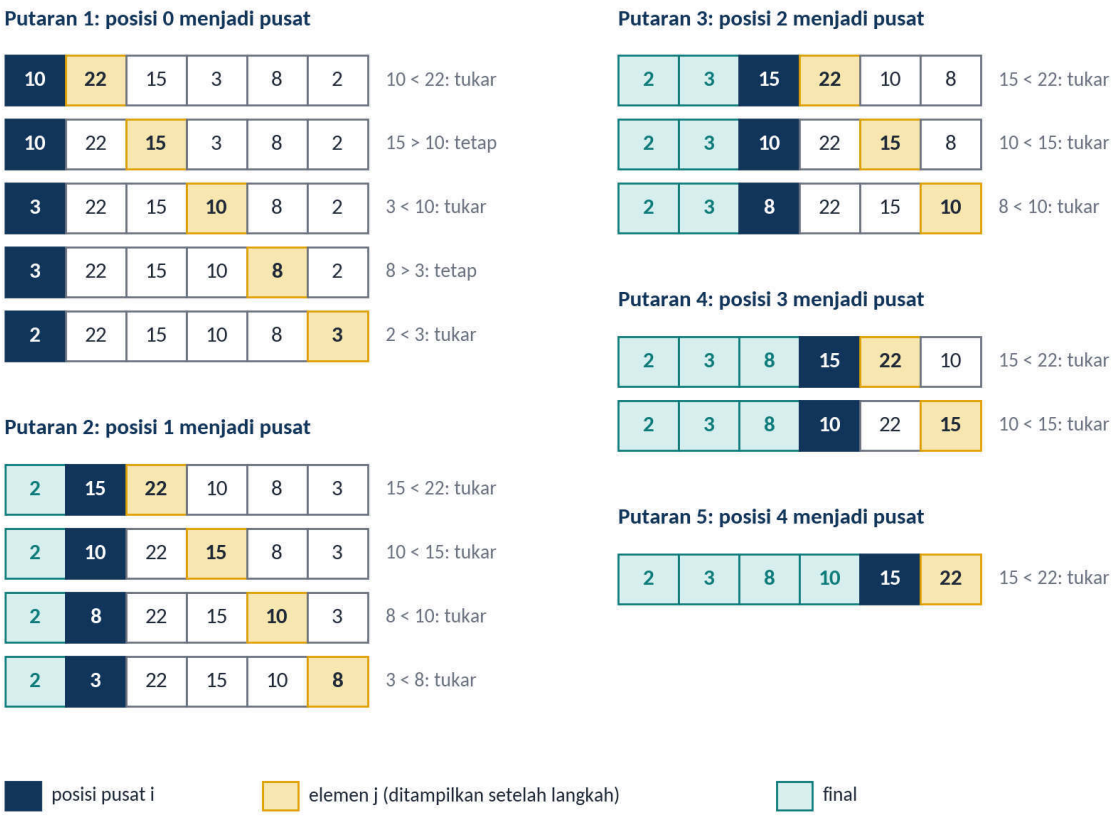
3.4 Exchange Sort

Exchange sort adalah saudara dekat *bubble sort*, dan banyak buku memakai kedua nama itu secara bergantian. Perbedaannya terletak pada pasangan *mana* yang dibandingkan, seperti diperlihatkan **Pseudokode 3.2**. *Bubble sort* selalu membandingkan tetangga, dan elemen yang dibawa berganti selama ia bergerak. *Exchange sort* menetapkan satu posisi i, yang boleh kita sebut posisi **pusat** (*pivot*) putaran itu, lalu membandingkannya dengan *setiap* posisi sesudahnya $j = i + 1, \ldots, n - 1$, dan menukar bila `A[j]` lebih kecil daripada `A[i]`. Setelah putaran itu, posisi i memuat nilai terkecil dari $A[i..n-1]$.

Pseudokode 3.2 *Exchange sort*: posisi i dibandingkan dengan setiap posisi sesudahnya dan mengambil nilai lebih kecil yang dijumpainya.

```
EXCHANGE-SORT(A, n)
 1  for i = 0 to n - 2
 2      for j = i + 1 to n - 1
 3          if A[j] < A[i]
 4              exchange A[i] with A[j]
```

Gambar 3.4 adalah jejaknya. Sel biru tua adalah posisi i; sel kuning tua adalah posisi j, ditampilkan sesudah langkah itu.



Gambar 3.4 *Exchange sort* pada A. Pada putaran 1 nilai di posisi 0 berganti tiga kali, 22 → 10 → 3 → 2, sebelum nilai terkecil menetap di sana.

Pada putaran 1, posisi 0 memuat 22; 10 lebih kecil, jadi keduanya ditukar dan posisi 0 memuat 10; 15 tidak lebih kecil; 3 lebih kecil, jadi posisi 0 kini memuat 3; 8 tidak; 2 lebih kecil, dan posisi 0 akhirnya memuat 2. Setiap putaran adalah pencarian minimum kecil yang calon sementaranya disimpan di posisi i itu sendiri, dibayar dengan satu pertukaran setiap kali muncul calon yang lebih baik. **Kode 3.7**, **Kode 3.8**, dan **Kode 3.9** mencetak setiap pertukaran.

Kode 3.7 `exchange_sort.cpp`: *exchange sort*, mencetak setiap pertukaran dan larik setelah setiap putaran.

```cpp
// exchange_sort.cpp - exchange sort: posisi i bertemu setiap elemen sesudahnya
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

void exchangeSort(vector<int>& a) {
    long long comparisons = 0, swaps = 0;
    int n = static_cast<int>(a.size());
    for (int i = 0; i < n - 1; ++i) {
        for (int j = i + 1; j < n; ++j) {
            ++comparisons;
            if (a[j] < a[i]) {  // a[i] adalah "pusat" putaran ini
                swap(a[i], a[j]);
                ++swaps;
                cout << "    swap a[" << i << "] and a[" << j << "]: "
                     << show(a) << "\n";
            }
        }
        cout << "  round " << i + 1 << " done: " << show(a) << "\n";
    }
    cout << "  comparisons = " << comparisons << ", swaps = " << swaps << "\n";
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Exchange sort on " << show(data) << "\n";
    exchangeSort(data);
    return 0;
}
```

Kode 3.8 `ExchangeSort.java`: *exchange sort* dalam Java.

```java
// ExchangeSort.java - exchange sort: posisi i bertemu setiap elemen sesudahnya
public class ExchangeSort {
    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static void exchangeSort(int[] a) {
        long comparisons = 0, swaps = 0;
        int n = a.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = i + 1; j < n; j++) {
                comparisons++;
                if (a[j] < a[i]) {  // a[i] adalah "pusat" putaran ini
                    int t = a[i]; a[i] = a[j]; a[j] = t;
                    swaps++;
                    System.out.println("    swap a[" + i + "] and a[" + j
                            + "]: " + show(a));
                }
            }
            System.out.println("  round " + (i + 1) + " done: " + show(a));
        }
        System.out.println("  comparisons = " + comparisons + ", swaps = "
                + swaps);
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println("Exchange sort on " + show(data));
        exchangeSort(data);
    }
}
```

Kode 3.9 `exchange_sort.py`: *exchange sort* dalam Python.

```python
# exchange_sort.py - exchange sort: posisi i bertemu setiap elemen sesudahnya
def show(a):
    return "[" + " ".join(map(str, a)) + "]"


def exchange_sort(a):
    comparisons = swaps = 0
    n = len(a)
    for i in range(n - 1):
        for j in range(i + 1, n):
            comparisons += 1
            if a[j] < a[i]:              # a[i] adalah "pusat" putaran ini
                a[i], a[j] = a[j], a[i]
                swaps += 1
                print(f"    swap a[{i}] and a[{j}]: {show(a)}")
        print(f"  round {i + 1} done: {show(a)}")
    print(f"  comparisons = {comparisons}, swaps = {swaps}")


data = [22, 10, 15, 3, 8, 2]
print(f"Exchange sort on {show(data)}")
exchange_sort(data)
```

```
Exchange sort on [22 10 15 3 8 2]
    swap a[0] and a[1]: [10 22 15 3 8 2]
    swap a[0] and a[3]: [3 22 15 10 8 2]
    swap a[0] and a[5]: [2 22 15 10 8 3]
  round 1 done: [2 22 15 10 8 3]
    swap a[1] and a[2]: [2 15 22 10 8 3]
    swap a[1] and a[3]: [2 10 22 15 8 3]
    swap a[1] and a[4]: [2 8 22 15 10 3]
    swap a[1] and a[5]: [2 3 22 15 10 8]
  round 2 done: [2 3 22 15 10 8]
    swap a[2] and a[3]: [2 3 15 22 10 8]
    swap a[2] and a[4]: [2 3 10 22 15 8]
    swap a[2] and a[5]: [2 3 8 22 15 10]
  round 3 done: [2 3 8 22 15 10]
    swap a[3] and a[4]: [2 3 8 15 22 10]
    swap a[3] and a[5]: [2 3 8 10 22 15]
  round 4 done: [2 3 8 10 22 15]
    swap a[4] and a[5]: [2 3 8 10 15 22]
  round 5 done: [2 3 8 10 15 22]
  comparisons = 15, swaps = 13
```

Kedua gelung bersarang itu selalu berjalan penuh, apa pun datanya: putaran i melakukan $n-1-i$ perbandingan, dan totalnya kembali $n(n-1)/2$ menurut jumlah yang sama seperti Persamaan (3.3). *Exchange sort* tidak mempunyai keluar dini dan memerlukan $\Theta(n^2)$ perbandingan pada *setiap* kasus, bahkan pada masukan terurut. Pertukarannya kebetulan 13 di sini, sama dengan *bubble sort*, tetapi itu kebetulan larik ini, bukan hukum: *exchange sort* menukar elemen yang *tidak* bertetangga, sehingga satu pertukaran dapat mengubah banyak inversi sekaligus, ke arah mana pun. Pada putaran 2 di atas, pertukaran 15 dan 22 memperbaiki satu pasangan, tetapi pertukaran 10 dengan 15 sesudahnya memindahkan 15 kembali ke belakang 22. *Exchange sort* bersifat di tempat tetapi **tidak stabil**: pertukaran jarak jauh dapat membawa sebuah elemen melewati elemen lain yang kuncinya sama.

3.5 Selection Sort

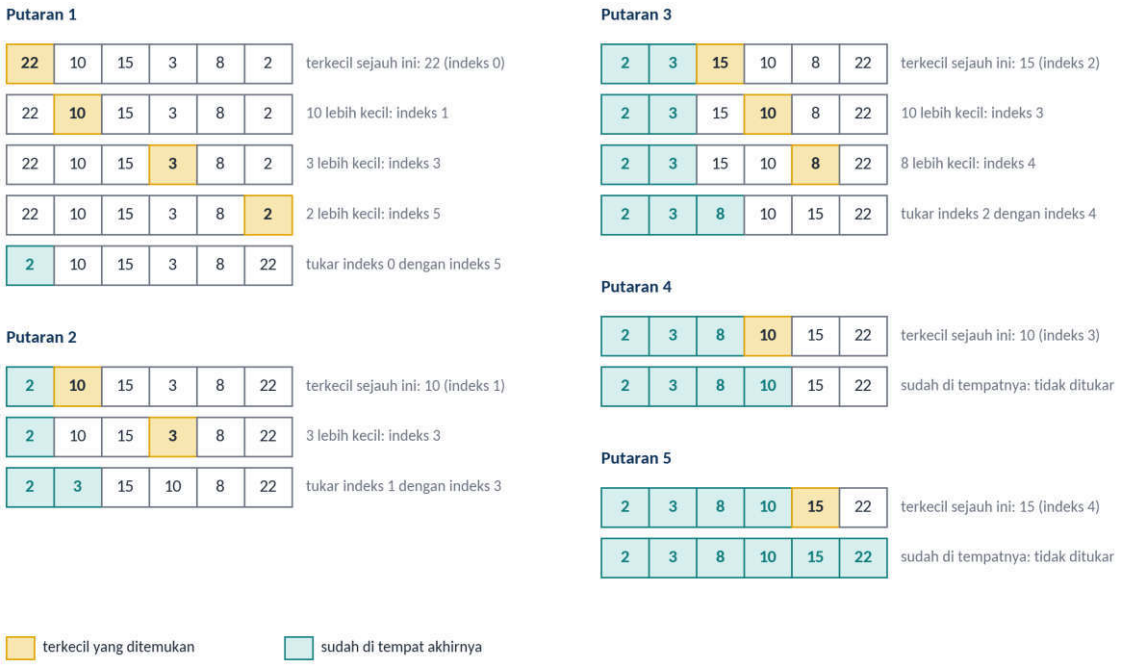
3.5.1 Cari dulu, tukar sekali

Exchange sort membuang tenaga: ia menukar ke posisi i setiap pemenang sementara, hanya untuk menukarnya keluar lagi ketika muncul yang lebih baik. **Selection sort** menghapus pemborosan itu dengan memisahkan dua pekerjaan. Mula-mula ia *mencari* posisi `pos` elemen terkecil di bagian yang belum terurut $A[i..n-1]$, dengan hanya mengingat indeksnya; lalu ia melakukan *satu* pertukaran saja, `A[i]` dengan `A[pos]`. *Selection sort* adalah pengurutan yang dibangun dari pencarian: setiap putaran dalam **Pseudokode 3.3** adalah pencarian sekuensial untuk minimum, persis penyusuran linear pada Bab 2.

Pseudokode 3.3 *Selection sort*: temukan elemen terkecil yang tersisa, lalu pindahkan ke tempatnya dengan satu pertukaran.

```
SELECTION-SORT(A, n)
 1  for i = 0 to n - 2
 2      pos = i                              // indeks terkecil sejauh ini
 3      for j = i + 1 to n - 1
 4          if A[j] < A[pos]
 5              pos = j                      // ingat; belum memindahkan
 6      if pos ≠ i
 7          exchange A[i] with A[pos]
```

Gambar 3.5 melacak keenam bilangan. Sel kuning tua adalah yang terkecil yang ditemukan sejauh ini dalam putaran itu; ia melompat ke kanan setiap kali penyusuran menemukan yang lebih kecil.



Gambar 3.5 *Selection sort* pada *A*. Putaran 1 menemukan minimum 2 di indeks 5 dan melakukan satu pertukaran; putaran 4 dan 5 mendapati minimumnya sudah di tempat dan tidak menukar apa pun.

Pada putaran 1 calonnya mula-mula 22 di indeks 0, menjadi 10 di indeks 1, 3 di indeks 3, dan 2 di indeks 5; satu pertukaran indeks 0 dan 5 menempatkan 2 di depan. Putaran 2 menyusuri [10,15,3,8,22], menemukan 3 di indeks 3, dan menukarnya dengan indeks 1. Putaran 3 menemukan 8 di indeks 4 dan menukarnya dengan indeks 2. Pada putaran 4 dan 5 elemen terkecil sudah berada di posisi *i*, dan sama sekali tidak ada pertukaran.

3.5.2 Programnya

Kode 3.10 `selection_sort.cpp`: *selection sort*, melaporkan di mana setiap putaran menemukan minimumnya.

```cpp
// selection_sort.cpp - selection sort: cari yang terkecil, lalu satu pertukaran
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

void selectionSort(vector<int>& a) {
    long long comparisons = 0, swaps = 0;
    int n = static_cast<int>(a.size());
    for (int i = 0; i < n - 1; ++i) {
        int pos = i;  // indeks yang terkecil sejauh ini
        for (int j = i + 1; j < n; ++j) {
            ++comparisons;
            if (a[j] < a[pos]) pos = j;  // cukup ingat; jangan pindahkan dulu
        }
        if (pos != i) {
            swap(a[i], a[pos]);
            ++swaps;
        }
        cout << "  round " << i + 1 << ": smallest is " << a[i]
             << " (was at index " << pos << ") -> " << show(a) << "\n";
    }
    cout << "  comparisons = " << comparisons << ", swaps = " << swaps << "\n";
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Selection sort on " << show(data) << "\n";
    selectionSort(data);
    return 0;
}
```

Kode 3.11 `SelectionSort.java`: *selection sort* dalam Java.

```java
// SelectionSort.java - selection sort: cari yang terkecil, lalu satu pertukaran
public class SelectionSort {
    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static void selectionSort(int[] a) {
        long comparisons = 0, swaps = 0;
        int n = a.length;
        for (int i = 0; i < n - 1; i++) {
            int pos = i;  // indeks yang terkecil sejauh ini
            for (int j = i + 1; j < n; j++) {
                comparisons++;
                // cukup ingat; jangan pindahkan dulu
                if (a[j] < a[pos]) pos = j;
            }
            if (pos != i) {
                int t = a[i]; a[i] = a[pos]; a[pos] = t;
                swaps++;
            }
            System.out.println("  round " + (i + 1) + ": smallest is " + a[i]
                    + " (was at index " + pos + ") -> " + show(a));
        }
        System.out.println("  comparisons = " + comparisons + ", swaps = "
                + swaps);
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println("Selection sort on " + show(data));
        selectionSort(data);
    }
}
```

Kode 3.12 `selection_sort.py`: *selection sort* dalam Python.

```python
# selection_sort.py - selection sort: cari yang terkecil, lalu satu pertukaran
def show(a):
    return "[" + " ".join(map(str, a)) + "]"


def selection_sort(a):
    comparisons = swaps = 0
    n = len(a)
    for i in range(n - 1):
        pos = i                          # indeks yang terkecil sejauh ini
        for j in range(i + 1, n):
            comparisons += 1
            if a[j] < a[pos]:
                pos = j                  # cukup ingat; jangan pindahkan dulu
        if pos != i:
            a[i], a[pos] = a[pos], a[i]
            swaps += 1
        print(f"  round {i + 1}: smallest is {a[i]} (was at index {pos}) -> "
              f"{show(a)}")
    print(f"  comparisons = {comparisons}, swaps = {swaps}")


data = [22, 10, 15, 3, 8, 2]
print(f"Selection sort on {show(data)}")
selection_sort(data)
```

Kode 3.10, **Kode 3.11**, dan **Kode 3.12** semuanya mencetak:

```
Selection sort on [22 10 15 3 8 2]
  round 1: smallest is 2 (was at index 5) -> [2 10 15 3 8 22]
  round 2: smallest is 3 (was at index 3) -> [2 3 15 10 8 22]
  round 3: smallest is 8 (was at index 4) -> [2 3 8 10 15 22]
  round 4: smallest is 10 (was at index 3) -> [2 3 8 10 15 22]
  round 5: smallest is 15 (was at index 4) -> [2 3 8 10 15 22]
  comparisons = 15, swaps = 3
```

3.5.3 Pemindahan paling sedikit, tetapi perbandingan tak pernah berkurang

Perbandingannya persis sama dengan *exchange sort*: $n(n-1)/2$ pada setiap kasus, 15 di sini, karena gelung dalam selalu menyusuri seluruh bagian yang belum terurut. Yang berubah adalah pertukarannya: paling banyak satu per putaran, jadi **paling banyak** $n-1$ **pertukaran** seluruhnya, dan hanya 3 untuk larik kita dibandingkan 13 pada *bubble sort* dan *exchange sort*. Bila memindahkan sebuah butir itu mahal, misalnya rekaman yang besar, atau operasi tulis ke memori *flash* yang aus setiap kali ditulisi, pemindahan $O(n)$ milik *selection sort* merupakan keunggulan nyata. Bila perbandinganlah yang mahal, ia sama lambatnya dengan yang lain, dan ia *tidak adaptif*: larik terurut tetap menghabiskan $n(n-1)/2$ perbandingan, karena algoritma itu tidak punya cara mengetahui bahwa tidak ada yang perlu dipindah sebelum ia melihat semuanya.

Selection sort bersifat di tempat dan, dalam bentuk di atas, **tidak stabil**. Pertukaran jarak jauh tunggal pada putaran i dapat melompatkan `A[i]` melewati elemen-elemen yang sama dengannya. Subbab 3.12 memperagakannya pada katalog Pustaka Ceria.

3.6 Insertion Sort

3.6.1 Mengurutkan kartu di tangan

Perhatikan seseorang memungut kartu remi satu per satu. Kartu-kartu yang sudah ada di tangan kiri dijaga tetap terurut. Setiap kartu baru dibandingkan dengan kartu-kartu di tangan mulai dari ujung kanan, kartu yang lebih besar digeser sedikit ke kanan untuk memberi ruang, dan kartu baru diselipkan ke celah itu. Setelah kartu terakhir dipungut, kartu di tangan sudah terurut. Itulah **insertion sort**, dan itulah pengurutan yang sungguh-sungguh dipakai orang.

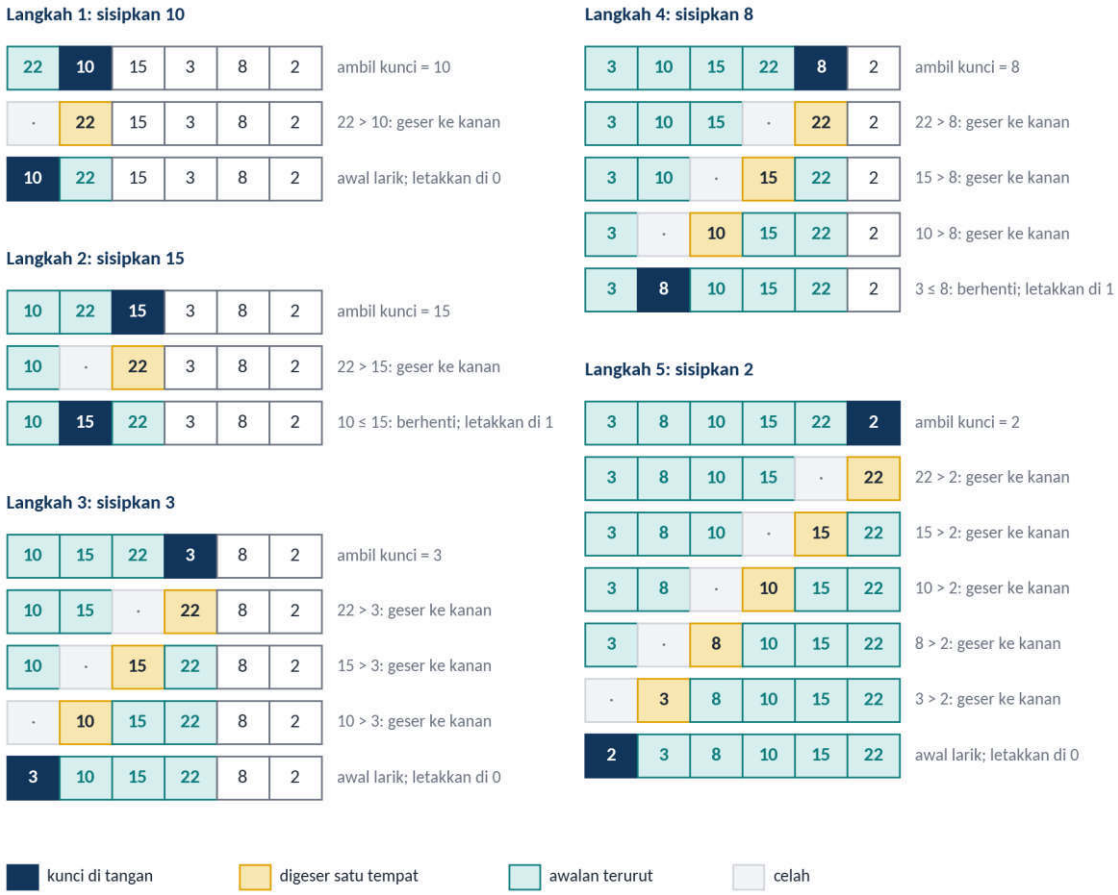
Dalam larik, "tangan" adalah awalan $A[0..i-1]$, yang dijaga tetap terurut. Langkah i mengambil elemen berikutnya, sang **kunci**, dari posisi i sehingga meninggalkan lubang di sana. Kemudian ia menyusuri awalan ke kiri: setiap elemen yang lebih besar daripada kunci digeser satu tempat ke kanan, masuk ke lubang, sehingga lubang berpindah satu tempat ke kiri. Ketika penyusuran bertemu elemen yang tidak lebih besar daripada kunci, atau keluar dari ujung kiri, kunci dijatuhkan ke dalam lubang.

Pseudokode 3.4 *Insertion sort*: tumbuhkan awalan terurut dengan menyisipkan satu kunci setiap kali, menggeser elemen yang lebih besar ke kanan.

```
INSERTION-SORT(A, n)
 1  for i = 1 to n - 1
 2      key = A[i]                          // kartu di tangan kita
 3      j = i - 1
 4      while j ≥ 0 and A[j] > key          // uji j ≥ 0 lebih dulu
 5          A[j+1] = A[j]                   // geser yang lebih besar ke kanan
 6          j = j - 1
 7      A[j+1] = key                        // jatuhkan kunci ke lubang
```

Invariannya sederhana dan kuat: pada awal langkah i, awalan $A[0..i-1]$ memuat elemen-elemen yang semula ada di sana, dalam urutan terurut. Langkah itu mempertahankannya, dan ketika i mencapai n, awalan itu adalah seluruh larik.

Gambar 3.6 memperlihatkan setiap pergeseran. Sel biru tua adalah kunci; sel pucat bertanda titik adalah lubang; sel kuning tua adalah elemen yang baru digeser ke dalamnya.



Gambar 3.6 *Insertion sort* pada A. Setiap langkah mengambil sebuah kunci, menggeser elemen yang lebih besar di awalan terurut satu tempat ke kanan, lalu menjatuhkan kunci ke celahnya.

Langkah 1 mengambil 10; $22 > 10$, jadi 22 bergeser ke kanan dan 10 jatuh ke indeks 0. Langkah 2 mengambil 15; 22 bergeser, tetapi $10 \leq 15$ menghentikan penyusuran, dan 15 mendarat di indeks 1. Langkah 3 mengambil 3, yang lebih kecil daripada semua isi awalan, sehingga 22, 15, dan 10 semuanya bergeser dan 3 mendarat di depan. Langkah 4 mengambil 8: 22, 15, dan 10 bergeser, $3 \leq 8$ menghentikan penyusuran, dan 8 mendarat di indeks 1. Langkah 5 mengambil 2, dan kelimanya bergeser.

3.6.2 Jebakan pada syarat gelung

Lihat lagi baris 4 dari **Pseudokode 3.4**: `j ≥ 0 and A[j] > key`. Urutan kedua uji itu penting. Bila kunci lebih kecil daripada semua isi awalan, j akhirnya menjadi -1, dan pada saat itu `A[j]` tidak ada. Dalam C++, Java, dan Python, operator `&&` / `and` mengevaluasi operan kirinya lebih dahulu dan sama sekali tidak mengevaluasi operan kanan bila operan kiri bernilai salah (ini disebut *short-circuit evaluation*, evaluasi hubung-singkat). Menulis `j >= 0` lebih dahulu menjamin bahwa `A[-1]` tidak pernah dibaca.

Bila ditulis terbalik, `while (data[j] > temp && j >= 0)`, program membaca `data[-1]` sebelum memeriksa. Dalam C++ itu adalah *undefined behavior* (perilaku tak terdefinisi): ia membaca

memori apa pun yang terletak sebelum larik, dan biasanya tampak berjalan baik, jenis *bug* yang paling buruk. Java melempar `ArrayIndexOutOfBoundsException`. Python lebih licin lagi: `a[-1]` sah dan berarti *elemen terakhir*, sehingga gelung membandingkan kunci dengan elemen yang salah dan dapat menggesernya ke tempat yang salah. Program-program kita menulis gelungnya sebagai `while j >= 0` dan menguji `a[j] <= key` di dalamnya, sehingga urutan itu mustahil keliru dan kita juga dapat menghitung setiap perbandingan.

3.6.3 Programnya

Kode 3.13 `insertion_sort.cpp`: *insertion sort* dengan hitungan perbandingan dan pergeseran, pada masukan acak, terurut, dan terbalik.

```cpp
// insertion_sort.cpp - insertion sort: ambil kartu, geser, letakkan di celah
#include <iostream>
#include <string>
#include <vector>
using namespace std;

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

void insertionSort(vector<int>& a, bool trace) {
    long long comparisons = 0, shifts = 0;
    int n = static_cast<int>(a.size());
    for (int i = 1; i < n; ++i) {
        int key = a[i];                              // kartu di tangan kita
        int j = i - 1;
        while (j >= 0) {  // uji j >= 0 SEBELUM membaca a[j]
            ++comparisons;
            if (a[j] <= key) break;  // kunci yang sama tetap di tempat: stabil
            a[j + 1] = a[j];  // geser yang lebih besar ke kanan
            ++shifts;
            --j;
        }
        a[j + 1] = key;
        if (trace)
            cout << "  insert " << key << " at index " << j + 1 << ": "
                 << show(a) << "\n";
    }
    cout << "  result " << show(a) << ", comparisons = " << comparisons
         << ", shifts = " << shifts << "\n";
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Insertion sort on " << show(data) << "\n";
    insertionSort(data, true);
    cout << "Already sorted input\n";
    insertionSort(data, false);  // kasus terbaik: n-1 perbandingan
    vector<int> reversed = {22, 15, 10, 8, 3, 2};
    cout << "Reversed input\n";
    insertionSort(reversed, false);                  // kasus terburuk: n(n-1)/2
    return 0;
}
```

Kode 3.14 `InsertionSort.java`: *insertion sort* dalam Java.

```java
// InsertionSort.java - insertion sort: ambil kartu, geser, letakkan di celah
public class InsertionSort {
    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static void insertionSort(int[] a, boolean trace) {
        long comparisons = 0, shifts = 0;
        for (int i = 1; i < a.length; i++) {
            int key = a[i];                          // kartu di tangan kita
            int j = i - 1;
            while (j >= 0) {  // uji j >= 0 SEBELUM membaca a[j]
                comparisons++;
                // kunci yang sama tetap di tempat: stabil
                if (a[j] <= key) break;
                a[j + 1] = a[j];  // geser yang lebih besar ke kanan
                shifts++;
                j--;
            }
            a[j + 1] = key;
            if (trace)
                System.out.println("  insert " + key + " at index " + (j + 1)
                        + ": " + show(a));
        }
        System.out.println("  result " + show(a) + ", comparisons = "
                + comparisons
                + ", shifts = " + shifts);
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println("Insertion sort on " + show(data));
        insertionSort(data, true);
        System.out.println("Already sorted input");
        insertionSort(data, false);  // kasus terbaik: n-1 perbandingan
        int[] reversed = {22, 15, 10, 8, 3, 2};
        System.out.println("Reversed input");
        insertionSort(reversed, false);              // kasus terburuk: n(n-1)/2
    }
}
```

Kode 3.15 `insertion_sort.py`: *insertion sort* dalam Python.

```python
# insertion_sort.py - insertion sort: ambil kartu, geser, letakkan di celah
def show(a):
    return "[" + " ".join(map(str, a)) + "]"


def insertion_sort(a, trace=False):
    comparisons = shifts = 0
    for i in range(1, len(a)):
        key = a[i]                      # kartu di tangan kita
        j = i - 1
        while j >= 0:                   # uji j >= 0 SEBELUM membaca a[j]
            comparisons += 1
            # kunci yang sama tetap di tempat: stabil
            if a[j] <= key:
                break
            a[j + 1] = a[j]             # geser yang lebih besar ke kanan
            shifts += 1
            j -= 1
        a[j + 1] = key
        if trace:
            print(f"  insert {key} at index {j + 1}: {show(a)}")
    print(f"  result {show(a)}, comparisons = {comparisons}, shifts = {shifts}")


data = [22, 10, 15, 3, 8, 2]
print(f"Insertion sort on {show(data)}")
insertion_sort(data, True)
print("Already sorted input")
insertion_sort(data)                    # kasus terbaik: n-1 perbandingan
print("Reversed input")
insertion_sort([22, 15, 10, 8, 3, 2])   # kasus terburuk: n(n-1)/2
```

Kode 3.13, **Kode 3.14**, dan **Kode 3.15** mencetak:

```
Insertion sort on [22 10 15 3 8 2]
  insert 10 at index 0: [10 22 15 3 8 2]
  insert 15 at index 1: [10 15 22 3 8 2]
  insert 3 at index 0: [3 10 15 22 8 2]
  insert 8 at index 1: [3 8 10 15 22 2]
  insert 2 at index 0: [2 3 8 10 15 22]
  result [2 3 8 10 15 22], comparisons = 15, shifts = 13
Already sorted input
  result [2 3 8 10 15 22], comparisons = 5, shifts = 0
Reversed input
  result [2 3 8 10 15 22], comparisons = 15, shifts = 15
```

3.6.4 Mengapa insertion sort adalah yang terbaik di antara pengurutan sederhana

Setiap pergeseran memindahkan satu elemen melewati kunci, dan kunci beserta elemen itu merupakan pasangan terbalik; setelah langkah itu keduanya berurutan, dan tidak ada pasangan lain yang berubah. Jadi, persis seperti pada *bubble sort*,

$$\text{pergeseran oleh insertion sort} = \text{inv}(A),$$

yaitu 13 untuk larik kita. Setiap langkah juga melakukan satu perbandingan yang *menghentikan* penyusuran, kecuali bila kunci berjalan sampai ke depan. Maka perbandingannya adalah inversi ditambah paling banyak $n - 1$: di sini $13 + 5 - 3 = 15$, karena tiga kunci (10, 3, dan 2) berjalan sampai ke depan dan tidak mempunyai perbandingan penghenti. Dengan menggabungkan kedua ujungnya:

$$n - 1 \;\le\; C_{\text{insertion}}(A) \;\le\; \text{inv}(A) + n - 1 \;\le\; \frac{n(n-1)}{2} + n - 1. \tag{3.6}$$

Pada masukan terurut tidak ada inversi dan *insertion sort* melakukan $n - 1$ perbandingan tanpa pergeseran, seperti dicetak pada jalannya yang kedua. Pada masukan terbalik ia melakukan $n(n-1)/2$ dari keduanya. Pada masukan acak ia melakukan sekitar $n^2/4$ dari masing-masing, separuh kasus terburuk, menurut Persamaan (3.5).

Kerjanya sebanding dengan $n + \text{inv}(A)$. Itulah makna tepat dari "adaptif": biaya *insertion sort* mengukur seberapa tidak terurutnya masukan. Daftar seratus ribu nama yang ditambahi selusin nama baru di ujungnya menghabiskan beberapa ratus ribu langkah, bukan lima miliar. Tambahkan bahwa ia bersifat di tempat, stabil (penyusuran berhenti pada elemen yang *sama* dengan kunci, sehingga kunci yang sama tidak pernah saling melewati), kecil, dan cepat untuk n kecil karena gelung dalamnya nyaris tidak mengerjakan apa-apa, maka Anda akan memahami mengapa setiap pustaka pengurutan yang serius pada Subbab 3.14 menutup pekerjaannya dengan menjalankan *insertion sort* pada potongan-potongan kecil.

3.7 Shell Sort

3.7.1 Lompatan jauh lebih dahulu

Wawasan Kunci pada Subbab 3.3.5 menyatakan bahwa pengurutan yang memindahkan elemen hanya satu tempat setiap kali pasti berujung pada waktu kuadratik. Pada tahun 1959 Donald Shell menerbitkan jalan keluar yang tetap mempertahankan kesederhanaan *insertion sort* [3]. Alih-alih membandingkan setiap elemen dengan tetangganya, bandingkan dulu dengan elemen yang berjarak h tempat, untuk suatu **celah** (*gap*) h yang besar; artinya, jalankan *insertion sort* pada masing-masing dari h subdaftar yang berselang-seling $A[0], A[h], A[2h], \ldots$ dan $A[1], A[1+h], \ldots$ dan seterusnya. Elemen yang jauh dari rumahnya kini menempuh h tempat setiap kali bergeser. Kemudian ulangi dengan celah yang lebih kecil, dan tutup dengan celah 1, yaitu *insertion sort* biasa. Pada saat itu larik sudah hampir terurut, dan *insertion sort*, karena adaptif, selesai dengan cepat.

Pseudokode 3.5 *Shell sort*: *insertion sort* pada elemen yang berjarak h, untuk barisan celah menurun yang berakhir di 1.

```
SHELL-SORT(A, n, gaps)                    // celah menurun, terakhir = 1
 1  for each gap h in gaps
 2      for i = h to n - 1                 // insertion sort langkah h
 3          key = A[i]
 4          j = i - h
 5          while j ≥ 0 and A[j] > key
 6              A[j+h] = A[j]
 7              j = j - h
 8          A[j+h] = key
```

Dengan celah 1, baris 2 sampai 8 persis **Pseudokode 3.4**, sehingga *Shell sort* selalu benar, apa pun celah-celah lainnya; celah-celah sebelumnya hanya membuat lintasan terakhir lebih murah.

3.7.2 Jejak dengan celah 3 dan 1

Pilihan asli Shell adalah memulai dari $h = \lfloor n/2 \rfloor$ dan membaginya dua sampai $h = 1$. Untuk $n = 6$ itu menghasilkan celah 3 dan 1. **Gambar 3.7** memperlihatkan kedua lintasan; setiap warna adalah satu subdaftar untuk celah 3.



Gambar 3.7 *Shell sort* pada A dengan celah 3 dan 1. Mengurutkan tiga pasangan yang berselang-seling menghapus 9 dari 13 inversi sekaligus; *insertion sort* kemudian menghapus 4 sisanya.

Dengan celah 3 subdaftarnya adalah pasangan pada indeks (0,3), (1,4), dan (2,5), berisi (22,3), (10,8), dan (15,2). Setiap pasangan urutannya salah, dan setiap satu pergeseran memindahkan sebuah bilangan tiga tempat: 22 melompat dari indeks 0 ke indeks 3 dalam satu gerakan. Larik menjadi [3,8,2,22,10,15], yang hanya mempunyai 4 inversi, sehingga *insertion sort* penutup dengan celah 1 tidak banyak bekerja. Seluruh pengurutan melakukan 11 perbandingan dibandingkan 15 untuk *insertion sort* saja, penghematan kecil pada enam bilangan tetapi, seperti diperlihatkan program, penghematan luar biasa pada seratus ribu bilangan.

3.7.3 Barisan celah menentukan segalanya

Kecepatan *Shell sort* bergantung pada celahnya, dan menemukan celah yang baik ternyata merupakan masalah matematika yang sulit dan hingga kini belum sepenuhnya terpecahkan [1], [4]. Ada tiga barisan yang patut diketahui.

- **Barisan Shell**: $\lfloor n/2 \rfloor, \lfloor n/4 \rfloor, \ldots, 1$. Sederhana, tetapi bila n pangkat dua, posisi genap dan ganjil tidak pernah dibandingkan satu sama lain sampai lintasan terakhir, dan kasus terburuknya tetap $\Theta(n^2)$.
- **Barisan Knuth** [1]: 1,4,13,40,121, ..., yang didefinisikan oleh

$$h_1 = 1, \qquad h_{k+1} = 3h_k + 1, \tag{3.7}$$

dipakai mulai dari h_k terbesar di bawah $n/3$ ke bawah. Kasus terburuknya $O(n^{3/2})$, yang untuk sejuta elemen seribu kali lebih baik daripada n^2.

- **Barisan Ciura** [4]: 1,4,10,23,57,132,301,701, ditemukan pada tahun 2001 melalui eksperimen komputer, bukan melalui pembuktian, dan biasanya diperpanjang melewati 701 dengan mengalikan sekitar 2.25. Dalam praktik inilah barisan terbaik yang dikenal untuk larik sampai beberapa juta elemen.

Kode 3.16 mengimplementasikan **Pseudokode 3.5** dengan daftar celah apa pun, melacak contoh enam bilangan, lalu menghitung perbandingan yang diperlukan masing-masing dari ketiga barisan pada 100,000 bilangan acak yang sama. Bilangan acaknya berasal dari pembangkit kecil (*linear congruential generator*, pembangkit kongruensial linear) yang ditulis lengkap di dalam program, identik dalam ketiga bahasa, sehingga ketiga versi mengurutkan bilangan yang persis sama dan mencetak hitungan yang persis sama.

Kode 3.16 `shell_sort.cpp`: *Shell sort* dengan tiga barisan celah, dibandingkan pada 100.000 bilangan acak.

```cpp
// shell_sort.cpp - Shell sort: insertion sort dengan celah yang menyusut
#include <cstdint>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

uint64_t state = 2026;  // generator kecil, sama di ketiga bahasa
int nextRandom(int bound) {
    state = state * 6364136223846793005ULL + 1442695040888963407ULL;
    return static_cast<int>((state >> 33) % static_cast<uint64_t>(bound));
}

// Insertion sort bercelah untuk tiap celah; celah terakhir harus 1.
long long shellSort(vector<int>& a, const vector<int>& gaps, bool trace) {
    long long comparisons = 0;
    int n = static_cast<int>(a.size());
    for (int gap : gaps) {
        for (int i = gap; i < n; ++i) {
            int key = a[i];
            int j = i - gap;
            while (j >= 0) {
                ++comparisons;
                if (a[j] <= key) break;
                // geser sejauh celah, bukan 1
                a[j + gap] = a[j];
                j -= gap;
            }
            a[j + gap] = key;
        }
        if (trace) cout << "  after gap " << gap << ": " << show(a) << "\n";
    }
    return comparisons;
}
```

Kode 3.16 (Lanjutan) *shell_sort.cpp: Shell sort dengan tiga barisan celah, dibandingkan pada 100.000 bilangan acak.*

```
vector<int> shellGaps(int n) {  // n/2, n/4, ..., 1 (Shell, 1959)
    vector<int> g;
    for (int h = n / 2; h >= 1; h /= 2) g.push_back(h);
    return g;
}

vector<int> knuthGaps(int n) {  // ..., 40, 13, 4, 1 (Knuth)
    vector<int> g;
    int h = 1;
    while (h < n / 3) h = 3 * h + 1;
    for (; h >= 1; h /= 3) g.push_back(h);
    return g;
}

vector<int> ciuraGaps(int n) {  // 1, 4, 10, 23, ... (Ciura, 2001)
    vector<int> up = {1, 4, 10, 23, 57, 132, 301, 701, 1750};
    while (up.back() * 9 / 4 < n) up.push_back(up.back() * 9 / 4);
    vector<int> g;
    for (int i = static_cast<int>(up.size()) - 1; i >= 0; --i)
        if (up[i] < n) g.push_back(up[i]);
    return g;
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Shell sort on " << show(data) << " with gaps 3, 1\n";
    long long c = shellSort(data, shellGaps(6), true);
    cout << "  comparisons = " << c << "\n";

    int n = 100000;
    vector<int> base(n);
    for (int& x : base) x = nextRandom(1000000);
    cout << "Comparisons on " << n << " random numbers:\n";
    const char* names[] = {"Shell n/2", "Knuth 3h+1", "Ciura"};
    vector<int> gapLists[] = {shellGaps(n), knuthGaps(n), ciuraGaps(n)};
    for (int k = 0; k < 3; ++k) {
        vector<int> a = base;
        long long comps = shellSort(a, gapLists[k], false);
        cout << "  " << names[k] << ": " << gapLists[k].size() << " gaps, "
             << comps << " comparisons\n";
    }
    return 0;
}
```

Kode 3.17 `ShellSort.java`: *Shell sort* dalam Java. `>>>` adalah geseran tanpa tanda milik Java, yang diperlukan pembangkitnya.

```java
// ShellSort.java - Shell sort: insertion sort dengan celah yang menyusut
import java.util.ArrayList;
import java.util.List;

public class ShellSort {
    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static long state = 2026;  // generator kecil, sama di ketiga bahasa
    static int nextRandom(int bound) {
        state = state * 6364136223846793005L + 1442695040888963407L;
        return (int) ((state >>> 33) % bound);
    }

    // Insertion sort bercelah untuk tiap celah; celah terakhir harus 1.
    static long shellSort(int[] a, List<Integer> gaps, boolean trace) {
        long comparisons = 0;
        int n = a.length;
        for (int gap : gaps) {
            for (int i = gap; i < n; i++) {
                int key = a[i];
                int j = i - gap;
                while (j >= 0) {
                    comparisons++;
                    if (a[j] <= key) break;
                    // geser sejauh celah, bukan 1
                    a[j + gap] = a[j];
                    j -= gap;
                }
                a[j + gap] = key;
            }
            if (trace) System.out.println("  after gap " + gap + ": "
                    + show(a));
        }
        return comparisons;
    }
```

Kode 3.17 (Lanjutan) *ShellSort.java: Shell sort dalam Java. >>> adalah geseran tanpa tanda milik Java, yang diperlukan pembangkitnya.*

```
    static List<Integer> shellGaps(int n) {  // n/2, n/4, ..., 1 (Shell, 1959)
        List<Integer> g = new ArrayList<>();
        for (int h = n / 2; h >= 1; h /= 2) g.add(h);
        return g;
    }

    static List<Integer> knuthGaps(int n) {  // ..., 40, 13, 4, 1 (Knuth)
        List<Integer> g = new ArrayList<>();
        int h = 1;
        while (h < n / 3) h = 3 * h + 1;
        for (; h >= 1; h /= 3) g.add(h);
        return g;
    }

    static List<Integer> ciuraGaps(int n) {  // 1, 4, 10, 23, ... (Ciura, 2001)
        List<Integer> up =
                new ArrayList<>(List.of(1, 4, 10, 23, 57, 132, 301, 701, 1750));
        while (up.get(up.size() - 1) * 9 / 4 < n)
            up.add(up.get(up.size() - 1) * 9 / 4);
        List<Integer> g = new ArrayList<>();
        for (int i = up.size() - 1; i >= 0; i--)
            if (up.get(i) < n) g.add(up.get(i));
        return g;
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println("Shell sort on " + show(data) + " with gaps 3, 1");
        long c = shellSort(data, shellGaps(6), true);
        System.out.println("  comparisons = " + c);

        int n = 100000;
        int[] base = new int[n];
        for (int i = 0; i < n; i++) base[i] = nextRandom(1000000);
        System.out.println("Comparisons on " + n + " random numbers:");
        String[] names = {"Shell n/2", "Knuth 3h+1", "Ciura"};
        List<List<Integer>> gapLists =
                List.of(shellGaps(n), knuthGaps(n), ciuraGaps(n));
        for (int k = 0; k < 3; k++) {
            int[] a = base.clone();
            long comps = shellSort(a, gapLists.get(k), false);
            System.out.println("  " + names[k] + ": " + gapLists.get(k).size()
                    + " gaps, "
                    + comps + " comparisons");
        }
    }
```

Kode 3.17 (Lanjutan) *ShellSort.java: Shell sort dalam Java. >>> adalah geseran tanpa tanda milik Java, yang diperlukan pembangkitnya.*

```
}
```

Kode 3.18 `shell_sort.py`: *Shell sort* dalam Python. Bilangan bulat Python tidak pernah meluap, sehingga pembangkitnya mengambil sisa pembagian dengan 2 pangkat 64 secara manual.

```python
# shell_sort.py - Shell sort: insertion sort dengan celah yang menyusut
def show(a):
    return "[" + " ".join(map(str, a)) + "]"


state = 2026  # generator kecil, sama di ketiga bahasa


def next_random(bound):
    global state
    state = (state * 6364136223846793005 + 1442695040888963407) % 2**64
    return (state >> 33) % bound


def shell_sort(a, gaps, trace=False):
    """Insertion sort bercelah untuk tiap celah; celah terakhir harus 1."""
    comparisons = 0
    n = len(a)
    for gap in gaps:
        for i in range(gap, n):
            key = a[i]
            j = i - gap
            while j >= 0:
                comparisons += 1
                if a[j] <= key:
                    break
                a[j + gap] = a[j]        # geser sejauh celah, bukan 1
                j -= gap
            a[j + gap] = key
        if trace:
            print(f"  after gap {gap}: {show(a)}")
    return comparisons


def shell_gaps(n):                       # n/2, n/4, ..., 1 (Shell, 1959)
    g, h = [], n // 2
    while h >= 1:
        g.append(h)
        h //= 2
    return g
```

Kode 3.18 (Lanjutan) *shell_sort.py: Shell sort dalam Python. Bilangan bulat Python tidak pernah meluap, sehingga pembangkitnya mengambil sisa pembagian dengan 2 pangkat 64 secara manual.*

```
def knuth_gaps(n):                      # ..., 40, 13, 4, 1 (Knuth)
    h = 1
    while h < n // 3:
        h = 3 * h + 1
    g = []
    while h >= 1:
        g.append(h)
        h //= 3
    return g


def ciura_gaps(n):                      # 1, 4, 10, 23, ... (Ciura, 2001)
    up = [1, 4, 10, 23, 57, 132, 301, 701, 1750]
    while up[-1] * 9 // 4 < n:
        up.append(up[-1] * 9 // 4)
    return [h for h in reversed(up) if h < n]


data = [22, 10, 15, 3, 8, 2]
print(f"Shell sort on {show(data)} with gaps 3, 1")
c = shell_sort(data, shell_gaps(6), True)
print(f"  comparisons = {c}")

n = 100000
base = [next_random(1000000) for _ in range(n)]
print(f"Comparisons on {n} random numbers:")
for name, gaps in (("Shell n/2", shell_gaps(n)), ("Knuth 3h+1", knuth_gaps(n)),
                   ("Ciura", ciura_gaps(n))):
    comps = shell_sort(base[:], gaps)
    print(f"  {name}: {len(gaps)} gaps, {comps} comparisons")
```

Kode 3.17 dan **Kode 3.18** mencetak hal yang sama dengan **Kode 3.16**:

```
Shell sort on [22 10 15 3 8 2] with gaps 3, 1
  after gap 3: [3 8 2 22 10 15]
  after gap 1: [2 3 8 10 15 22]
  comparisons = 11
Comparisons on 100000 random numbers:
  Shell n/2: 16 gaps, 4488726 comparisons
  Knuth 3h+1: 11 gaps, 3842607 comparisons
  Ciura: 13 gaps, 2554426 comparisons
```

Insertion sort biasa pada 100,000 bilangan itu akan melakukan sekitar $n^2/4 \approx 2.5$ miliar perbandingan (Subbab 3.13 mengukur 2,500,332,969). Barisan Shell memerlukan 4.5 juta, barisan Knuth 3.8 juta, dan barisan Ciura 2.6 juta: kira-kira seribu kali lebih sedikit kerja, hasil dari perubahan delapan baris. *Shell sort* bersifat di tempat dan adaptif, tetapi **tidak stabil**, karena lompatan sejauh h tempat dapat melewati kunci yang sama.

3.8 Merge Sort

3.8.1 Bagi dan kuasai

Semua pengurutan sejauh ini bekerja pada seluruh larik sekaligus. **Merge sort** menempuh pendekatan sebaliknya, yang disebut **bagi dan kuasai** (*divide and conquer*):

1. **Bagi** larik menjadi dua paruh.
2. **Kuasai** setiap paruh dengan mengurutkannya, memakai metode yang sama, secara rekursif.

3. **Gabungkan** kedua paruh yang sudah terurut menjadi satu larik terurut dengan cara **menggabung** (*merge*).

Potongan berisi satu elemen sudah terurut, sehingga rekursi berhenti di sana. Seluruh kerja yang sesungguhnya terjadi pada langkah 3, dan menggabung itu mudah. Bayangkan dua tumpukan kartu indeks yang terurut, yang terkecil di atas. Bandingkan dua kartu teratas, ambil yang lebih kecil, letakkan telungkup pada tumpukan baru, dan ulangi. Setiap perbandingan menempatkan satu kartu, dan ketika satu tumpukan habis, sisa tumpukan lain tinggal dipindahkan. Dua *run* (deretan) terurut dengan panjang total m digabung dengan paling banyak $m - 1$ perbandingan.

Pseudokode 3.6 `MERGE` dan `MERGE-SORT`. Penyangga T sama panjang dengan A dan dialokasikan sekali saja, sebelum pemanggilan pertama.

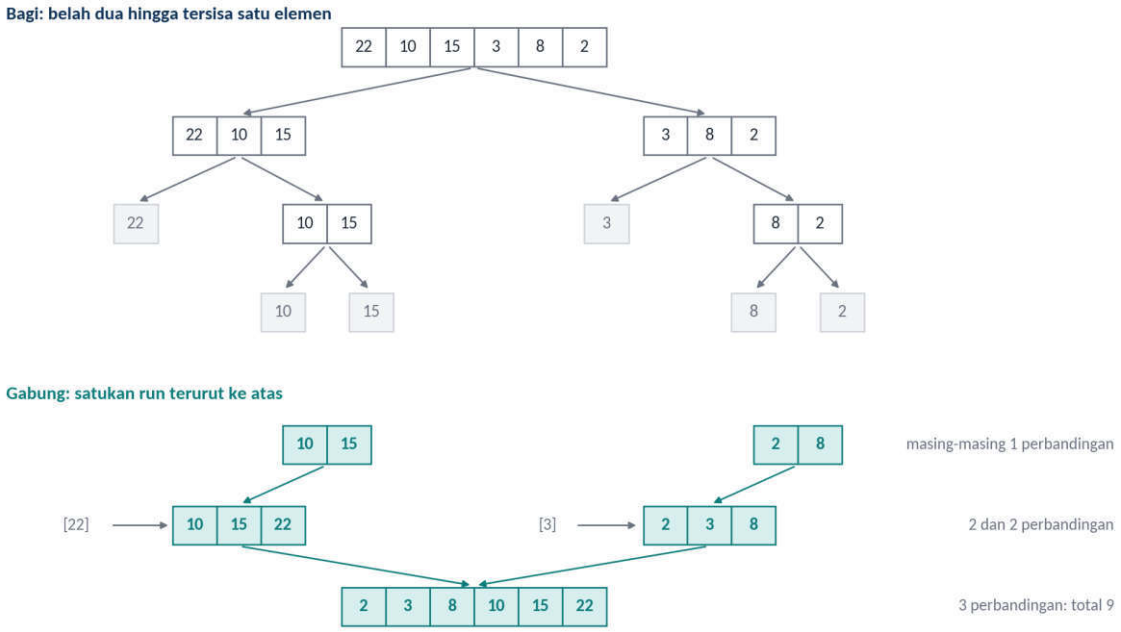
```
MERGE(A, T, lo, mid, hi)                    // kedua run terurut
 1  i = lo; j = mid; k = lo
 2  while i < mid and j < hi
 3      if A[i] ≤ A[j]                       // ≤ menjaga kestabilan
 4          T[k] = A[i]; i = i + 1
 5      else
 6          T[k] = A[j]; j = j + 1
 7      k = k + 1
 8  salin sisa A[i..mid-1], lalu sisa A[j..hi-1], ke T[k..hi-1]
 9  salin T[lo..hi-1] kembali ke A[lo..hi-1]

MERGE-SORT(A, T, lo, hi)                    // mengurutkan A[lo..hi-1]
 1  if hi - lo < 2
 2      return                               // 0 atau 1 elemen: terurut
 3  mid = lo + ⌊(hi - lo)/2⌋
 4  MERGE-SORT(A, T, lo, mid)
 5  MERGE-SORT(A, T, mid, hi)
 6  MERGE(A, T, lo, mid, hi)
```

Rentang ditulis sebagai selang setengah-terbuka: `lo..hi-1` berarti indeks $lo, lo + 1, \ldots, hi - 1$, sehingga rentang itu memuat $hi - lo$ elemen dan terbelah rapi di `mid`. Titik tengah dihitung sebagai `lo + (hi - lo)/2` alih-alih `(lo + hi)/2` karena jumlah `lo + hi` dapat meluapi `int` untuk larik berisi lebih dari satu miliar elemen, sebuah *bug* yang bertahun-tahun tidak disadari dalam kode pencarian biner dan *merge sort* yang dipakai luas.

3.8.2 Jejaknya

Gambar 3.8 memperlihatkan seluruh komputasi pada A. Paruh atas adalah pembagian, dari atas ke bawah; paruh bawah adalah penggabungan, dari bawah ke atas.



Gambar 3.8 *Merge sort* pada *A*. Pembagian (atas) membelah larik sampai setiap potongan berisi satu elemen; penggabungan (bawah) menyatukan *run* terurut, dan perbandingan pada setiap tingkat dicatat di sebelah kanan.

[22,10,15,3,8,2] terbelah menjadi [22,10,15] dan [3,8,2]; yang pertama terbelah menjadi [22] dan [10,15], dan [10,15] menjadi [10] dan [15]. Menggabung [10] dengan [15] memerlukan satu perbandingan dan menghasilkan [10,15]. Menggabung [22] dengan [10,15]: $22 > 10$ mengambil 10, $22 > 15$ mengambil 15, *run* kanan habis, dan 22 disalin: dua perbandingan, [10,15,22]. Paruh kanan mengerjakan hal yang sama dan menghasilkan [2,3,8]. Penggabungan terakhir membandingkan 10 dengan 2, 10 dengan 3, dan 10 dengan 8, mengambil 2, 3, dan 8; *run* kanan kemudian habis dan 10,15,22 disalin tanpa perbandingan lagi. Seluruhnya $1 + 2 + 1 + 2 + 3 = 9$ perbandingan.

3.8.3 Programnya

Kode 3.19 mencetak setiap penggabungan, menjorok menurut kedalamannya dalam rekursi, sehingga cetakannya berbentuk seperti paruh bawah **Gambar 3.8**. *String* `left` dan `right` ditangkap *sebelum* `merge` menimpa rentang itu, agar baris cetakan dapat memperlihatkan apa yang digabung.

Kode 3.19 `merge_sort.cpp`: *merge sort* atas-bawah dengan satu penyangga bersama, mencetak setiap penggabungan.

```cpp
// merge_sort.cpp - merge sort atas-bawah, mencetak setiap penggabungan
#include <iostream>
#include <string>
#include <vector>
using namespace std;

string show(const vector<int>& a, int lo, int hi) {  // irisan a[lo..hi-1]
    string s = "[";
    for (int i = lo; i < hi; ++i) s += (i > lo ? " " : "") + to_string(a[i]);
    return s + "]";
}

long long comparisons = 0, moves = 0;

// Menggabungkan run terurut a[lo..mid-1] dan a[mid..hi-1] melalui penyangga
// tmp.
void merge(vector<int>& a, vector<int>& tmp, int lo, int mid, int hi) {
    int i = lo, j = mid, k = lo;
    while (i < mid && j < hi) {
        ++comparisons;
        // <= mengambil yang kiri bila sama: stabil
        if (a[i] <= a[j]) tmp[k++] = a[i++];
        else              tmp[k++] = a[j++];
    }
    while (i < mid) tmp[k++] = a[i++];
    while (j < hi)  tmp[k++] = a[j++];
    for (k = lo; k < hi; ++k) { a[k] = tmp[k]; ++moves; }
}

void mergeSort(vector<int>& a, vector<int>& tmp, int lo, int hi, int depth) {
    if (hi - lo < 2) return;  // 0 atau 1 elemen: sudah terurut
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, tmp, lo, mid, depth + 1);
    mergeSort(a, tmp, mid, hi, depth + 1);
    string left = show(a, lo, mid), right = show(a, mid, hi);
    merge(a, tmp, lo, mid, hi);
    cout << string(2 * depth + 2, ' ') << "merge " << left << " + " << right
         << " -> " << show(a, lo, hi) << "\n";
}
```

Kode 3.19 (Lanjutan) *merge_sort.cpp: merge sort atas-bawah dengan satu penyangga bersama, mencetak setiap penggabungan.*

```cpp
int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    vector<int> tmp(data.size());  // satu penyangga, dialokasikan sekali
    cout << "Merge sort on " << show(data, 0, 6) << "\n";
    mergeSort(data, tmp, 0, static_cast<int>(data.size()), 0);
    cout << "  result " << show(data, 0, 6) << ", comparisons = " << comparisons
         << ", moves = " << moves << "\n";
    return 0;
}
```

Kode 3.20 `MergeSort.java`: *merge sort* dalam Java.

```java
// MergeSort.java - merge sort atas-bawah, mencetak setiap penggabungan
public class MergeSort {
    static String show(int[] a, int lo, int hi) {    // irisan a[lo..hi-1]
        StringBuilder sb = new StringBuilder("[");
        for (int i = lo; i < hi; i++)
            sb.append(i > lo ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static long comparisons = 0, moves = 0;

    // Gabungkan run terurut a[lo..mid-1] dan a[mid..hi-1] lewat penyangga tmp.
    static void merge(int[] a, int[] tmp, int lo, int mid, int hi) {
        int i = lo, j = mid, k = lo;
        while (i < mid && j < hi) {
            comparisons++;
            // <= mengambil yang kiri bila sama: stabil
            if (a[i] <= a[j]) tmp[k++] = a[i++];
            else              tmp[k++] = a[j++];
        }
        while (i < mid) tmp[k++] = a[i++];
        while (j < hi)  tmp[k++] = a[j++];
        for (k = lo; k < hi; k++) { a[k] = tmp[k]; moves++; }
    }

    static void mergeSort(int[] a, int[] tmp, int lo, int hi, int depth) {
        if (hi - lo < 2) return;  // 0 atau 1 elemen: sudah terurut
        int mid = lo + (hi - lo) / 2;
        mergeSort(a, tmp, lo, mid, depth + 1);
        mergeSort(a, tmp, mid, hi, depth + 1);
        String left = show(a, lo, mid), right = show(a, mid, hi);
        merge(a, tmp, lo, mid, hi);
        System.out.println(" ".repeat(2 * depth + 2) + "merge " + left + " + "
                + right
                + " -> " + show(a, lo, hi));
    }
```

Kode 3.20 (Lanjutan) *MergeSort.java: merge sort dalam Java.*

```java
    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        // satu penyangga, dialokasikan sekali
        int[] tmp = new int[data.length];
        System.out.println("Merge sort on " + show(data, 0, 6));
        mergeSort(data, tmp, 0, data.length, 0);
        System.out.println("  result " + show(data, 0, 6) + ", comparisons = "
                + comparisons
                + ", moves = " + moves);
    }
}
```

Kode 3.21 `merge_sort.py`: *merge sort* dalam Python.

```
# merge_sort.py - merge sort atas-bawah, mencetak setiap penggabungan
def show(a, lo=0, hi=None):              # irisan a[lo..hi-1]
    return "[" + " ".join(map(str, a[lo:hi])) + "]"


comparisons = moves = 0


def merge(a, tmp, lo, mid, hi):
    """Gabungkan run terurut a[lo..mid-1] dan a[mid..hi-1] lewat penyangga."""
    global comparisons, moves
    i, j, k = lo, mid, lo
    while i < mid and j < hi:
        comparisons += 1
        # <= mengambil yang kiri bila sama: stabil
        if a[i] <= a[j]:
            tmp[k] = a[i]
            i += 1
        else:
            tmp[k] = a[j]
            j += 1
        k += 1
    while i < mid:
        tmp[k] = a[i]
        i += 1
        k += 1
    while j < hi:
        tmp[k] = a[j]
        j += 1
        k += 1
    for k in range(lo, hi):
        a[k] = tmp[k]
        moves += 1


def merge_sort(a, tmp, lo, hi, depth=0):
    if hi - lo < 2:                      # 0 atau 1 elemen: sudah terurut
        return
    mid = lo + (hi - lo) // 2
    merge_sort(a, tmp, lo, mid, depth + 1)
    merge_sort(a, tmp, mid, hi, depth + 1)
    left, right = show(a, lo, mid), show(a, mid, hi)
    merge(a, tmp, lo, mid, hi)
    indent = " " * (2 * depth + 2)
    print(f"{indent}merge {left} + {right} -> {show(a, lo, hi)}")
```

Kode 3.21 (Lanjutan) *merge_sort.py: merge sort dalam Python.*

```

data = [22, 10, 15, 3, 8, 2]
tmp = [0] * len(data)                    # satu penyangga, dialokasikan sekali
print(f"Merge sort on {show(data)}")
merge_sort(data, tmp, 0, len(data))
print(f"  result {show(data)}, comparisons = {comparisons}, moves = {moves}")
```

Kode 3.19, **Kode 3.20**, dan **Kode 3.21** semuanya mencetak:

```
Merge sort on [22 10 15 3 8 2]
      merge [10] + [15] -> [10 15]
    merge [22] + [10 15] -> [10 15 22]
      merge [8] + [2] -> [2 8]
    merge [3] + [2 8] -> [2 3 8]
  merge [10 15 22] + [2 3 8] -> [2 3 8 10 15 22]
  result [2 3 8 10 15 22], comparisons = 9, moves = 16
```

3.8.4 Mengapa merge sort selalu $n\lg n$

Misalkan $T(n)$ adalah banyak langkah yang dilakukan *merge sort* pada n elemen. Ia melakukan dua pemanggilan rekursif pada paruh-paruhnya lalu satu penggabungan yang menyentuh setiap elemen dari n elemen itu sebanyak konstanta kali, sehingga untuk suatu konstanta c,

$$T(n) = 2\,T(n/2) + c\,n, \qquad T(1) = c. \tag{3.8}$$

Cara termudah menyelesaikan Persamaan (3.8) adalah menggambarnya sebagai pohon. Pemanggilan teratas berbiaya cn untuk penggabungannya. Ia mempunyai dua anak, masing-masing menggabung $n/2$ elemen dengan biaya $cn/2$, bersama-sama cn. Keduanya mempunyai empat anak berbiaya $cn/4$ masing-masing, bersama-sama lagi cn. Setiap tingkat pohon berbiaya cn, karena banyaknya potongan berlipat dua tepat ketika ukurannya terbelah dua. Ukurannya berjalan $n, n/2, n/4, \ldots, 1$, jadi ada $\lg n$ kali pembelahan dan $\lg n + 1$ tingkat. Karena itu

$$T(n) = c\,n\,(\lg n + 1) = \Theta(n \lg n). \tag{3.9}$$

Tidak ada bagian argumen itu yang bergantung pada data. *Merge sort* membelah di tengah entah masukannya terurut, terbalik, atau acak, sehingga ia $\Theta(n \lg n)$ pada kasus terbaik, rata-rata, *dan* terburuk. Bila perbandingan dihitung persis, kasus terburuknya $n\lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1$ [1]; untuk $n = 6$ itu $18 - 8 + 1 = 11$, dan larik kita hanya memerlukan 9.

Merge sort membayar jaminannya dengan memori: penyangga T memuat n elemen, sehingga ia **tidak di tempat**, memerlukan ruang tambahan $\Theta(n)$. Ia **stabil**, karena `MERGE` mengambil dari *run* kiri bila kuncinya sama (baris 3 memakai `≤`, bukan `<`), sehingga kunci yang sama mempertahankan urutan aslinya. Dalam bentuk sederhana di atas ia tidak adaptif, tetapi pengurutan pustaka pada Subbab 3.14, Timsort dan Powersort, adalah *merge sort* yang dibuat adaptif dengan menemukan *run* terurut yang sudah ada dalam data.

3.9 Quicksort

3.9.1 Partisi, lalu rekursi

Merge sort bekerja remeh saat membagi dan bekerja sungguhan saat menggabung. **Quicksort**, yang ditemukan C. A. R. Hoare pada tahun 1959 dan diterbitkan pada tahun 1962 [5], melakukan kebalikannya: kerja sungguhan saat membagi, dan sama sekali tanpa kerja saat menggabung. Pilih satu elemen, sang **pivot**. **Partisi** larik sehingga semua yang tidak lebih besar daripada pivot berada sebelum pivot dan semua yang lebih besar berada sesudahnya. Pivot kini berada di posisi akhirnya, kedua sisi dapat diurutkan secara terpisah melalui rekursi, dan ketika keduanya terurut seluruh larik terurut, tanpa penggabungan.

Pseudokode 3.7 `QUICKSORT` dengan partisi yang mengembalikan indeks akhir pivot.

```
QUICKSORT(A, lo, hi)                          // mengurutkan A[lo..hi], inklusif
 1  if lo < hi
 2      p = PARTITION(A, lo, hi)              // pivot kini di indeks p
 3      QUICKSORT(A, lo, p - 1)
 4      QUICKSORT(A, p + 1, hi)
```

Segalanya bergantung pada `PARTITION`, dan ada dua cara klasik menuliskannya.

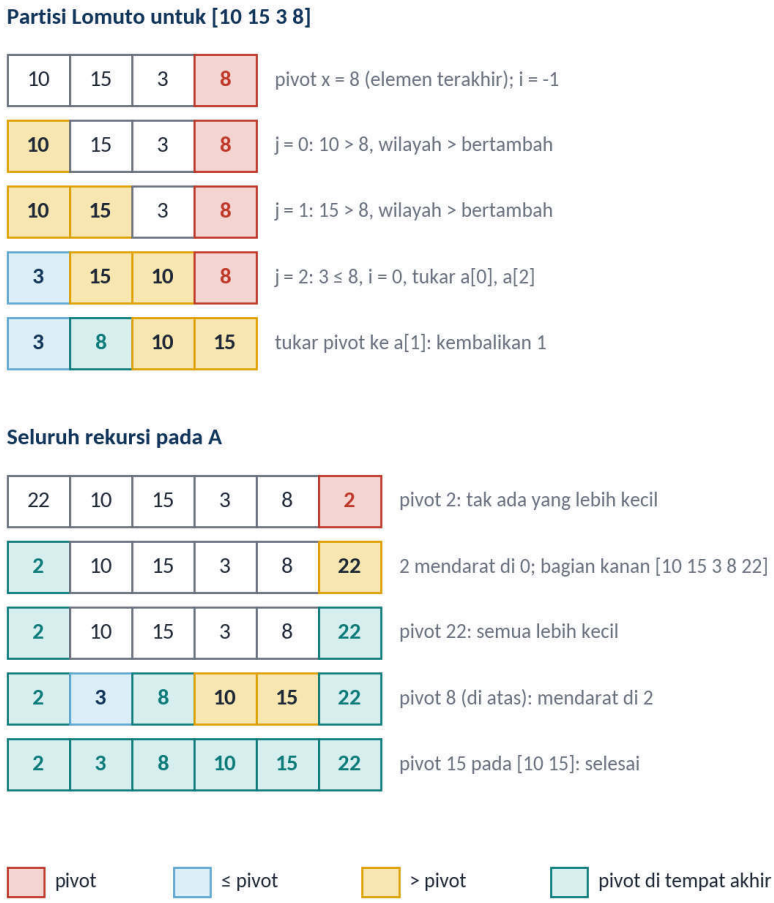
3.9.2 Partisi Lomuto

Skema Lomuto, yang dipopulerkan Bentley dan dipakai dalam CLRS [6], mengambil elemen terakhir sebagai pivot x dan menyusuri rentang sekali dari kiri ke kanan dengan indeks j. Ia menjaga sebuah batas i: posisi $lo..i$ memuat elemen $\leq x$, posisi $i+1..j-1$ memuat elemen $>x$, dan $j..hi-1$ belum diperiksa. Bila `A[j]` $\leq x$, elemen itu termasuk wilayah kiri, sehingga i maju satu langkah dan `A[j]` ditukar ke posisi i. Di akhir, pivot ditukar ke posisi $i+1$, di antara kedua wilayah.

Pseudokode 3.8 Partisi Lomuto: satu penyusuran kiri-ke-kanan yang menumbuhkan wilayah elemen yang tidak lebih besar daripada pivot.

```
LOMUTO-PARTITION(A, lo, hi)
 1  x = A[hi]                              // sang pivot
 2  i = lo - 1                             // A[lo..i] ≤ x
 3  for j = lo to hi - 1
 4      if A[j] ≤ x
 5          i = i + 1
 6          exchange A[i] with A[j]
 7  exchange A[i+1] with A[hi]             // pivot di antara wilayah
 8  return i + 1
```

Gambar 3.9 melacaknya. Panel atas adalah partisi [10,15,3,8], yang paling menarik dari empat partisi yang mengurutkan A; panel bawah adalah seluruh rekursinya.



Gambar 3.9 Partisi Lomuto. Atas: mempartisi [10,15,3,8] di sekitar pivot 8. Bawah: empat partisi yang mengurutkan A dengan elemen terakhir sebagai pivot.

Pada panel atas, pivotnya 8 dan $i=-1$. $10>8$ dan $15>8$ hanya memperluas wilayah "lebih besar". $3\leq 8$: i menjadi 0 dan 3 ditukar dengan 10, elemen pertama wilayah "lebih besar", yang

pindah ke ujung wilayah itu. Akhirnya pivot ditukar dengan `A[i+1] = 15`, menghasilkan [3,8,10,15] dengan 8 di indeks 1 dari sublarik itu. Pada panel bawah terlihat kelemahan memilih elemen terakhir sebagai pivot: pivot pertama, 2, adalah elemen terkecil, sehingga partisinya tidak menghasilkan apa pun di kiri dan lima elemen di kanan; pivot kedua, 22, adalah yang terbesar dari kelima elemen itu, dan lagi-lagi satu sisi kosong. Hanya pivot ketiga, 8, yang membelah secara seimbang.

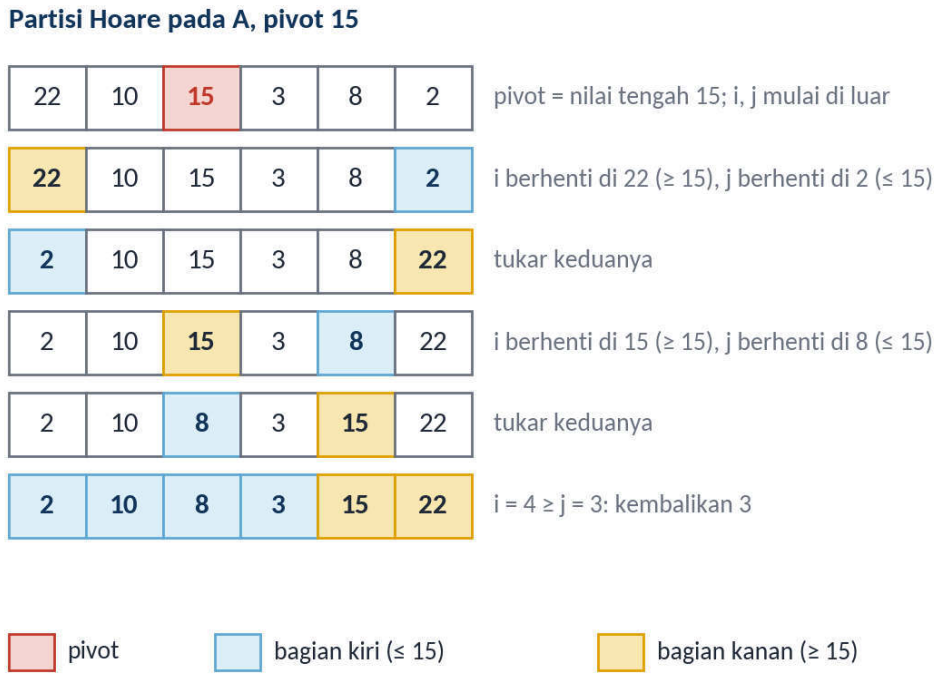
3.9.3 Partisi Hoare

Skema asli Hoare menjalankan dua indeks yang saling mendekat. i bergerak ke kanan dari ujung kiri dan berhenti pada elemen $\geq x$; j bergerak ke kiri dari ujung kanan dan berhenti pada elemen $\leq x$. Kedua elemen tempat mereka berhenti berada di sisi yang salah, sehingga keduanya ditukar, dan penyusuran berlanjut. Ketika kedua indeks bersilangan, semua yang berada di j atau di kirinya $\leq x$ dan semua yang berada di kanan $j \geq x$. Versi kita mengambil elemen tengah sebagai pivot.

Pseudokode 3.9 Partisi Hoare: dua indeks bergerak saling mendekat dan menukar pasangan yang berada di sisi yang salah.

```
HOARE-PARTITION(A, lo, hi)
 1  x = A[lo + ⌊(hi − lo)/2⌋]              // nilai pivot
 2  i = lo − 1; j = hi + 1
 3  loop
 4      repeat i = i + 1 until A[i] ≥ x
 5      repeat j = j − 1 until A[j] ≤ x
 6      if i ≥ j
 7          return j                        // A[lo..j] ≤ x ≤ A[j+1..hi]
 8      exchange A[i] with A[j]
```

Dengan partisi Hoare, pivot *tidak* selalu berakhir di indeks j, sehingga pemanggilan rekursifnya adalah `QUICKSORT(A, lo, j)` dan `QUICKSORT(A, j+1, hi)`, bukan pemanggilan pada **Pseudokode 3.7**. **Gambar 3.10** mempartisi seluruh A di sekitar nilai tengah 15.



Gambar 3.10 Partisi Hoare pada A dengan pivot 15. Dua pertukaran sudah cukup; kedua indeks bersilangan dengan $j = 3$, membelah A menjadi [2,10,8,3] dan [15,22].

i langsung berhenti pada $22 \geq 15$ dan j langsung berhenti pada $2 \leq 15$; keduanya ditukar, menghasilkan [2,10,15,3,8,22]. Kemudian i melewati 10 dan berhenti di indeks 2, pada 15 itu sendiri (ujinya $\geq$, sehingga elemen yang sama dengan pivot menghentikannya), dan j melangkah ke indeks 4 lalu langsung berhenti pada 8; ditukar, menghasilkan [2,10,8,3,15,22]. Pada putaran ketiga i melewati 3 dan berhenti di indeks 4 pada 15, sedangkan j berhenti di indeks 3 pada 3. Kini $i = 4 \geq j = 3$: kedua indeks sudah bersilangan dan partisi mengembalikan 3. Skema Hoare rata-rata melakukan lebih sedikit pertukaran daripada skema Lomuto, kira-kira sepertiganya [1], dan ia membelah larik berisi kunci-kunci yang sama secara seimbang, sementara skema Lomuto merosot; itulah sebabnya implementasi pustaka diturunkan darinya.

3.9.4 Programnya, dan pemilihan pivot

Kode 3.22 memuat kedua partisi. Versi Lomuto menerima sebuah **aturan pivot**: 0 memakai elemen terakhir, 1 memakai **median dari tiga** (*median of three*, nilai tengah dari elemen pertama, tengah, dan terakhir, dipindah ke ujung sebelum dipartisi), dan 2 memakai elemen **acak**. Setelah melacak kedua skema pada A, program mengurutkan 3000 elemen yang sudah terurut dan 3000 elemen acak dengan setiap aturan, sambil menghitung perbandingan dan kedalaman maksimum rekursi.

Kode 3.22 `quick_sort.cpp`: *quicksort* dengan partisi Lomuto dan Hoare, serta eksperimen dengan tiga aturan pivot.

```cpp
// quick_sort.cpp - quicksort: partisi Lomuto dan Hoare, tiga aturan pivot
#include <algorithm>
#include <cstdint>
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

// irisan a[lo..hi], inklusif
string show(const vector<int>& a, int lo, int hi) {
    string s = "[";
    for (int i = lo; i <= hi; ++i) s += (i > lo ? " " : "") + to_string(a[i]);
    return s + "]";
}

uint64_t state = 2026;
int nextRandom(int bound) {
    state = state * 6364136223846793005ULL + 1442695040888963407ULL;
    return static_cast<int>((state >> 33) % static_cast<uint64_t>(bound));
}

long long comparisons = 0, swaps = 0;
int maxDepth = 0;
bool trace = false;

// Lomuto: pivot = a[hi]; a[lo..i] memuat elemen <= pivot yang sudah dilihat.
int lomutoPartition(vector<int>& a, int lo, int hi) {
    int pivot = a[hi];
    int i = lo - 1;
    for (int j = lo; j < hi; ++j) {
        ++comparisons;
        if (a[j] <= pivot) {
            ++i;
            if (i != j) { swap(a[i], a[j]); ++swaps; }
        }
    }
    if (i + 1 != hi) { swap(a[i + 1], a[hi]); ++swaps; }
    return i + 1;                                    // indeks akhir pivot
}
```

Kode 3.22 (Lanjutan) *quick_sort.cpp: quicksort dengan partisi Lomuto dan Hoare, serta eksperimen dengan tiga aturan pivot.*

```
// Hoare: pivot = nilai tengah; dua indeks bergerak ke dalam dan menukar.
int hoarePartition(vector<int>& a, int lo, int hi) {
    int pivot = a[lo + (hi - lo) / 2];
    int i = lo - 1, j = hi + 1;
    while (true) {
        do { ++i; ++comparisons; } while (a[i] < pivot);
        do { --j; ++comparisons; } while (a[j] > pivot);
        if (i >= j) return j;  // a[lo..j] <= pivot <= a[j+1..hi]
        swap(a[i], a[j]);
        ++swaps;
    }
}

void quickLomuto(vector<int>& a, int lo, int hi, int rule, int depth) {
    if (lo >= hi) return;
    maxDepth = max(maxDepth, depth);
    if (rule == 1) {  // median dari tiga: a[lo], a[mid], a[hi]
        int mid = lo + (hi - lo) / 2;
        if (a[mid] < a[lo]) swap(a[mid], a[lo]);
        if (a[hi] < a[lo]) swap(a[hi], a[lo]);
        if (a[mid] < a[hi]) swap(a[mid], a[hi]);  // kini a[hi] memuat mediannya
    } else if (rule == 2) {                          // pivot acak
        swap(a[lo + nextRandom(hi - lo + 1)], a[hi]);
    }
    string before = trace ? show(a, lo, hi) : "";
    int p = lomutoPartition(a, lo, hi);
    if (trace)
        cout << string(2 * depth + 2, ' ') << "partition " << before
             << " pivot " << a[p] << " -> " << show(a, lo, hi)
             << ", pivot at " << p << "\n";
    quickLomuto(a, lo, p - 1, rule, depth + 1);
    quickLomuto(a, p + 1, hi, rule, depth + 1);
}

void quickHoare(vector<int>& a, int lo, int hi, int depth) {
    if (lo >= hi) return;
    string before = show(a, lo, hi);
    int pivot = a[lo + (hi - lo) / 2];
    int p = hoarePartition(a, lo, hi);
    cout << string(2 * depth + 2, ' ') << "partition " << before
         << " pivot " << pivot << " -> " << show(a, lo, p) << " "
         << show(a, p + 1, hi) << "\n";
    quickHoare(a, lo, p, depth + 1);
    quickHoare(a, p + 1, hi, depth + 1);
}
```

Kode 3.22 (Lanjutan) *quick_sort.cpp: quicksort dengan partisi Lomuto dan Hoare, serta eksperimen dengan tiga aturan pivot.*

```
int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Quicksort, Lomuto partition, last element as pivot:\n";
    trace = true;
    quickLomuto(data, 0, 5, 0, 0);
    cout << "  result " << show(data, 0, 5) << ", comparisons = " << comparisons
         << ", swaps = " << swaps << "\n";

    vector<int> data2 = {22, 10, 15, 3, 8, 2};
    comparisons = swaps = 0;
    cout << "Quicksort, Hoare partition, middle element as pivot:\n";
    quickHoare(data2, 0, 5, 0);
    cout << "  result " << show(data2, 0, 5) << ", comparisons = "
         << comparisons << ", swaps = " << swaps << "\n";

    trace = false;
    const int n = 3000;
    const char* rules[] = {"last element", "median of three", "random"};
    const char* inputs[] = {"sorted", "random"};
    cout << "Pivot rules on " << n << " elements (Lomuto):\n";
    for (int in = 0; in < 2; ++in) {
        for (int rule = 0; rule < 3; ++rule) {
            vector<int> a(n);
            state = 2026;
            for (int k = 0; k < n; ++k)
                a[k] = (in == 0) ? k : nextRandom(1000000);
            comparisons = swaps = 0;
            maxDepth = 0;
            quickLomuto(a, 0, n - 1, rule, 0);
            cout << "  " << inputs[in] << " input, " << rules[rule] << ": "
                 << comparisons << " comparisons, depth " << maxDepth << "\n";
        }
    }
    return 0;
}
```

Versi Java memerlukan satu langkah pengaman tambahan, dan ini mendidik. Mengurutkan 3000 elemen terurut dengan elemen terakhir sebagai pivot berekursi hampir 3000 tingkat, dan setiap tingkat adalah pemanggilan metode Java dengan *frame*-nya sendiri pada tumpukan panggilan (*call stack*); pada *runtime* Java yang dipakai di sini, itu melampaui *stack* bawaan *thread* utama dan program berhenti dengan `StackOverflowError`. Karena itu **Kode 3.23** menjalankan eksperimennya pada `Thread` baru yang dibuat dengan *stack* 64 MB.

Kode 3.23 `QuickSort.java`: *quicksort* dalam Java; eksperimen pivot berjalan pada *thread* dengan *stack* yang lebih besar.

```java
// QuickSort.java - quicksort: partisi Lomuto dan Hoare, tiga aturan pivot
public class QuickSort {
    // irisan a[lo..hi], inklusif
    static String show(int[] a, int lo, int hi) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = lo; i <= hi; i++)
            sb.append(i > lo ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static long state = 2026;
    static int nextRandom(int bound) {
        state = state * 6364136223846793005L + 1442695040888963407L;
        return (int) ((state >>> 33) % bound);
    }

    static long comparisons = 0, swaps = 0;
    static int maxDepth = 0;
    static boolean trace = false;

    static void swap(int[] a, int i, int j) {
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }

    // Lomuto: pivot = a[hi]; a[lo..i] memuat elemen <= pivot yang sudah
    // dilihat.
    static int lomutoPartition(int[] a, int lo, int hi) {
        int pivot = a[hi];
        int i = lo - 1;
        for (int j = lo; j < hi; j++) {
            comparisons++;
            if (a[j] <= pivot) {
                i++;
                if (i != j) { swap(a, i, j); swaps++; }
            }
        }
        if (i + 1 != hi) { swap(a, i + 1, hi); swaps++; }
        return i + 1;                                  // indeks akhir pivot
    }
```

Kode 3.23 (Lanjutan) *QuickSort.java: quicksort dalam Java; eksperimen pivot berjalan pada thread dengan stack yang lebih besar.*

```
    // Hoare: pivot = nilai tengah; dua indeks bergerak ke dalam dan menukar.
    static int hoarePartition(int[] a, int lo, int hi) {
        int pivot = a[lo + (hi - lo) / 2];
        int i = lo - 1, j = hi + 1;
        while (true) {
            do { i++; comparisons++; } while (a[i] < pivot);
            do { j--; comparisons++; } while (a[j] > pivot);
            if (i >= j) return j;  // a[lo..j] <= pivot <= a[j+1..hi]
            swap(a, i, j);
            swaps++;
        }
    }

    static void quickLomuto(int[] a, int lo, int hi, int rule, int depth) {
        if (lo >= hi) return;
        maxDepth = Math.max(maxDepth, depth);
        if (rule == 1) {  // median dari tiga: a[lo], a[mid], a[hi]
            int mid = lo + (hi - lo) / 2;
            if (a[mid] < a[lo]) swap(a, mid, lo);
            if (a[hi] < a[lo]) swap(a, hi, lo);
            // kini a[hi] memuat mediannya
            if (a[mid] < a[hi]) swap(a, mid, hi);
        } else if (rule == 2) {                      // pivot acak
            swap(a, lo + nextRandom(hi - lo + 1), hi);
        }
        String before = trace ? show(a, lo, hi) : "";
        int p = lomutoPartition(a, lo, hi);
        if (trace)
            System.out.println(" ".repeat(2 * depth + 2) + "partition "
                    + before + " pivot "
                    + a[p] + " -> " + show(a, lo, hi) + ", pivot at " + p);
        quickLomuto(a, lo, p - 1, rule, depth + 1);
        quickLomuto(a, p + 1, hi, rule, depth + 1);
    }

    static void quickHoare(int[] a, int lo, int hi, int depth) {
        if (lo >= hi) return;
        String before = show(a, lo, hi);
        int pivot = a[lo + (hi - lo) / 2];
        int p = hoarePartition(a, lo, hi);
        System.out.println(" ".repeat(2 * depth + 2) + "partition " + before
                + " pivot " + pivot
                + " -> " + show(a, lo, p) + " " + show(a, p + 1, hi));
        quickHoare(a, lo, p, depth + 1);
        quickHoare(a, p + 1, hi, depth + 1);
    }
```

Kode 3.23 (Lanjutan) *QuickSort.java: quicksort dalam Java; eksperimen pivot berjalan pada thread dengan stack yang lebih besar.*

```
    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println(
                "Quicksort, Lomuto partition, last element as pivot:");
        trace = true;
        quickLomuto(data, 0, 5, 0, 0);
        System.out.println("  result " + show(data, 0, 5) + ", comparisons = "
                + comparisons
                + ", swaps = " + swaps);

        int[] data2 = {22, 10, 15, 3, 8, 2};
        comparisons = swaps = 0;
        System.out.println(
                "Quicksort, Hoare partition, middle element as pivot:");
        quickHoare(data2, 0, 5, 0);
        System.out.println("  result " + show(data2, 0, 5)
                + ", comparisons = " + comparisons
                + ", swaps = " + swaps);

        trace = false;
        // Masukan terurut dengan aturan elemen terakhir berekursi sekitar 3000
        // tingkat, melebihi daya tampung stack thread bawaan, jadi bagian ini
        // berjalan di thread dengan stack 64 MB.
        long stackBytes = 64L << 20;
        Thread big = new Thread(null, QuickSort::pivotExperiment, "pivots",
                stackBytes);
        big.start();
        try {
            big.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    static void pivotExperiment() {
        final int n = 3000;
        String[] rules = {"last element", "median of three", "random"};
        String[] inputs = {"sorted", "random"};
        System.out.println("Pivot rules on " + n + " elements (Lomuto):");
        for (int in = 0; in < 2; in++) {
            for (int rule = 0; rule < 3; rule++) {
                int[] a = new int[n];
                state = 2026;
                for (int k = 0; k < n; k++)
                    a[k] = (in == 0) ? k : nextRandom(1000000);
```

Kode 3.23 (Lanjutan) *QuickSort.java: quicksort dalam Java; eksperimen pivot berjalan pada thread dengan stack yang lebih besar.*

```
                comparisons = swaps = 0;
                maxDepth = 0;
                quickLomuto(a, 0, n - 1, rule, 0);
                System.out.println("  " + inputs[in] + " input, "
                        + rules[rule] + ": "
                        + comparisons + " comparisons, depth " + maxDepth);
            }
        }
    }
}
```

Python membatasi kedalaman rekursi sampai 1000 pemanggilan secara bawaan, sehingga **Kode 3.24** menaikkan batas itu dengan `sys.setrecursionlimit` sebelum eksperimen. Python

tidak mempunyai gelung `do … while`, sehingga setiap `repeat … until` pada **Pseudokode 3.9** menjadi satu langkah awal yang diikuti `while`.

Kode 3.24 `quick_sort.py`: *quicksort* dalam Python.

```
# quick_sort.py - quicksort: partisi Lomuto dan Hoare, tiga aturan pivot
import sys


def show(a, lo, hi):                    # irisan a[lo..hi], inklusif
    return "[" + " ".join(map(str, a[lo:hi + 1])) + "]"


state = 2026


def next_random(bound):
    global state
    state = (state * 6364136223846793005 + 1442695040888963407) % 2**64
    return (state >> 33) % bound


comparisons = swaps = max_depth = 0
trace = False


def lomuto_partition(a, lo, hi):
    """Pivot = a[hi]; a[lo..i] memuat elemen <= pivot yang sudah dilihat."""
    global comparisons, swaps
    pivot = a[hi]
    i = lo - 1
    for j in range(lo, hi):
        comparisons += 1
        if a[j] <= pivot:
            i += 1
            if i != j:
                a[i], a[j] = a[j], a[i]
                swaps += 1
    if i + 1 != hi:
        a[i + 1], a[hi] = a[hi], a[i + 1]
        swaps += 1
    return i + 1                        # indeks akhir pivot
```

Kode 3.24 (Lanjutan) *quick_sort.py: quicksort dalam Python.*

```
def hoare_partition(a, lo, hi):
    """Pivot = nilai tengah; dua indeks mendekat, menukar yang salah."""
    global comparisons, swaps
    pivot = a[lo + (hi - lo) // 2]
    i, j = lo - 1, hi + 1
    while True:
        i += 1
        comparisons += 1
        while a[i] < pivot:
            i += 1
            comparisons += 1
        j -= 1
        comparisons += 1
        while a[j] > pivot:
            j -= 1
            comparisons += 1
        if i >= j:
            return j                    # a[lo..j] <= pivot <= a[j+1..hi]
        a[i], a[j] = a[j], a[i]
        swaps += 1


def quick_lomuto(a, lo, hi, rule, depth=0):
    global max_depth
    if lo >= hi:
        return
    max_depth = max(max_depth, depth)
    # median dari tiga: a[lo], a[mid], a[hi]
    if rule == 1:
        mid = lo + (hi - lo) // 2
        if a[mid] < a[lo]:
            a[mid], a[lo] = a[lo], a[mid]
        if a[hi] < a[lo]:
            a[hi], a[lo] = a[lo], a[hi]
        if a[mid] < a[hi]:
            a[mid], a[hi] = a[hi], a[mid]    # kini a[hi] memuat mediannya
    elif rule == 2:                      # pivot acak
        r = lo + next_random(hi - lo + 1)
        a[r], a[hi] = a[hi], a[r]
    before = show(a, lo, hi) if trace else ""
    p = lomuto_partition(a, lo, hi)
    if trace:
        print(" " * (2 * depth + 2) + f"partition {before} pivot {a[p]} -> "
              f"{show(a, lo, hi)}, pivot at {p}")
    quick_lomuto(a, lo, p - 1, rule, depth + 1)
    quick_lomuto(a, p + 1, hi, rule, depth + 1)
```

Kode 3.24 (Lanjutan) *quick_sort.py: quicksort dalam Python.*

```
def quick_hoare(a, lo, hi, depth=0):
    if lo >= hi:
        return
    before = show(a, lo, hi)
    pivot = a[lo + (hi - lo) // 2]
    p = hoare_partition(a, lo, hi)
    print(" " * (2 * depth + 2) + f"partition {before} pivot {pivot} -> "
          f"{show(a, lo, p)} {show(a, p + 1, hi)}")
    quick_hoare(a, lo, p, depth + 1)
    quick_hoare(a, p + 1, hi, depth + 1)


data = [22, 10, 15, 3, 8, 2]
print("Quicksort, Lomuto partition, last element as pivot:")
trace = True
quick_lomuto(data, 0, 5, 0)
print(f"  result {show(data, 0, 5)}, comparisons = {comparisons}, "
      f"swaps = {swaps}")

data2 = [22, 10, 15, 3, 8, 2]
comparisons = swaps = 0
print("Quicksort, Hoare partition, middle element as pivot:")
quick_hoare(data2, 0, 5)
print(f"  result {show(data2, 0, 5)}, comparisons = {comparisons}, "
      f"swaps = {swaps}")

trace = False
# masukan terurut + pivot elemen terakhir: ~3000 tingkat
sys.setrecursionlimit(10000)
n = 3000
print(f"Pivot rules on {n} elements (Lomuto):")
for kind in ("sorted", "random"):
    for rule, name in enumerate(("last element", "median of three", "random")):
        state = 2026
        if kind == "sorted":
            a = list(range(n))
        else:
            a = [next_random(1000000) for _ in range(n)]
        comparisons = swaps = max_depth = 0
        quick_lomuto(a, 0, n - 1, rule)
        print(f"  {kind} input, {name}: {comparisons} comparisons, "
              f"depth {max_depth}")
```

Ketiga program mencetak keluaran yang sama:

```
Quicksort, Lomuto partition, last element as pivot:
  partition [22 10 15 3 8 2] pivot 2 -> [2 10 15 3 8 22], pivot at 0
    partition [10 15 3 8 22] pivot 22 -> [10 15 3 8 22], pivot at 5
      partition [10 15 3 8] pivot 8 -> [3 8 10 15], pivot at 2
        partition [10 15] pivot 15 -> [10 15], pivot at 4
  result [2 3 8 10 15 22], comparisons = 13, swaps = 3
Quicksort, Hoare partition, middle element as pivot:
  partition [22 10 15 3 8 2] pivot 15 -> [2 10 8 3] [15 22]
    partition [2 10 8 3] pivot 10 -> [2 3 8] [10]
      partition [2 3 8] pivot 3 -> [2 3] [8]
        partition [2 3] pivot 2 -> [2] [3]
    partition [15 22] pivot 15 -> [15] [22]
  result [2 3 8 10 15 22], comparisons = 24, swaps = 3
Pivot rules on 3000 elements (Lomuto):
  sorted input, last element: 4498500 comparisons, depth 2998
  sorted input, median of three: 28917 comparisons, depth 10
  sorted input, random: 41022 comparisons, depth 23
  random input, last element: 38004 comparisons, depth 23
  random input, median of three: 33207 comparisons, depth 18
  random input, random: 40509 comparisons, depth 27
```

3.9.5 Kasus terbaik, terburuk, dan rata-rata

Mempartisi m elemen memerlukan $m-1$ perbandingan. Apa yang terjadi sesudahnya sepenuhnya bergantung pada di mana pivot mendarat.

Kasus terburuk. Bila pivot selalu elemen terkecil atau terbesar, satu sisi kosong dan sisi lain berisi $m-1$ elemen. Rekursinya lalu mempunyai n tingkat, berbiaya $(n-1)+(n-2)+\cdots+1$ perbandingan:

$$T_{\text{worst}}(n) = T_{\text{worst}}(n-1) + (n-1) = \frac{n(n-1)}{2} = \Theta(n^2). \tag{3.10}$$

Dengan elemen terakhir sebagai pivot, hal ini terjadi pada masukan *terurut*, masukan yang paling lazim dari semuanya. Eksperimen itu menegaskan Persamaan (3.10) sampai ke digit terakhir: $3000 \cdot 2999/2 = 4{,}498{,}500$ perbandingan dan rekursi sedalam 2998 tingkat, dibandingkan sekitar 38,000 perbandingan pada masukan acak berukuran sama. Kedalaman itu sama berbahayanya dengan waktunya. Setiap tingkat rekursi menempati memori pada tumpukan panggilan, dan larik terurut sejuta elemen akan memerlukan sejuta pemanggilan bersarang, yang meruntuhkan program dalam bahasa mana pun dari ketiga bahasa kita.

Kasus terbaik. Bila setiap pivot mendarat di tengah, setiap pemanggilan membelah rentangnya menjadi dua, dan rekurensinya kembali Persamaan (3.8), dengan penyelesaian $\Theta(n\lg n)$ menurut Persamaan (3.9).

Kasus rata-rata. Dengan masukan acak, atau dengan pivot acak, pivot mendarat di posisi acak. Sebagian belahan buruk dan sebagian baik, tetapi belahan 1:9 sama bergunanya dengan 1:1 sampai faktor konstanta, dan belahan buruk jarang terjadi. Analisis cermat [6], [1] memberikan harapan banyaknya perbandingan

$$\overline{C}_{\text{quick}}(n) = 2(n+1)H_n - 4n \approx 2n\ln n \approx 1.39\, n\lg n, \tag{3.11}$$

dengan $H_n = 1 + 1/2 + \cdots + 1/n \approx \ln n$ sebagai bilangan harmonik ke-n. Itu hanya 39% lebih banyak perbandingan daripada kasus terbaik, dan gelung dalam *quicksort* begitu sederhana (satu perbandingan, satu penambahan, sesekali satu pertukaran, semuanya pada memori yang

bertetangga) sehingga dalam praktik ia biasanya pengurutan berbasis perbandingan yang tercepat.

Memilih pivot. Eksperimen itu memperlihatkan apa yang diperoleh dari aturan pivot.

- *Elemen terakhir (atau pertama)*: baik pada data acak, kuadratik pada data terurut atau terbalik. Jangan pernah memakainya pada data yang tidak Anda kendalikan.
- *Median dari tiga*: pada data terurut, median dari elemen pertama, tengah, dan terakhir adalah tengah yang sebenarnya, sehingga masukan terurut justru menjadi kasus *terbaik*: 28,917 perbandingan dan kedalaman 10, dekat dengan $\lg 3000 \approx 11.6$. Pada data acak ia juga sedikit membantu, karena median tiga sampel merupakan tebakan tengah yang lebih baik daripada satu sampel. Ia masih dapat dikalahkan oleh masukan yang sengaja dirancang untuk mengelabuinya.
- *Elemen acak*: waktu jalan tidak lagi bergantung sama sekali pada urutan masukan; setiap masukan mempunyai biaya harapan yang sama Persamaan (3.11). Jalan kuadratik masih mungkin tetapi peluangnya luar biasa kecil.

Quicksort bersifat **di tempat** kecuali tumpukan rekursinya, dan **tidak stabil**: partisi menukar elemen melintasi jarak jauh. Dua penyempurnaan membuatnya aman untuk produksi. Pertama, berekursilah hanya ke sisi yang *lebih kecil* dan ulangi dengan gelung untuk sisi yang lebih besar; dengan begitu tumpukan tidak pernah melampaui $\lg n$ tingkat, dan program tolok ukur pada Subbab 3.13 melakukan persis hal itu. Kedua, bila rekursi menjadi terlalu dalam, berhenti dan beralih ke *heap sort*, yang tidak pernah kuadratik. Gagasan kedua itu adalah **introsort**, algoritma di dalam `std::sort` (Subbab 3.14).

3.10 Heap Sort: Pandangan Pertama

Selection sort menghabiskan hampir seluruh waktunya untuk *menemukan* elemen terkecil (atau terbesar) yang tersisa: satu penyusuran penuh, putaran demi putaran, yang melupakan semua yang dipelajarinya pada putaran sebelumnya. *Heap sort* mempertahankan rencana *selection sort*, "berulang kali ambil yang terbesar dan letakkan di ujung", tetapi menyimpan elemen-elemen yang belum terurut dalam bentuk yang membuat elemen terbesar gratis ditemukan dan murah diganti. Bentuk itu adalah **binary heap** (*heap* biner). Subbab ini hanya memperlihatkan secukupnya untuk mengurutkan; Bab 14 membangun *heap* secara lengkap, membuktikan biayanya, dan memakainya sebagai antrean prioritas.

3.10.1 Pohon yang tersembunyi di dalam larik

Bacalah larik $A[0..n-1]$ sebagai pohon biner: $A[0]$ adalah akar, dan anak-anak simpul di indeks i berada di indeks

$$\text{left}(i) = 2i + 1, \qquad \text{right}(i) = 2i + 2, \qquad \text{parent}(i) = \lfloor (i-1)/2 \rfloor. \tag{3.12}$$

Tidak ada pointer yang disimpan; pohon itu murni aritmetika indeks. Pohon terisi tingkat demi tingkat dari kiri, sehingga tingginya $\lfloor \lg n \rfloor$: enam elemen membentuk pohon setinggi 2, sejuta elemen membentuk pohon setinggi 19.

> **DEFINISI.** Sebuah larik adalah **max-heap** (tumpukan-maks) bila setiap simpul paling sedikit sama besar dengan anak-anaknya: $A[\text{parent}(i)] \geq A[i]$ untuk setiap $i \geq 1$. Maka elemen terbesar berada di akar, $A[0]$.

Dua operasi saja yang kita perlukan.

- **Sift down** (menenggelamkan) memperbaiki simpul yang mungkin lebih kecil daripada salah satu anaknya, bila kedua subpohon di bawahnya sudah berupa *heap*: tukar simpul itu dengan anaknya yang lebih besar, dan ulangi dari posisi anak itu, sampai nilainya paling sedikit sama besar dengan kedua anaknya atau mencapai daun. Ia menuruni satu lintasan pohon, sehingga biayanya paling banyak $2\lfloor \lg n \rfloor$ perbandingan.
- **Membangun heap** dari larik sembarang dengan menenggelamkan setiap simpul yang mempunyai anak, dari yang terakhir, indeks $\lfloor n/2 \rfloor - 1$, mundur sampai akar. Bab 14 membuktikan bahwa biaya totalnya hanya $O(n)$, lebih kecil daripada $O(n \lg n)$ yang tampaknya harus dibayar.

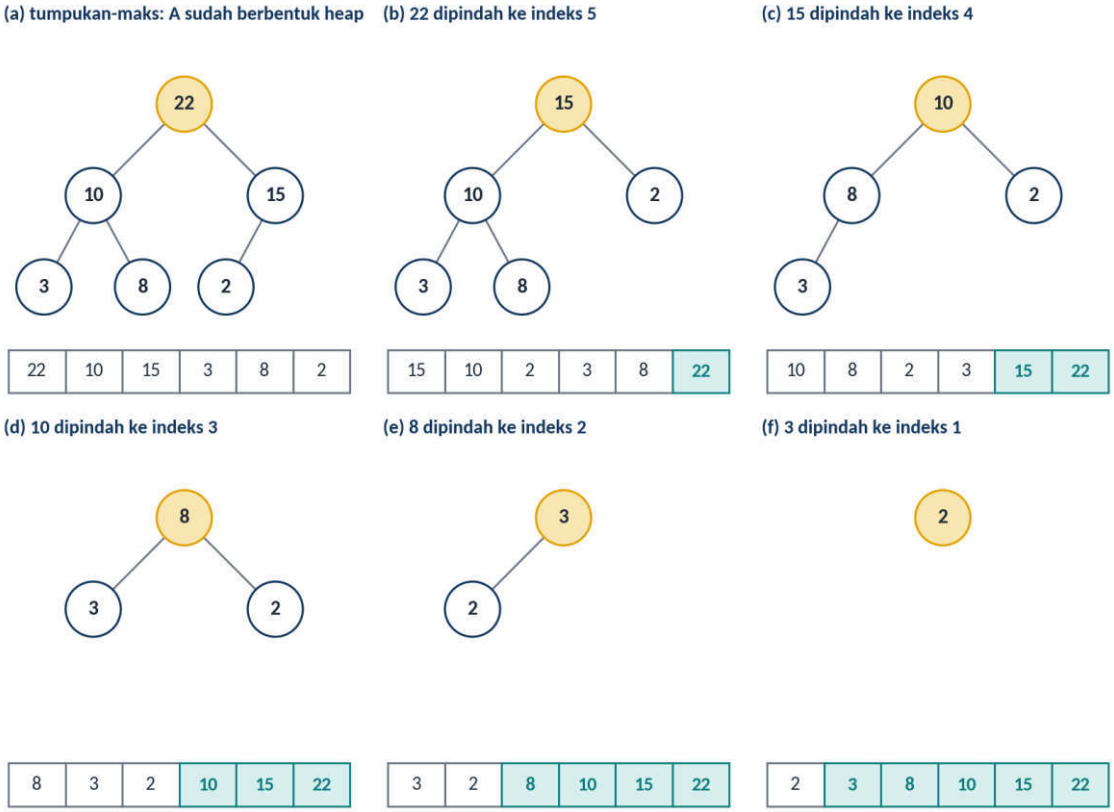
Pseudokode 3.10 `SIFT-DOWN` dan `HEAP-SORT`: bangun *max-heap*, lalu berulang kali pindahkan akar ke ujung *heap* yang menyusut dan perbaiki akarnya.

```
SIFT-DOWN(A, i, size)                          // heap-nya A[0..size-1]
 1  loop
 2      largest = i; l = 2i + 1; r = 2i + 2
 3      if l < size and A[l] > A[largest]  then largest = l
 4      if r < size and A[r] > A[largest]  then largest = r
 5      if largest == i  then return
 6      exchange A[i] with A[largest]
 7      i = largest

HEAP-SORT(A, n)
 1  for i = ⌊n/2⌋ - 1 down to 0               // bangun max-heap
 2      SIFT-DOWN(A, i, n)
 3  for end = n - 1 down to 1
 4      exchange A[0] with A[end]              // maksimum ke tempatnya
 5      SIFT-DOWN(A, 0, end)                   // perbaiki heap yang lebih kecil
```

3.10.2 Jejaknya

Larik berjalan kita kebetulan sudah berupa *max-heap*: $22 \geq 10$ dan $22 \geq 15$ di akar, $10 \geq 3$ dan $10 \geq 8$ di bawah anak kiri, $15 \geq 2$ di bawah anak kanan. Jadi fase pembangunan memeriksa tiga simpul dan tidak memindahkan apa pun. **Gambar 3.11** memperlihatkan *heap* itu dan lima langkah pengambilan sesudahnya.



Gambar 3.11 *Heap sort* pada *A*. Setiap panel memperlihatkan *heap* sebagai pohon (akar berwarna kuning tua) dan seluruh larik di bawahnya; ekor berwarna hijau toska adalah bagian yang sudah terurut, yang bertambah satu elemen pada setiap langkah.

Langkah (b) menukar akar 22 dengan elemen *heap* terakhir 2, sehingga 22 berada di tempat akhirnya di indeks 5 dan *heap* menyusut menjadi lima elemen. Akar baru 2 lebih kecil daripada kedua anaknya 10 dan 15, sehingga ia tenggelam, bertukar dengan anak yang lebih besar, 15; di indeks 2 ia tidak lagi mempunyai anak di dalam *heap*, dan *heap*-nya menjadi [15,10,2,3,8]. Langkah (c) memindahkan 15 ke indeks 4; akar baru 8 tenggelam ke bawah 10, menghasilkan [10,8,2,3]. Langkah (d) sampai (f) berlanjut hingga *heap* berisi satu elemen dan larik terurut.

Kode 3.25 `heap_sort.cpp`: *heap sort* pada larik berjalan, dan pada larik yang harus dijadikan *heap* lebih dahulu.

```cpp
// heap_sort.cpp - sekilas heap sort: bangun heap-maks, ambil maksimumnya
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

string show(const vector<int>& a, int lo, int hi) {  // a[lo..hi-1]
    string s = "[";
    for (int i = lo; i < hi; ++i) s += (i > lo ? " " : "") + to_string(a[i]);
    return s + "]";
}

long long comparisons = 0, swaps = 0;

// Biarkan a[i] tenggelam sampai tak ada anak yang lebih besar; heap-nya
// a[0..size-1].
void siftDown(vector<int>& a, int i, int size) {
    while (true) {
        int largest = i, left = 2 * i + 1, right = 2 * i + 2;
        if (left < size) {
            ++comparisons;
            if (a[left] > a[largest]) largest = left;
        }
        if (right < size) {
            ++comparisons;
            if (a[right] > a[largest]) largest = right;
        }
        if (largest == i) return;
        swap(a[i], a[largest]);
        ++swaps;
        i = largest;
    }
}

void heapSort(vector<int>& a) {
    int n = static_cast<int>(a.size());
    // bangun heap dari bawah ke atas
    for (int i = n / 2 - 1; i >= 0; --i) siftDown(a, i, n);
    cout << "  max-heap built: " << show(a, 0, n) << "\n";
    for (int end = n - 1; end > 0; --end) {
        swap(a[0], a[end]);  // maksimum masuk ke slot akhirnya
        ++swaps;
        // perbaiki heap yang lebih kecil
        siftDown(a, 0, end);
        cout << "  " << a[end] << " placed: heap " << show(a, 0, end)
```

Kode 3.25 (Lanjutan) *heap_sort.cpp: heap sort pada larik berjalan, dan pada larik yang harus dijadikan heap lebih dahulu.*

```
              << " | sorted " << show(a, end, n) << "\n";
    }
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Heap sort on " << show(data, 0, 6) << "\n";
    heapSort(data);
    cout << "  result " << show(data, 0, 6) << ", comparisons = " << comparisons
         << ", swaps = " << swaps << "\n";

    vector<int> other = {3, 8, 2, 22, 10, 15};
    comparisons = swaps = 0;
    cout << "Heap sort on " << show(other, 0, 6) << "\n";
    heapSort(other);
    cout << "  result " << show(other, 0, 6) << ", comparisons = "
         << comparisons << ", swaps = " << swaps << "\n";
    return 0;
}
```

Kode 3.26 `HeapSort.java`: *heap sort* dalam Java.

```java
// HeapSort.java - sekilas heap sort: bangun heap-maks, ambil maksimumnya
public class HeapSort {
    static String show(int[] a, int lo, int hi) {    // a[lo..hi-1]
        StringBuilder sb = new StringBuilder("[");
        for (int i = lo; i < hi; i++)
            sb.append(i > lo ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static long comparisons = 0, swaps = 0;

    static void swap(int[] a, int i, int j) {
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }

    // Biarkan a[i] tenggelam sampai tak ada anak yang lebih besar; heap-nya
    // a[0..size-1].
    static void siftDown(int[] a, int i, int size) {
        while (true) {
            int largest = i, left = 2 * i + 1, right = 2 * i + 2;
            if (left < size) {
                comparisons++;
                if (a[left] > a[largest]) largest = left;
            }
            if (right < size) {
                comparisons++;
                if (a[right] > a[largest]) largest = right;
            }
            if (largest == i) return;
            swap(a, i, largest);
            swaps++;
            i = largest;
        }
    }

    static void heapSort(int[] a) {
        int n = a.length;
        // bangun heap dari bawah ke atas
        for (int i = n / 2 - 1; i >= 0; i--) siftDown(a, i, n);
        System.out.println("  max-heap built: " + show(a, 0, n));
        for (int end = n - 1; end > 0; end--) {
            swap(a, 0, end);  // maksimum masuk ke slot akhirnya
            swaps++;
            // perbaiki heap yang lebih kecil
            siftDown(a, 0, end);
            System.out.println("  " + a[end] + " placed: heap "
```

Kode 3.26 (Lanjutan) *HeapSort.java: heap sort dalam Java.*

```
                    + show(a, 0, end)
                    + " | sorted " + show(a, end, n));
        }
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println("Heap sort on " + show(data, 0, 6));
        heapSort(data);
        System.out.println("  result " + show(data, 0, 6) + ", comparisons = "
                + comparisons
                + ", swaps = " + swaps);

        int[] other = {3, 8, 2, 22, 10, 15};
        comparisons = swaps = 0;
        System.out.println("Heap sort on " + show(other, 0, 6));
        heapSort(other);
        System.out.println("  result " + show(other, 0, 6)
                + ", comparisons = " + comparisons
                + ", swaps = " + swaps);
    }
}
```

Kode 3.27 `heap_sort.py`: *heap sort* dalam Python.

```
# heap_sort.py - sekilas heap sort: bangun heap-maks, ambil maksimumnya
def show(a, lo, hi):                    # a[lo..hi-1]
    return "[" + " ".join(map(str, a[lo:hi])) + "]"


comparisons = swaps = 0


def sift_down(a, i, size):
    """a[i] tenggelam sampai tak ada anak lebih besar; heap: a[0..size-1]."""
    global comparisons, swaps
    while True:
        largest, left, right = i, 2 * i + 1, 2 * i + 2
        if left < size:
            comparisons += 1
            if a[left] > a[largest]:
                largest = left
        if right < size:
            comparisons += 1
            if a[right] > a[largest]:
                largest = right
        if largest == i:
            return
        a[i], a[largest] = a[largest], a[i]
        swaps += 1
        i = largest


def heap_sort(a):
    global swaps
    n = len(a)
    for i in range(n // 2 - 1, -1, -1):  # bangun heap dari bawah ke atas
        sift_down(a, i, n)
    print(f"  max-heap built: {show(a, 0, n)}")
    for end in range(n - 1, 0, -1):
        a[0], a[end] = a[end], a[0]      # maksimum masuk ke slot akhirnya
        swaps += 1
        sift_down(a, 0, end)             # perbaiki heap yang lebih kecil
        print(f"  {a[end]} placed: heap {show(a, 0, end)} | "
              f"sorted {show(a, end, n)}")
```

Kode 3.27 (Lanjutan) *heap_sort.py: heap sort dalam Python.*

```
for data in ([22, 10, 15, 3, 8, 2], [3, 8, 2, 22, 10, 15]):
    comparisons = swaps = 0
    print(f"Heap sort on {show(data, 0, 6)}")
    heap_sort(data)
    print(f"  result {show(data, 0, 6)}, comparisons = {comparisons}, "
          f"swaps = {swaps}")
```

Kode 3.25, **Kode 3.26**, dan **Kode 3.27** mencetak:

```
Heap sort on [22 10 15 3 8 2]
  max-heap built: [22 10 15 3 8 2]
  22 placed: heap [15 10 2 3 8] | sorted [22]
  15 placed: heap [10 8 2 3] | sorted [15 22]
  10 placed: heap [8 3 2] | sorted [10 15 22]
  8 placed: heap [3 2] | sorted [8 10 15 22]
  3 placed: heap [2] | sorted [3 8 10 15 22]
  result [2 3 8 10 15 22], comparisons = 13, swaps = 9
Heap sort on [3 8 2 22 10 15]
  max-heap built: [22 10 15 8 3 2]
  22 placed: heap [15 10 2 8 3] | sorted [22]
  15 placed: heap [10 8 2 3] | sorted [15 22]
  10 placed: heap [8 3 2] | sorted [10 15 22]
  8 placed: heap [3 2] | sorted [8 10 15 22]
  3 placed: heap [2] | sorted [3 8 10 15 22]
  result [2 3 8 10 15 22], comparisons = 15, swaps = 14
```

Jalannya yang kedua dimulai dari [3,8,2,22,10,15], yang bukan *heap*; pembangunan *heap* memindahkan 22 ke akar dan menghasilkan [22,10,15,8,3,2], dan dari sana pengambilan berlangsung seperti sebelumnya.

Masing-masing dari $n-1$ pengambilan berbiaya satu pertukaran dan satu penenggelaman dengan paling banyak $2\lg n$ perbandingan, dan pembangunannya berbiaya $O(n)$, sehingga *heap sort* melakukan $O(n\lg n)$ perbandingan **pada kasus terburuk**, tanpa masukan buruk sama sekali, dan ia bersifat **di tempat**: *heap* dan ekor terurut berbagi satu larik. Inilah kombinasi yang tidak dimiliki algoritma lain mana pun di bab ini: *merge sort* $O(n\lg n)$ tetapi memerlukan penyangga, dan *quicksort* di tempat tetapi bisa kuadratik. Harga yang dibayar *heap sort* adalah penenggelamannya melompat-lompat di dalam larik (i, $2i+1$, $4i+3$, …), yang tidak ramah terhadap *cache* prosesor, sehingga dalam praktik ia biasanya lebih lambat daripada *quicksort*, seperti diukur pada Subbab 3.13. Ia **tidak stabil**. *Heap sort* diterbitkan oleh Williams pada tahun 1964 [7], dengan pembangunan waktu-linear oleh Floyd pada tahun yang sama [8].

3.11 Counting Sort dan Radix Sort

3.11.1 Batas yang tidak dapat dilintasi pengurutan berbasis perbandingan

Semua algoritma sejauh ini mempelajari data hanya dengan bertanya "apakah x sebelum y?". Ada batas seberapa cepat algoritma semacam itu dapat bekerja. Bayangkan pengurutan sebagai permainan tebak-tebakan dua puluh pertanyaan. Ada $n!$ kemungkinan urutan dari n kunci berbeda, dan pengurutan harus membedakan semuanya, karena masing-masing memerlukan penataan ulang yang berbeda. Setiap perbandingan mempunyai dua kemungkinan jawaban, sehingga k perbandingan dapat membedakan paling banyak 2^k kasus. Untuk membedakan $n!$ kasus kita memerlukan $2^k \geq n!$, yaitu

$$k \;\geq\; \lg(n!) \;\geq\; \lg\left(\left(\frac{n}{2}\right)^{n/2}\right) = \frac{n}{2}\lg\frac{n}{2} = \Omega(n\lg n). \tag{3.13}$$

(Langkah tengahnya hanya mempertahankan $n/2$ faktor terbesar dari $n!$, yang masing-masing paling sedikit $n/2$.) Jadi **setiap pengurutan berbasis perbandingan memerlukan** $\Omega(n\lg n)$ **perbandingan pada kasus terburuk**; *merge sort* dan *heap sort* optimal sampai faktor konstanta [6]. Untuk enam elemen, $\lg(6!) = \lg 720 \approx 9.49$, sehingga ada masukan yang memerlukan paling sedikit 10 perbandingan apa pun algoritmanya; *merge sort* kita hanya memerlukan 9 pada A, dan itu diperbolehkan, karena batas itu berbicara tentang masukan *terburuk*.

Jalan melewati batas itu adalah berhenti membandingkan. Bila kuncinya bilangan bulat kecil, atau untaian digit, kita dapat memakai *nilai* kunci itu langsung sebagai indeks larik.

3.11.2 Counting sort

Misalkan setiap kunci adalah bilangan bulat antara 0 dan $k-1$. **Counting sort**, **Pseudokode 3.11**, bekerja dalam tiga gelung sederhana.

1. **Hitung**: `count[v]` = banyaknya elemen dengan kunci v.
2. **Jumlah kumulatif**: ganti setiap hitungan dengan jumlah dirinya dan semua hitungan sebelumnya. Kini `count[v]` adalah banyaknya elemen dengan kunci $\leq v$, yang persis satu lewat dari posisi terakhir tempat kunci v berada dalam keluaran.
3. **Tempatkan**: susuri masukan *dari kanan ke kiri*; untuk setiap elemen dengan kunci v, kurangi `count[v]` dan letakkan elemen itu pada posisi tersebut di larik keluaran.

Pseudokode 3.11 *Counting sort* untuk kunci bilangan bulat di $0..k-1$. Tidak ada dua kunci yang pernah dibandingkan.

```
COUNTING-SORT(A, n, k)                          // kunci 0..k-1; keluaran B
 1  count[0..k-1] = 0
 2  for i = 0 to n - 1
 3      count[A[i]] = count[A[i]] + 1
 4  for v = 1 to k - 1
 5      count[v] = count[v] + count[v-1]          // kini: banyak kunci ≤ v
 6  for i = n - 1 down to 0                      // kanan ke kiri: stabil
 7      count[A[i]] = count[A[i]] - 1
 8      B[count[A[i]]] = A[i]
 9  return B
```

Gambar 3.12 menerapkannya pada A dengan $k = 23$ (kunci 0 sampai 22).



Gambar 3.12 *Counting sort* pada A dengan kunci 0 sampai 22. Hitungan (kuning tua) menjadi jumlah kumulatif (biru), dan setiap jumlah kumulatif menyatakan di mana kuncinya berakhir dalam keluaran.

Jumlah kumulatif pada 8 adalah 3: tiga kunci (2, 3, dan 8) bernilai ≤ 8, sehingga 8 berada di indeks $3-1=2$. Baris 6 menyusuri dari kanan ke kiri agar, bila beberapa elemen berbagi kunci, yang terakhir menempati slot terakhir, yang sebelumnya menempati slot sebelumnya, dan seterusnya; kunci yang sama dengan demikian mempertahankan urutan aslinya, dan *counting sort* **stabil**. Kestabilan itu bukan kemewahan: *radix sort* bergantung padanya.

Counting sort berjalan dalam waktu yang sebanding dengan $n+k$: dua gelung atas n elemen dan satu gelung atas k pencacah.

$$T_{\text{counting}}(n,k) = \Theta(n+k). \tag{3.14}$$

Bila k tidak lebih besar daripada suatu konstanta kali n, waktunya linear, dan itu mengalahkan $n\lg n$. Bila k sangat besar (kunci sampai satu miliar), larik pencacahnya terlalu besar untuk dipakai, dan di situlah *radix sort* berperan.

3.11.3 Radix sort

Radix sort mengurutkan bilangan digit demi digit, mulai dari digit yang **paling tidak signifikan** (*least significant digit*, LSD), dengan *counting sort* yang stabil dan $k = 10$ untuk setiap digit. Kedengarannya terbalik, karena kita membandingkan bilangan menurut digit terdepannya, dan justru karena itulah kestabilan penting: ketika digit puluhan diurutkan pada lintasan kedua, bilangan-bilangan dengan digit puluhan yang sama mempertahankan urutan yang diberikan lintasan pertama, yaitu urutan digit satuannya. **Gambar 3.13** mengurutkan A dalam dua lintasan.



Gambar 3.13 *Radix sort* LSD pada A. Lintasan pertama menyebar menurut digit satuan; lintasan kedua, menurut digit puluhan, menjaga setiap ember dalam urutan lintasan pertama, dan larik keluar dalam keadaan terurut.

Setelah lintasan digit satuan urutannya [10,22,2,3,15,8]: 22 datang sebelum 2 karena ia datang lebih dahulu dalam masukan dan keduanya berdigit satuan 2. Pada lintasan digit puluhan, 2, 3, dan 8 semuanya berdigit puluhan 0 dan, karena lintasannya stabil, tetap dalam urutan 2,3,8 yang dihasilkan lintasan pertama; 10 dan 15 berbagi digit puluhan 1 dan tetap berurutan 10,15. Hasilnya terurut. Untuk kunci d digit dalam basis b, *radix sort* melakukan d lintasan *counting sort*:

$$T_{\text{radix}}(n) = \Theta(d\,(n+b)). \tag{3.15}$$

Untuk bilangan bulat 32-bit, tolok ukur pada Subbab 3.13 memakai basis $b = 256$, satu byte per lintasan, sehingga $d = 4$ dan setiap lintasan memakai 256 pencacah. **Kode 3.28** mengimplementasikan kedua algoritma; digit yang menjadi dasar pengurutan diteruskan ke *counting sort* sebagai fungsi kecil (sebuah *lambda*), sehingga satu *counting sort* melayani kedua keperluan. Contoh terakhir mengurutkan tujuh bilangan tiga digit dalam tiga lintasan.

Kode 3.28 `counting_radix.cpp`: *counting sort* stabil dengan parameter fungsi kunci, dan *radix sort* LSD yang dibangun di atasnya.

```cpp
// counting_radix.cpp - counting sort dan radix sort LSD, tanpa perbandingan
#include <iostream>
#include <string>
#include <vector>
using namespace std;

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

// Counting sort stabil atas a menurut key(x) di 0..k-1; mengembalikan salinan
// terurut.
template <typename Key>
vector<int> countingSort(const vector<int>& a, int k, Key key, bool trace) {
    vector<int> count(k, 0);
    // 1. berapa banyak tiap kunci
    for (int x : a) ++count[key(x)];
    if (trace) cout << "  counts:        " << show(count) << "\n";
    // 2. jumlah kumulatif
    for (int v = 1; v < k; ++v) count[v] += count[v - 1];
    if (trace) cout << "  running total: " << show(count) << "\n";
    vector<int> out(a.size());
    // 3. kanan ke kiri: stabil
    for (int i = static_cast<int>(a.size()) - 1; i >= 0; --i)
        out[--count[key(a[i])]] = a[i];
    return out;
}

// basis 10, digit paling tidak signifikan dulu
vector<int> radixSort(vector<int> a, bool trace) {
    int maxValue = 0;
    for (int x : a) if (x > maxValue) maxValue = x;
    for (int place = 1; maxValue / place > 0; place *= 10) {
        auto digit = [place](int x) { return x / place % 10; };
        a = countingSort(a, 10, digit, false);
        if (trace)
            cout << "  by digit of " << place << "s: " << show(a) << "\n";
    }
    return a;
}
```

Kode 3.28 (Lanjutan) *counting_radix.cpp: counting sort stabil dengan parameter fungsi kunci, dan radix sort LSD yang dibangun di atasnya.*

```cpp
int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Counting sort on " << show(data) << " (keys 0..22)\n";
    vector<int> sorted = countingSort(data, 23, [](int x) { return x; }, true);
    cout << "  result " << show(sorted) << "\n";

    cout << "Radix sort (base 10) on " << show(data) << "\n";
    sorted = radixSort(data, true);
    cout << "  result " << show(sorted) << "\n";

    vector<int> big = {329, 457, 657, 839, 436, 720, 355};
    cout << "Radix sort (base 10) on " << show(big) << "\n";
    sorted = radixSort(big, true);
    cout << "  result " << show(sorted) << "\n";
    return 0;
}
```

Kode 3.29 `CountingRadix.java`: *counting sort* dan *radix sort* dalam Java, dengan `IntUnaryOperator` sebagai fungsi kunci.

```java
// CountingRadix.java - counting sort dan radix sort LSD, tanpa perbandingan
import java.util.function.IntUnaryOperator;

public class CountingRadix {
    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    // Counting sort stabil atas a menurut key(x) di 0..k-1; mengembalikan
    // salinan terurut.
    static int[] countingSort(int[] a, int k, IntUnaryOperator key,
                              boolean trace) {
        int[] count = new int[k];
        // 1. berapa banyak tiap kunci
        for (int x : a) count[key.applyAsInt(x)]++;
        if (trace) System.out.println("  counts:        " + show(count));
        // 2. jumlah kumulatif
        for (int v = 1; v < k; v++) count[v] += count[v - 1];
        if (trace) System.out.println("  running total: " + show(count));
        int[] out = new int[a.length];
        for (int i = a.length - 1; i >= 0; i--)  // 3. kanan ke kiri: stabil
            out[--count[key.applyAsInt(a[i])]] = a[i];
        return out;
    }

    // basis 10, digit terendah dulu
    static int[] radixSort(int[] a, boolean trace) {
        int maxValue = 0;
        for (int x : a) if (x > maxValue) maxValue = x;
        for (int place = 1; maxValue / place > 0; place *= 10) {
            final int p = place;
            a = countingSort(a, 10, x -> x / p % 10, false);
            if (trace) System.out.println("  by digit of " + place + "s: "
                    + show(a));
        }
        return a;
    }
```

Kode 3.29 (Lanjutan) *CountingRadix.java: counting sort dan radix sort dalam Java, dengan IntUnaryOperator sebagai fungsi kunci.*

```java
    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println("Counting sort on " + show(data) + " (keys 0..22)");
        int[] sorted = countingSort(data, 23, x -> x, true);
        System.out.println("  result " + show(sorted));

        System.out.println("Radix sort (base 10) on " + show(data));
        sorted = radixSort(data, true);
        System.out.println("  result " + show(sorted));

        int[] big = {329, 457, 657, 839, 436, 720, 355};
        System.out.println("Radix sort (base 10) on " + show(big));
        sorted = radixSort(big, true);
        System.out.println("  result " + show(sorted));
    }
}
```

Kode 3.30 `counting_radix.py`: *counting sort* dan *radix sort* dalam Python, dengan sebuah `lambda` sebagai fungsi kunci.

```python
# counting_radix.py - counting sort dan radix sort LSD, tanpa perbandingan
def show(a):
    return "[" + " ".join(map(str, a)) + "]"


def counting_sort(a, k, key, trace=False):
    """Counting sort stabil atas a menurut key(x) di 0..k-1; hasil: salinan."""
    count = [0] * k
    for x in a:                          # 1. berapa banyak tiap kunci
        count[key(x)] += 1
    if trace:
        print(f"  counts:         {show(count)}")
    for v in range(1, k):                # 2. jumlah kumulatif
        count[v] += count[v - 1]
    if trace:
        print(f"  running total: {show(count)}")
    out = [0] * len(a)
    for x in reversed(a):                # 3. kanan ke kiri: stabil
        count[key(x)] -= 1
        out[count[key(x)]] = x
    return out


def radix_sort(a, trace=False):  # basis 10, digit paling tidak signifikan dulu
    max_value = max(a)
    place = 1
    while max_value // place > 0:
        a = counting_sort(a, 10, lambda x: x // place % 10)
        if trace:
            print(f"  by digit of {place}s: {show(a)}")
        place *= 10
    return a


data = [22, 10, 15, 3, 8, 2]
print(f"Counting sort on {show(data)} (keys 0..22)")
print(f"  result {show(counting_sort(data, 23, lambda x: x, True))}")

print(f"Radix sort (base 10) on {show(data)}")
result = radix_sort(data, True)
print(f"  result {show(result)}")

big = [329, 457, 657, 839, 436, 720, 355]
print(f"Radix sort (base 10) on {show(big)}")
result = radix_sort(big, True)
print(f"  result {show(result)}")
```

Kode 3.29 dan **Kode 3.30** memberikan keluaran yang sama dengan **Kode 3.28**:

```
Counting sort on [22 10 15 3 8 2] (keys 0..22)
  counts:         [0 0 1 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1]
  running total: [0 0 1 2 2 2 2 2 3 3 4 4 4 4 4 5 5 5 5 5 5 5 6]
  result [2 3 8 10 15 22]
Radix sort (base 10) on [22 10 15 3 8 2]
  by digit of 1s: [10 22 2 3 15 8]
  by digit of 10s: [2 3 8 10 15 22]
  result [2 3 8 10 15 22]
Radix sort (base 10) on [329 457 657 839 436 720 355]
  by digit of 1s: [720 355 436 457 657 329 839]
  by digit of 10s: [720 329 436 839 355 457 657]
  by digit of 100s: [329 355 436 457 657 720 839]
  result [329 355 436 457 657 720 839]
```

Kedua pengurutan memerlukan larik keluaran berisi n elemen sehingga **tidak di tempat**; keduanya **stabil**. Keduanya tidak umum: bekerja pada kunci berupa bilangan bulat, atau yang dapat dipotong menjadi digit (*string* berpanjang tetap, tanggal), bukan pada objek sembarang

dengan fungsi pembanding. Dalam jangkauan itu keduanya adalah pengurutan tercepat di bab ini, seperti akan diperlihatkan tolok ukurnya.

3.12 Kestabilan, Di Tempat, dan Tabel Lengkapnya

3.12.1 Kestabilan yang dapat dilihat

Kestabilan paling mudah dipahami saat ia gagal. **Kode 3.31** mengambil delapan buku Pustaka Ceria dalam urutan id-nya, `B1001` sampai `B1008`, lalu mengurutkannya menurut kategori dua kali: sekali dengan *insertion sort*, yang stabil, dan sekali dengan *selection sort*, yang tidak stabil. Pembandingnya hanya melihat kategori, sehingga keempat buku berkategori *Fiksi* mempunyai kunci yang sama.

Kode 3.31 `stability_demo.cpp`: katalog diurutkan menurut kategori dengan pengurutan stabil dan tidak stabil. Pembandingnya diteruskan sebagai pointer fungsi.

```
// stability_demo.cpp - pengurutan stabil dan tidak stabil katalog menurut
// kategori
#include <cstdio>
#include <string>
#include <utility>
#include <vector>
using namespace std;

struct Book {
    string id, title, category;
};

bool categoryLess(const Book& a, const Book& b) {
    return a.category < b.category;
}

void insertionSort(vector<Book>& a, bool (*less)(const Book&, const Book&)) {
    for (size_t i = 1; i < a.size(); ++i) {
        Book key = a[i];
        size_t j = i;
        // tegas lebih kecil: kunci sama tak saling melewati
        while (j > 0 && less(key, a[j - 1])) {
            a[j] = a[j - 1];
            --j;
        }
        a[j] = key;
    }
}

void selectionSort(vector<Book>& a, bool (*less)(const Book&, const Book&)) {
    for (size_t i = 0; i + 1 < a.size(); ++i) {
        size_t pos = i;
        for (size_t j = i + 1; j < a.size(); ++j)
            if (less(a[j], a[pos])) pos = j;
        // lompatan jauh bisa melewati kunci yang sama
        if (pos != i) swap(a[i], a[pos]);
    }
}

void print(const char* heading, const vector<Book>& a) {
    printf("%s\n", heading);
    for (const Book& b : a)
        printf("  %-10s %s  %s\n", b.category.c_str(), b.id.c_str(),
               b.title.c_str());
}
```

Kode 3.31 (Lanjutan) *stability_demo.cpp: katalog diurutkan menurut kategori dengan pengurutan stabil dan tidak stabil. Pembandingnya diteruskan sebagai pointer fungsi.*

```
int main() {
    vector<Book> catalog = {
        {"B1001", "Laskar Pelangi", "Fiksi"},
        {"B1002", "Bumi Manusia", "Fiksi"},
        {"B1003", "Filosofi Kopi", "Fiksi"},
        {"B1004", "Sejarah Nusantara", "Sejarah"},
        {"B1005", "Algoritma & Struktur Data", "Teknologi"},
        {"B1006", "Cerita Rakyat Nusantara", "Anak-anak"},
        {"B1007", "Fisika Dasar", "Sains"},
        {"B1008", "Negeri 5 Menara", "Fiksi"},
    };
    // keduanya mulai dalam urutan id
    vector<Book> stable = catalog, unstable = catalog;
    insertionSort(stable, categoryLess);
    selectionSort(unstable, categoryLess);
    print("By category with insertion sort (stable):", stable);
    print("By category with selection sort (not stable):", unstable);
    return 0;
}
```

Kode 3.32 `StabilityDemo.java`: peragaan yang sama dalam Java, dengan `record` untuk buku dan `Comparator` untuk kuncinya.

```
// StabilityDemo.java - pengurutan stabil dan tidak stabil katalog menurut
// kategori
import java.util.Comparator;

public class StabilityDemo {
    record Book(String id, String title, String category) { }

    static void insertionSort(Book[] a, Comparator<Book> cmp) {
        for (int i = 1; i < a.length; i++) {
            Book key = a[i];
            int j = i;
            // tegas lebih kecil: kunci yang sama tak pernah melewati kunci
            // sebelumnya
            while (j > 0 && cmp.compare(key, a[j - 1]) < 0) {
                a[j] = a[j - 1];
                j--;
            }
            a[j] = key;
        }
    }

    static void selectionSort(Book[] a, Comparator<Book> cmp) {
        for (int i = 0; i + 1 < a.length; i++) {
            int pos = i;
            for (int j = i + 1; j < a.length; j++)
                if (cmp.compare(a[j], a[pos]) < 0) pos = j;
            // lompatan jauh bisa melewati kunci yang sama
            if (pos != i) {
                Book t = a[i]; a[i] = a[pos]; a[pos] = t;
            }
        }
    }

    static void print(String heading, Book[] a) {
        System.out.println(heading);
        for (Book b : a)
            System.out.printf("  %-10s %s  %s%n",
                    b.category(), b.id(), b.title());
    }
```

Kode 3.32 (Lanjutan) *StabilityDemo.java: peragaan yang sama dalam Java, dengan record untuk buku dan Comparator untuk kuncinya.*

```
    public static void main(String[] args) {
        Book[] catalog = {
            new Book("B1001", "Laskar Pelangi", "Fiksi"),
            new Book("B1002", "Bumi Manusia", "Fiksi"),
            new Book("B1003", "Filosofi Kopi", "Fiksi"),
            new Book("B1004", "Sejarah Nusantara", "Sejarah"),
            new Book("B1005", "Algoritma & Struktur Data", "Teknologi"),
            new Book("B1006", "Cerita Rakyat Nusantara", "Anak-anak"),
            new Book("B1007", "Fisika Dasar", "Sains"),
            new Book("B1008", "Negeri 5 Menara", "Fiksi"),
        };
        Comparator<Book> byCategory = Comparator.comparing(Book::category);
        // keduanya mulai dalam urutan id
        Book[] stable = catalog.clone(), unstable = catalog.clone();
        insertionSort(stable, byCategory);
        selectionSort(unstable, byCategory);
        print("By category with insertion sort (stable):", stable);
        print("By category with selection sort (not stable):", unstable);
    }
}
```

Kode 3.33 `stability_demo.py`: peragaan yang sama dalam Python, dengan `dataclass` dan fungsi kunci.

```
# stability_demo.py - pengurutan stabil dan tidak stabil katalog menurut
# kategori
from dataclasses import dataclass


@dataclass
class Book:
    id: str
    title: str
    category: str


def insertion_sort(a, key):
    for i in range(1, len(a)):
        item = a[i]
        j = i
        # tegas lebih kecil: kunci sama tak saling melewati
        while j > 0 and key(item) < key(a[j - 1]):
            a[j] = a[j - 1]
            j -= 1
        a[j] = item


def selection_sort(a, key):
    for i in range(len(a) - 1):
        pos = i
        for j in range(i + 1, len(a)):
            if key(a[j]) < key(a[pos]):
                pos = j
        if pos != i:
            # lompatan jauh bisa melewati kunci yang sama
            a[i], a[pos] = a[pos], a[i]


def print_books(heading, a):
    print(heading)
    for b in a:
        print(f"  {b.category:<10} {b.id}  {b.title}")
```

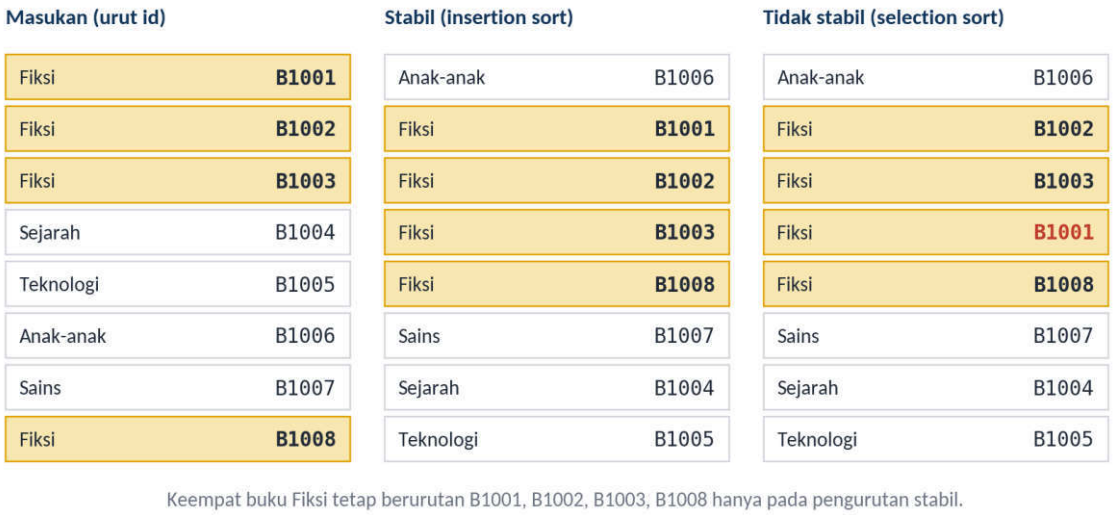
Kode 3.33 (Lanjutan) *stability_demo.py: peragaan yang sama dalam Python, dengan dataclass dan fungsi kunci.*

```
catalog = [
    Book("B1001", "Laskar Pelangi", "Fiksi"),
    Book("B1002", "Bumi Manusia", "Fiksi"),
    Book("B1003", "Filosofi Kopi", "Fiksi"),
    Book("B1004", "Sejarah Nusantara", "Sejarah"),
    Book("B1005", "Algoritma & Struktur Data", "Teknologi"),
    Book("B1006", "Cerita Rakyat Nusantara", "Anak-anak"),
    Book("B1007", "Fisika Dasar", "Sains"),
    Book("B1008", "Negeri 5 Menara", "Fiksi"),
]
stable, unstable = catalog[:], catalog[:]    # keduanya mulai dalam urutan id
insertion_sort(stable, key=lambda b: b.category)
selection_sort(unstable, key=lambda b: b.category)
print_books("By category with insertion sort (stable):", stable)
print_books("By category with selection sort (not stable):", unstable)
```

Kode 3.32 dan **Kode 3.33** mencetak hal yang sama dengan **Kode 3.31**:

```
By category with insertion sort (stable):
  Anak-anak  B1006  Cerita Rakyat Nusantara
  Fiksi      B1001  Laskar Pelangi
  Fiksi      B1002  Bumi Manusia
  Fiksi      B1003  Filosofi Kopi
  Fiksi      B1008  Negeri 5 Menara
  Sains      B1007  Fisika Dasar
  Sejarah    B1004  Sejarah Nusantara
  Teknologi  B1005  Algoritma & Struktur Data
By category with selection sort (not stable):
  Anak-anak  B1006  Cerita Rakyat Nusantara
  Fiksi      B1002  Bumi Manusia
  Fiksi      B1003  Filosofi Kopi
  Fiksi      B1001  Laskar Pelangi
  Fiksi      B1008  Negeri 5 Menara
  Sains      B1007  Fisika Dasar
  Sejarah    B1004  Sejarah Nusantara
  Teknologi  B1005  Algoritma & Struktur Data
```

Gambar 3.14 menempatkan ketiga urutan berdampingan. *Insertion sort* mempertahankan keempat buku *Fiksi* dalam urutan id. *Selection sort* tidak: pada putaran pertamanya ia menukar `B1001` (posisi 0) dengan `B1006` (satu-satunya buku *Anak-anak*, kategori terkecil), yang melemparkan `B1001` ke posisi 5, di belakang `B1002` dan `B1003`. Setiap buku masih berada dalam kelompok kategori yang benar; hanya urutan *di dalam* kelompok yang berubah. Bila pustakawan sebelumnya menata rak menurut id dan berharap urutan itu bertahan, pengurutan yang tidak stabil mematahkan harapan itu.



Gambar 3.14 Mengurutkan katalog menurut kategori. Hanya pengurutan stabil yang mempertahankan keempat buku *Fiksi* dalam urutan id aslinya; *selection sort* memindahkan B1001 ke belakang B1002 dan B1003.

Aturan praktisnya pun jelas: **untuk mengurutkan menurut beberapa kunci, urutkan menurut kunci yang paling tidak penting lebih dahulu dan menurut kunci yang paling penting terakhir, dengan pengurutan stabil setiap kali.** Untuk mendaftar buku menurut kategori dan, di dalam setiap kategori, menurut judul, urutkan menurut judul lalu secara stabil menurut kategori. *Radix sort* adalah aturan ini yang diterapkan pada digit. Pengurutan tidak stabil mana pun dapat dibuat stabil dengan menambahkan posisi asli sebagai kunci penentu terakhir, dengan biaya menyimpan posisi itu.

3.12.2 Tabel lengkapnya

Tabel 3.2 menghimpun semua yang telah diturunkan bab ini. "Adaptif" berarti biayanya turun pada masukan yang hampir terurut.

Tabel 3.2 Waktu jalan (setiap isian adalah Θ dari ungkapannya, kecuali rata-rata *Shell sort*, yang merupakan taksiran empiris), memori tambahan, dan ketiga sifat setiap pengurutan di bab ini.

Algoritma	Terbaik	Rata-rata	Terburuk	Ruang tambahan	Stabil	Di tempat	Adaptif
Bubble (keluar dini)	n	n^2	n^2	1	ya	ya	ya
Exchange	n^2	n^2	n^2	1	tidak	ya	tidak
Selection	n^2	n^2	n^2	1	tidak	ya	tidak
Insertion	n	n^2	n^2	1	ya	ya	ya
Shell (celah Knuth)	$n\lg n$	sekitar $n^{1.25}$	$n^{1.5}$	1	tidak	ya	ya
Merge	$n\lg n$	$n\lg n$	$n\lg n$	n	ya	tidak	tidak
Quick (pivot acak)	$n\lg n$	$n\lg n$	n^2	$\lg n$	tidak	ya	tidak
Heap	$n\lg n$	$n\lg n$	$n\lg n$	1	tidak	ya	tidak
Counting	$n+k$	$n+k$	$n+k$	$n+k$	ya	tidak	tidak
Radix (d digit, basis b)	$d(n+b)$	$d(n+b)$	$d(n+b)$	$n+b$	ya	tidak	tidak

Tiga isian perlu diberi catatan. Ruang tambahan *quicksort* adalah tumpukan rekursinya, $\lg n$ tingkat bila ia selalu berekursi ke sisi yang lebih kecil. Kasus rata-rata *Shell sort* tidak mempunyai rumus yang diketahui untuk barisan celah praktis mana pun; angka yang diberikan adalah taksiran empiris yang lazim [1], [9]. Dan kasus terbaik *heap sort* adalah $n\lg n$ hanya bila kuncinya berbeda-beda; bila semua kunci sama, setiap penenggelaman langsung berhenti dan waktunya linear.

Tabel 3.3 mendaftar apa yang sesungguhnya dikerjakan setiap algoritma pada keenam bilangan kita. Masukan yang sama, jawaban yang sama, dan kerja yang sangat berbeda.

Tabel 3.3 Kerja setiap algoritma pada $A = [22,10,15,3,8,2]$, sebagaimana dicetak program-program pada Subbab 3.3 sampai 3.11.

Algoritma	Perbandingan	Pertukaran atau pemindahan
Bubble sort	15	13 pertukaran
Exchange sort	15	13 pertukaran
Selection sort	15	3 pertukaran
Insertion sort	15	13 pergeseran
Shell sort (celah 3, 1)	11	7 pergeseran
Merge sort	9	16 pemindahan
Quicksort, Lomuto, pivot terakhir	13	3 pertukaran
Quicksort, Hoare, pivot tengah	24	3 pertukaran
Heap sort	13	9 pertukaran
Counting / radix sort	0	6 / 12 pemindahan

Hitungan 24 milik Hoare tampak tinggi hanya karena program kita menghitung setiap uji pada kedua gelung penyusurannya, termasuk uji yang menghentikannya; pada masukan besar skema Hoare melakukan perbandingan kira-kira sebanyak Lomuto dan pertukaran jauh lebih sedikit. Pada enam elemen, perbedaan semacam itu hanyalah derau. Untuk melihat algoritma mana yang benar-benar cepat, kita harus memakai data besar.

3.13 Mengukur Pengurutan

3.13.1 Program tolok ukur

Rumus meramalkan bagaimana kerja *bertumbuh*; hanya pengukuran yang menyatakan berapa lama kerja itu sesungguhnya *berlangsung*. **Kode 3.34** menjalankan setiap algoritma dalam bab ini pada bilangan acak yang sama untuk $n = 10^3, 10^4, 10^5$, dan 10^6, lalu melaporkan perbandingan, pertukaran atau pemindahan, dan waktu. Ketiga pengurutan kuadratik berhenti pada 10^5 elemen: pada 10^6, *bubble sort* akan memerlukan sekitar lima puluh menit. Dua baris terakhir setiap ukuran adalah pengurutan milik pustaka C++ sendiri, yang perbandingannya dihitung oleh komparator yang menaikkan sebuah pencacah; pustaka itu tidak memberi tahu berapa elemen yang dipindahkannya, sehingga kolom itu menampilkan tanda pisah. Program memeriksa bahwa setiap hasil benar-benar terurut, dan ditutup dengan mengukur waktu ketiga pengurutan kuadratik pada $n = 10^4$ untuk masukan acak, menaik, dan menurun.

Quicksort dalam program ini adalah bentuk produksi dari Subbab 3.9: pivot median dari tiga, dan sebuah gelung menggantikan pemanggilan rekursif kedua sehingga ia hanya berekursi ke sisi yang lebih kecil dan tumpukannya tetap di bawah lg$n$ tingkat. *Shell sort*-nya memakai celah Ciura.

Kode 3.34 `sort_benchmark.cpp`: setiap pengurutan dalam bab ini pada seribu sampai sejuta bilangan bulat acak, dengan hitungan dan waktu.

```cpp
// sort_benchmark.cpp - hitungan dan waktu setiap pengurutan di bab ini
#include <algorithm>
#include <chrono>
#include <cstdint>
#include <cstdio>
#include <string>
#include <utility>
#include <vector>
using namespace std;

// perbandingan, pertukaran atau pemindahan
struct Counts { long long cmp = 0, mov = 0; };

uint64_t state = 2026;
int nextRandom(int bound) {
    state = state * 6364136223846793005ULL + 1442695040888963407ULL;
    return static_cast<int>((state >> 33) % static_cast<uint64_t>(bound));
}

void bubble(vector<int>& a, Counts& c) {
    int n = static_cast<int>(a.size());
    for (int pass = 1; pass < n; ++pass) {
        bool swapped = false;
        for (int j = 0; j < n - pass; ++j) {
            ++c.cmp;
            if (a[j] > a[j + 1]) {
                swap(a[j], a[j + 1]); ++c.mov;
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}

void selection(vector<int>& a, Counts& c) {
    int n = static_cast<int>(a.size());
    for (int i = 0; i < n - 1; ++i) {
        int pos = i;
        for (int j = i + 1; j < n; ++j) { ++c.cmp; if (a[j] < a[pos]) pos = j; }
        if (pos != i) { swap(a[i], a[pos]); ++c.mov; }
    }
}
```

Kode 3.34 (Lanjutan) *sort_benchmark.cpp: setiap pengurutan dalam bab ini pada seribu sampai sejuta bilangan bulat acak, dengan hitungan dan waktu.*

```
void gappedInsertion(vector<int>& a, int gap, Counts& c) {
    int n = static_cast<int>(a.size());
    for (int i = gap; i < n; ++i) {
        int key = a[i], j = i - gap;
        while (j >= 0) {
            ++c.cmp;
            if (a[j] <= key) break;
            a[j + gap] = a[j]; ++c.mov;
            j -= gap;
        }
        a[j + gap] = key;
    }
}

void insertion(vector<int>& a, Counts& c) { gappedInsertion(a, 1, c); }

// celah Ciura, diperpanjang dengan x2,25
void shell(vector<int>& a, Counts& c) {
    vector<int> gaps = {1, 4, 10, 23, 57, 132, 301, 701, 1750};
    while (gaps.back() * 9 / 4 < static_cast<int>(a.size()))
        gaps.push_back(gaps.back() * 9 / 4);
    for (int k = static_cast<int>(gaps.size()) - 1; k >= 0; --k)
        if (gaps[k] < static_cast<int>(a.size()))
            gappedInsertion(a, gaps[k], c);
}

void mergeRec(vector<int>& a, vector<int>& t, int lo, int hi, Counts& c) {
    if (hi - lo < 2) return;
    int mid = lo + (hi - lo) / 2;
    mergeRec(a, t, lo, mid, c);
    mergeRec(a, t, mid, hi, c);
    int i = lo, j = mid, k = lo;
    while (i < mid && j < hi) {
        ++c.cmp;
        t[k++] = (a[i] <= a[j]) ? a[i++] : a[j++];
    }
    while (i < mid) t[k++] = a[i++];
    while (j < hi) t[k++] = a[j++];
    for (k = lo; k < hi; ++k) { a[k] = t[k]; ++c.mov; }
}

void mergeSort(vector<int>& a, Counts& c) {
    vector<int> t(a.size());
    mergeRec(a, t, 0, static_cast<int>(a.size()), c);
}
```

Kode 3.34 (Lanjutan) *sort_benchmark.cpp: setiap pengurutan dalam bab ini pada seribu sampai sejuta bilangan bulat acak, dengan hitungan dan waktu.*

```
void quickRec(vector<int>& a, int lo, int hi, Counts& c) {
    while (lo < hi) {
        // median dari tiga ke a[hi]
        int mid = lo + (hi - lo) / 2;
        if (a[mid] < a[lo]) swap(a[mid], a[lo]);
        if (a[hi] < a[lo]) swap(a[hi], a[lo]);
        if (a[mid] < a[hi]) swap(a[mid], a[hi]);
        c.cmp += 3;
        int pivot = a[hi], i = lo - 1;
        for (int j = lo; j < hi; ++j) {
            ++c.cmp;
            if (a[j] <= pivot) {
                ++i;
                if (i != j) { swap(a[i], a[j]); ++c.mov; }
            }
        }
        swap(a[i + 1], a[hi]); ++c.mov;
        int p = i + 1;
        // rekursi ke sisi yang lebih kecil, gelung untuk sisi yang lebih besar
        if (p - lo < hi - p) { quickRec(a, lo, p - 1, c); lo = p + 1; }
        else                 { quickRec(a, p + 1, hi, c); hi = p - 1; }
    }
}

void quick(vector<int>& a, Counts& c) {
    quickRec(a, 0, static_cast<int>(a.size()) - 1, c);
}

void siftDown(vector<int>& a, int i, int size, Counts& c) {
    while (true) {
        int largest = i, l = 2 * i + 1, r = l + 1;
        if (l < size) { ++c.cmp; if (a[l] > a[largest]) largest = l; }
        if (r < size) { ++c.cmp; if (a[r] > a[largest]) largest = r; }
        if (largest == i) return;
        swap(a[i], a[largest]); ++c.mov;
        i = largest;
    }
}
```

Kode 3.34 (Lanjutan) *sort_benchmark.cpp: setiap pengurutan dalam bab ini pada seribu sampai sejuta bilangan bulat acak, dengan hitungan dan waktu.*

```
void heap(vector<int>& a, Counts& c) {
    int n = static_cast<int>(a.size());
    for (int i = n / 2 - 1; i >= 0; --i) siftDown(a, i, n, c);
    for (int end = n - 1; end > 0; --end) {
        swap(a[0], a[end]); ++c.mov;
        siftDown(a, 0, end, c);
    }
}

// basis 256, empat lintasan, tanpa perbandingan
void radix(vector<int>& a, Counts& c) {
    vector<int> out(a.size());
    for (int shift = 0; shift < 32; shift += 8) {
        vector<size_t> count(257, 0);
        for (int x : a) ++count[((x >> shift) & 255) + 1];
        for (int v = 0; v < 256; ++v) count[v + 1] += count[v];
        for (int x : a) { out[count[(x >> shift) & 255]++] = x; ++c.mov; }
        a.swap(out);
    }
}

void stdSort(vector<int>& a, Counts& c) {  // pemindahan tak terlihat: -1
    c.mov = -1;
    sort(a.begin(), a.end(), [&c](int x, int y) { ++c.cmp; return x < y; });
}

void stdStable(vector<int>& a, Counts& c) {
    c.mov = -1;
    stable_sort(a.begin(), a.end(),
                [&c](int x, int y) { ++c.cmp; return x < y; });
}

string cell(long long v) { return v < 0 ? "-" : to_string(v); }

double run(void (*sortFn)(vector<int>&, Counts&), vector<int> a, Counts& c) {
    auto t0 = chrono::steady_clock::now();
    sortFn(a, c);
    auto t1 = chrono::steady_clock::now();
    if (!is_sorted(a.begin(), a.end())) printf("  NOT SORTED!\n");
    return chrono::duration<double, milli>(t1 - t0).count();
}
```

Kode 3.34 (Lanjutan) *sort_benchmark.cpp: setiap pengurutan dalam bab ini pada seribu sampai sejuta bilangan bulat acak, dengan hitungan dan waktu.*

```
int main() {
    struct Algo {
        const char* name;
        void (*fn)(vector<int>&, Counts&);
        int maxN;
    };
    Algo algos[] = {
        {"bubble", bubble, 100000},     {"selection", selection, 100000},
        {"insertion", insertion, 100000}, {"shell", shell, 1000000},
        {"merge", mergeSort, 1000000},  {"quick", quick, 1000000},
        {"heap", heap, 1000000},        {"radix", radix, 1000000},
        {"std::sort", stdSort, 1000000},
        {"std::stable_sort", stdStable, 1000000},
    };
    printf("%-17s %8s %15s %15s %11s\n", "algorithm", "n", "comparisons",
           "swaps/moves", "time ms");
    for (int n = 1000; n <= 1000000; n *= 10) {
        state = 2026;
        vector<int> data(n);
        for (int& x : data) x = nextRandom(1000000000);
        for (const Algo& al : algos) {
            if (n > al.maxN) continue;
            Counts c;
            double ms = run(al.fn, data, c);
            printf("%-17s %8d %15s %15s %11.2f\n", al.name, n,
                   cell(c.cmp).c_str(), cell(c.mov).c_str(), ms);
        }
    }
    printf("\nn = 10000, time ms by input order\n");
    printf("%-17s %11s %11s %11s\n", "algorithm", "random", "ascending",
           "descending");
    int n = 10000;
    state = 2026;
    vector<int> rnd(n), asc(n), desc(n);
    for (int i = 0; i < n; ++i) {
        rnd[i] = nextRandom(1000000000); asc[i] = i; desc[i] = n - i;
    }
    for (int k = 0; k < 3; ++k) {
        Counts c1, c2, c3;
        double t1 = run(algos[k].fn, rnd, c1), t2 = run(algos[k].fn, asc, c2),
               t3 = run(algos[k].fn, desc, c3);
        printf("%-17s %11.2f %11.2f %11.2f\n", algos[k].name, t1, t2, t3);
    }
    return 0;
}
```

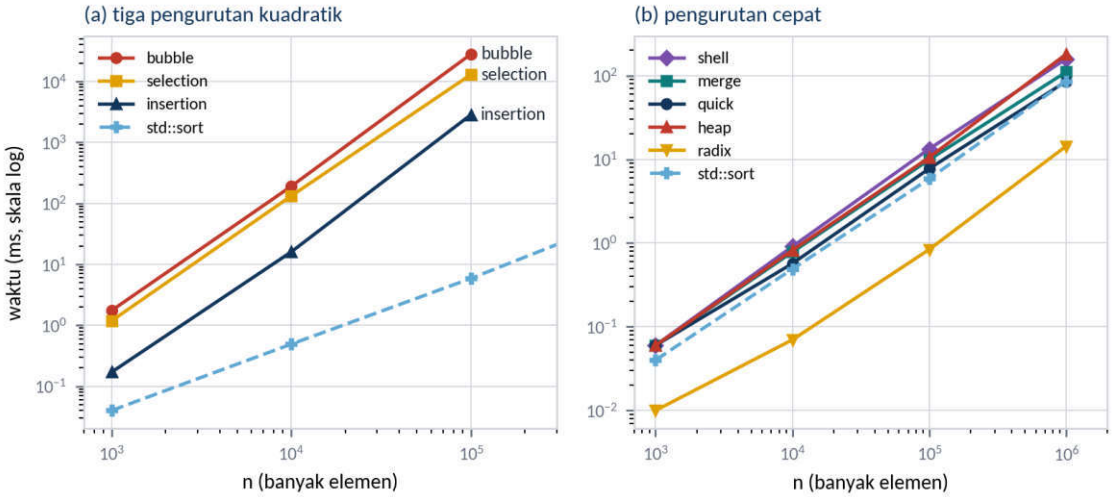
Pengukuran di bawah ini diambil pada server awan bersama dengan prosesor Intel Xeon 2,1 GHz, memakai satu inti, dengan `g++` 13.3 pada `-O2`, OpenJDK 21, dan Python 3.11. Waktu pada komputer Anda akan berbeda, dan bahkan pada mesin ini jalan kedua berbeda sepuluh sampai dua puluh persen; sebaliknya, hitungannya identik pada setiap mesin dan dalam setiap bahasa, karena setiap versi mengurutkan bilangan yang sama dari pembangkit yang sama.

```
algorithm               n     comparisons     swaps/moves     time ms
bubble               1000          499290          252221        1.74
selection            1000          499500             987        1.18
insertion            1000          253216          252221        0.17
shell                1000           13187            6893        0.06
merge                1000            8723            9976        0.06
quick                1000           10666            3682        0.06
heap                 1000           16846            9095        0.06
radix                1000               0            4000        0.01
std::sort            1000           11870               -        0.04
std::stable_sort     1000            9864               -        0.05
bubble              10000        49994994        24649368      190.22
selection           10000        49995000            9982      132.19
insertion           10000        24659362        24649368       15.98
shell               10000          190702          101054        0.91
merge               10000          120445          133616        0.78
quick               10000          151336           62176        0.57
heap                10000          235396          124322        0.82
radix               10000               0           40000        0.07
std::sort           10000          156934               -        0.49
std::stable_sort    10000          128101               -        0.57
bubble             100000      4999861169      2500232977    28013.78
selection          100000      4999950000           99988    12961.01
insertion          100000      2500332969      2500232977     2841.53
shell              100000         2554967         1384924       13.34
merge              100000         1536335         1668928        9.91
quick              100000         1881706          828252        7.76
heap               100000         3019684         1574735       10.56
radix              100000               0          400000        0.84
std::sort          100000         1956930               -        5.87
std::stable_sort   100000         1595435               -        7.58
shell             1000000        32014545        17456447      158.35
merge             1000000        18675220        19951424      110.50
quick             1000000        22507428        10026671       85.28
heap              1000000        36794517        19047984      180.60
radix             1000000               0         4000000       14.37
std::sort         1000000        23732145               -       85.77
std::stable_sort  1000000        19824445               -      125.10

n = 10000, time ms by input order
algorithm              random   ascending  descending
bubble                265.30        0.01      337.10
selection             124.65      138.71      128.45
insertion              17.85        0.01       35.96
```

3.13.2 Membaca angka-angkanya

Gambar 3.15 memplot waktu-waktu itu pada sumbu logaritmik, tempat hukum pangkat $t = c\,n^p$ menjadi garis lurus dengan kemiringan p: kemiringan 2 untuk algoritma kuadratik, kemiringan sedikit di atas 1 untuk $n\lg n$.



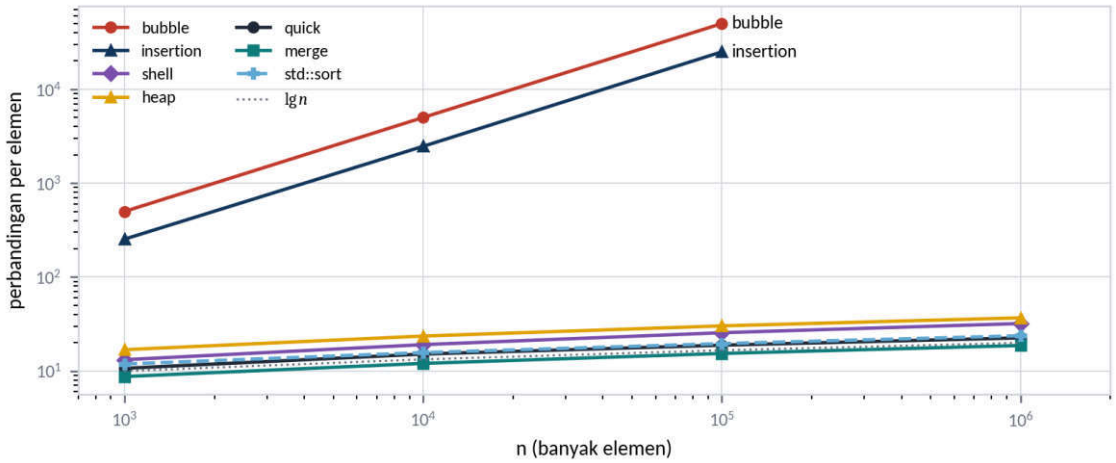
Gambar 3.15 Waktu terukur program-program C++, pada sumbu log–log. (a) Ketiga pengurutan kuadratik naik dua dekade waktu untuk setiap dekade n; `std::sort` ditampilkan sebagai pembanding skala. (b) Pengurutan cepat hanya naik sedikit lebih dari satu dekade per dekade; *radix sort* yang tercepat.

Pengurutan kuadratik berperilaku persis seperti diramalkan Persamaan (3.3)**.** Perbandingan *bubble sort* adalah 499,290, 49,994,994, dan 4,999,861,169: masing-masing hampir tepat $n(n-1)/2$, sedikit kurang karena keluar dini menghemat beberapa lintasan terakhir. Pertukarannya, 252,221 untuk $n = 1000$, adalah banyaknya inversi bilangan-bilangan acak itu, sangat dekat dengan $n(n-1)/4 = 249{,}750$ dari Persamaan (3.5). Pergeseran *insertion sort identik* dengan pertukaran *bubble sort* pada setiap ukuran, sebagaimana diharuskan oleh Persamaan (3.4) dan Subbab 3.6.4: keduanya sama dengan banyaknya inversi. *Selection sort* melakukan 25,000 kali lebih sedikit pertukaran daripada *bubble sort* pada $n = 10^5$ dan tetap lambat, karena perbandingannya tetap lima miliar yang sama.

Waktu tumbuh lebih cepat daripada hitungan. Dari 10^4 ke 10^5 hitungan tumbuh tepat 100 kali, tetapi waktu *bubble sort* tumbuh dari 190 ms menjadi 28 s, sekitar 147 kali, dan waktu *insertion sort* sekitar 178 kali. Faktor tambahan itu berasal dari mesin, bukan dari algoritma: 10^4 bilangan bulat (40 KB) muat di *cache* tercepat prosesor, 10^5 bilangan bulat (400 KB) tidak, dan setiap akses memori menjadi lebih mahal. *Insertion sort* tetap sepuluh kali lebih cepat daripada *bubble sort* dengan banyak pemindahan yang sama, karena satu pergeseran adalah satu operasi tulis sedangkan satu pertukaran adalah tiga, dan karena gelung dalamnya berhenti begitu kunci sudah ditempatkan.

Pengurutan $n\lg n$ tumbuh landai. *Merge sort* memerlukan 0.78 ms, 9.9 ms, dan 110 ms untuk 10^4, 10^5, dan 10^6 elemen, faktor 12 sampai 13 per kenaikan sepuluh kali, sedangkan $n\lg n$ meramalkan $10 \times \lg(10^5)/\lg(10^4) = 12.5$ dan $10 \times \lg(10^6)/\lg(10^5) = 12$. Pada sejuta elemen, ketika *bubble sort* akan memerlukan hampir satu jam, setiap pengurutan $n\lg n$ selesai dalam waktu kurang dari seperlima detik.

Perbandingan bukan keseluruhan cerita. Gambar 3.16 membagi perbandingan dengan n. Untuk pengurutan $n\lg n$ hasilnya tumbuh seperti $\lg n$ (garis bertitik); untuk yang kuadratik hasilnya tumbuh seperti n.



Gambar 3.16 Perbandingan per elemen. Pengurutan kuadratik menanjak curam; yang lain tetap dekat dengan lg*n*, dengan *merge sort* paling hemat dan *heap sort* paling boros.

Merge sort melakukan perbandingan paling sedikit dari semuanya (18.7 juta pada $n = 10^6$, sedikit di bawah $n\lg n = 19.9$ juta), dan *heap sort* paling banyak (36.8 juta, hampir dua kali lipat). Namun pengurutan berbasis perbandingan tercepat pada 10^6 adalah *quicksort* dan `std::sort`, sekitar 85 ms, di depan *merge sort* dengan 110 ms dan *heap sort* dengan 181 ms. *Quicksort* melakukan lebih banyak perbandingan daripada *merge sort* tetapi memindahkan data separuhnya, tidak memerlukan penyangga, dan menyusuri memori dari kiri ke kanan, yang dihargai oleh *cache*; *heap sort* melompat dari i ke $2i + 1$ dan kalah dari keduanya. *Radix sort*, yang sama sekali tidak membandingkan, mengurutkan sejuta bilangan bulat dalam 14 ms, enam kali lebih cepat daripada `std::sort`. Keterbatasannya bukan kecepatan melainkan keumuman: ia hanya mengurutkan hal-hal yang dapat dipotong menjadi digit.

3.13.3 Urutan masukan berpengaruh pada sebagian pengurutan

Tabel terakhir pada keluaran itu mengulang, pada mesin masa kini, pengukuran yang telah bertahun-tahun dipakai untuk mengajarkan bab ini. **Tabel 3.4** adalah versi lamanya: 10,000 bilangan pada prosesor Pentium II 450 MHz, komputer desktop yang lazim pada tahun 1998.

Tabel 3.4 Pengukuran historis: mengurutkan 10.000 bilangan pada Pentium II 450 MHz (akhir 1990-an). Waktu menurun *selection sort* mengulang waktu *bubble sort* dan hampir pasti merupakan salah salin; waktu *selection sort* tidak bergantung pada urutan masukan.

Algoritma	Acak	Menaik	Menurun
Bubble sort	11,20 s	1,32 s	19,77 s
Insertion sort	1,09 s	0,00 s	2,25 s
Selection sort	1,32 s	1,32 s	19,77 s

Polanya tetap sama hari ini, dalam C++ pada satu inti, seperti diperlihatkan **Tabel 3.5**:

Tabel 3.5 Eksperimen yang sama pada tahun 2026 (dari keluaran **Kode 3.34**). Waktu absolut turun dengan faktor 10 sampai 60; hubungan antaralgoritmanya tidak berubah.

Algoritma	Acak	Menaik	Menurun
Bubble sort	265 ms	0,01 ms	337 ms
Insertion sort	17,9 ms	0,01 ms	36,0 ms
Selection sort	125 ms	139 ms	128 ms

Bubble sort dan *insertion sort* bersifat adaptif: pada masukan menaik, keluar dini *bubble sort* berhenti setelah satu lintasan, dan gelung dalam *insertion sort* tidak pernah berjalan, sehingga

keduanya hanya memerlukan sekitar sepuluh mikrodetik. Pada masukan menurun keduanya mengerjakan kerja maksimum, dan kasus acak *insertion sort* yang dua kali lebih cepat dijelaskan oleh Persamaan (3.5): masukan acak mempunyai separuh inversi masukan terbalik. *Selection sort* memerlukan waktu yang sama apa pun urutannya. (*Bubble sort* lama dengan 1.32 s pada masukan terurut memperlihatkan bahwa ia tidak mempunyai keluar dini.) Dalam seperempat abad perangkat keras membuat gelung-gelung ini sepuluh sampai enam puluh kali lebih cepat; perbedaan antara algoritma kuadratik dan $n \lg n$ pada sejuta elemen adalah faktor *dua puluh lima ribu* (**Tabel 3.1**). Algoritma yang lebih baik mengalahkan perangkat keras yang lebih baik.

3.13.4 Tolok ukur yang sama dalam Java dan Python

Kode 3.35 adalah versi Java-nya. Sebagai pengganti `std::sort` ia mengukur dua pengurutan pustaka yang ditawarkan Java: `Arrays.sort` pada `int[]`, dan `Arrays.sort` pada larik objek `Integer` dengan komparator penghitung.

Kode 3.35 `SortBenchmark.java`: tolok ukur dalam Java, termasuk `Arrays.sort` pada tipe primitif dan pada objek.

```java
// SortBenchmark.java - hitungan dan waktu setiap pengurutan di bab ini
import java.util.Arrays;

public class SortBenchmark {
    static long cmp, mov;  // perbandingan, pertukaran atau pemindahan

    static long state = 2026;
    static int nextRandom(int bound) {
        state = state * 6364136223846793005L + 1442695040888963407L;
        return (int) ((state >>> 33) % bound);
    }

    static void swap(int[] a, int i, int j) {
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }

    static void bubble(int[] a) {
        int n = a.length;
        for (int pass = 1; pass < n; pass++) {
            boolean swapped = false;
            for (int j = 0; j < n - pass; j++) {
                cmp++;
                if (a[j] > a[j + 1]) {
                    swap(a, j, j + 1); mov++;
                    swapped = true;
                }
            }
            if (!swapped) break;
        }
    }

    static void selection(int[] a) {
        int n = a.length;
        for (int i = 0; i < n - 1; i++) {
            int pos = i;
            for (int j = i + 1; j < n; j++) {
                cmp++;
                if (a[j] < a[pos]) pos = j;
            }
            if (pos != i) { swap(a, i, pos); mov++; }
        }
    }
```

Kode 3.35 (Lanjutan) *SortBenchmark.java: tolok ukur dalam Java, termasuk Arrays.sort pada tipe primitif dan pada objek.*

```
    static void gappedInsertion(int[] a, int gap) {
        for (int i = gap; i < a.length; i++) {
            int key = a[i], j = i - gap;
            while (j >= 0) {
                cmp++;
                if (a[j] <= key) break;
                a[j + gap] = a[j]; mov++;
                j -= gap;
            }
            a[j + gap] = key;
        }
    }

    static void insertion(int[] a) { gappedInsertion(a, 1); }

    static void shell(int[] a) {  // celah Ciura, diperpanjang dengan x2,25
        // ruang untuk celah tambahan
        int[] gaps = new int[20];
        int[] ciura = {1, 4, 10, 23, 57, 132, 301, 701, 1750};
        System.arraycopy(ciura, 0, gaps, 0, ciura.length);
        int count = 9;
        while (gaps[count - 1] * 9 / 4 < a.length) {
            gaps[count] = gaps[count - 1] * 9 / 4;
            count++;
        }
        for (int k = count - 1; k >= 0; k--)
            if (gaps[k] < a.length) gappedInsertion(a, gaps[k]);
    }

    static void mergeRec(int[] a, int[] t, int lo, int hi) {
        if (hi - lo < 2) return;
        int mid = lo + (hi - lo) / 2;
        mergeRec(a, t, lo, mid);
        mergeRec(a, t, mid, hi);
        int i = lo, j = mid, k = lo;
        while (i < mid && j < hi) {
            cmp++;
            t[k++] = (a[i] <= a[j]) ? a[i++] : a[j++];
        }
        while (i < mid) t[k++] = a[i++];
        while (j < hi) t[k++] = a[j++];
        for (k = lo; k < hi; k++) { a[k] = t[k]; mov++; }
    }
```

Kode 3.35 (Lanjutan) *SortBenchmark.java: tolok ukur dalam Java, termasuk Arrays.sort pada tipe primitif dan pada objek.*

```
    static void mergeSort(int[] a) {
        mergeRec(a, new int[a.length], 0, a.length);
    }

    static void quickRec(int[] a, int lo, int hi) {
        while (lo < hi) {
            // median dari tiga ke a[hi]
            int mid = lo + (hi - lo) / 2;
            if (a[mid] < a[lo]) swap(a, mid, lo);
            if (a[hi] < a[lo]) swap(a, hi, lo);
            if (a[mid] < a[hi]) swap(a, mid, hi);
            cmp += 3;
            int pivot = a[hi], i = lo - 1;
            for (int j = lo; j < hi; j++) {
                cmp++;
                if (a[j] <= pivot) {
                    i++;
                    if (i != j) { swap(a, i, j); mov++; }
                }
            }
            swap(a, i + 1, hi); mov++;
            int p = i + 1;
            // rekursi ke sisi yang lebih kecil, gelung untuk sisi yang lebih
            // besar
            if (p - lo < hi - p) { quickRec(a, lo, p - 1); lo = p + 1; }
            else                 { quickRec(a, p + 1, hi); hi = p - 1; }
        }
    }

    static void quick(int[] a) { quickRec(a, 0, a.length - 1); }

    static void siftDown(int[] a, int i, int size) {
        while (true) {
            int largest = i, l = 2 * i + 1, r = l + 1;
            if (l < size) { cmp++; if (a[l] > a[largest]) largest = l; }
            if (r < size) { cmp++; if (a[r] > a[largest]) largest = r; }
            if (largest == i) return;
            swap(a, i, largest); mov++;
            i = largest;
        }
    }
```

Kode 3.35 (Lanjutan) *SortBenchmark.java: tolok ukur dalam Java, termasuk Arrays.sort pada tipe primitif dan pada objek.*

```
    static void heap(int[] a) {
        int n = a.length;
        for (int i = n / 2 - 1; i >= 0; i--) siftDown(a, i, n);
        for (int end = n - 1; end > 0; end--) {
            swap(a, 0, end); mov++;
            siftDown(a, 0, end);
        }
    }

    // basis 256, empat lintasan, tanpa perbandingan
    static void radix(int[] a) {
        int[] out = new int[a.length];
        for (int shift = 0; shift < 32; shift += 8) {
            int[] count = new int[257];
            for (int x : a) count[((x >> shift) & 255) + 1]++;
            for (int v = 0; v < 256; v++) count[v + 1] += count[v];
            for (int x : a) { out[count[(x >> shift) & 255]++] = x; mov++; }
            System.arraycopy(out, 0, a, 0, a.length);
        }
    }

    static void arraysSortInt(int[] a) { cmp = -1; mov = -1; Arrays.sort(a); }

    static void arraysSortObj(int[] a) {  // TimSort pada objek Integer
        Integer[] boxed = new Integer[a.length];
        for (int i = 0; i < a.length; i++) boxed[i] = a[i];
        mov = -1;
        Arrays.sort(boxed, (x, y) -> { cmp++; return Integer.compare(x, y); });
        for (int i = 0; i < a.length; i++) a[i] = boxed[i];
    }

    interface Sorter { void sort(int[] a); }
    record Algo(String name, Sorter fn, int maxN) { }

    static String cell(long v) { return v < 0 ? "-" : Long.toString(v); }
```

Kode 3.35 (Lanjutan) *SortBenchmark.java: tolok ukur dalam Java, termasuk Arrays.sort pada tipe primitif dan pada objek.*

```
    static double run(Sorter fn, int[] data) {
        int[] a = data.clone();
        cmp = 0; mov = 0;
        long t0 = System.nanoTime();
        fn.sort(a);
        long t1 = System.nanoTime();
        for (int i = 1; i < a.length; i++)
            if (a[i - 1] > a[i]) { System.out.println("  NOT SORTED!"); break; }
        return (t1 - t0) / 1e6;
    }

    public static void main(String[] args) {
        Algo[] algos = {
            new Algo("bubble", SortBenchmark::bubble, 100000),
            new Algo("selection", SortBenchmark::selection, 100000),
            new Algo("insertion", SortBenchmark::insertion, 100000),
            new Algo("shell", SortBenchmark::shell, 1000000),
            new Algo("merge", SortBenchmark::mergeSort, 1000000),
            new Algo("quick", SortBenchmark::quick, 1000000),
            new Algo("heap", SortBenchmark::heap, 1000000),
            new Algo("radix", SortBenchmark::radix, 1000000),
            new Algo("Arrays.sort int", SortBenchmark::arraysSortInt, 1000000),
            new Algo("Arrays.sort Obj", SortBenchmark::arraysSortObj, 1000000),
        };
        System.out.printf("%-17s %8s %15s %15s %11s%n", "algorithm", "n",
                "comparisons", "swaps/moves", "time ms");
        for (int n = 1000; n <= 1000000; n *= 10) {
            state = 2026;
            int[] data = new int[n];
            for (int i = 0; i < n; i++) data[i] = nextRandom(1000000000);
            for (Algo al : algos) {
                if (n > al.maxN()) continue;
                double ms = run(al.fn(), data);
                System.out.printf("%-17s %8d %15s %15s %11.2f%n",
                        al.name(), n, cell(cmp), cell(mov), ms);
            }
        }
        System.out.printf("%nn = 10000, time ms by input order%n");
        System.out.printf("%-17s %11s %11s %11s%n", "algorithm", "random",
                "ascending", "descending");
        int n = 10000;
        state = 2026;
        int[] rnd = new int[n], asc = new int[n], desc = new int[n];
        for (int i = 0; i < n; i++) {
            rnd[i] = nextRandom(1000000000); asc[i] = i; desc[i] = n - i;
        }
```

Kode 3.35 (Lanjutan) *SortBenchmark.java: tolok ukur dalam Java, termasuk Arrays.sort pada tipe primitif dan pada objek.*

```
        for (int k = 0; k < 3; k++) {
            double t1 = run(algos[k].fn(), rnd), t2 = run(algos[k].fn(), asc),
                   t3 = run(algos[k].fn(), desc);
            System.out.printf("%-17s %11.2f %11.2f %11.2f%n",
                    algos[k].name(), t1, t2, t3);
        }
    }
}
```

```
algorithm                n      comparisons     swaps/moves     time ms
bubble                1000           499290          252221        9.23
selection             1000           499500             987        6.98
insertion             1000           253216          252221        8.10
shell                 1000            13187            6893        0.10
merge                 1000             8723            9976        4.23
quick                 1000            10666            3682        0.50
heap                  1000            16846            9095        0.85
radix                 1000                0            4000        0.23
Arrays.sort int       1000                -               -        4.75
Arrays.sort Obj       1000             8690               -        2.02
bubble               10000         49994994        24649368      186.51
selection            10000         49995000            9982       77.25
insertion            10000         24659362        24649368       32.82
shell                10000           190702          101054        1.97
merge                10000           120445          133616        1.23
quick                10000           151336           62176        5.62
heap                 10000           235396          124322        2.12
radix                10000                0           40000        9.73
Arrays.sort int      10000                -               -       13.57
Arrays.sort Obj      10000           120420               -       32.26
bubble              100000       4999861169      2500232977    19339.70
selection           100000       4999950000           99988     3744.82
insertion           100000       2500332969      2500232977     3764.32
shell               100000          2554967         1384924       11.34
merge               100000          1536335         1668928       13.89
quick               100000          1881706          828252        9.06
heap                100000          3019684         1574735       16.62
radix               100000                0          400000        9.32
Arrays.sort int     100000                -               -       15.90
Arrays.sort Obj     100000          1535047               -       67.50
shell              1000000         32014545        17456447      207.64
merge              1000000         18675220        19951424      141.01
quick              1000000         22507428        10026671       95.51
heap               1000000         36794517        19047984      199.82
radix              1000000                0         4000000       20.43
Arrays.sort int    1000000                -               -      226.85
Arrays.sort Obj    1000000         18640977               -      734.53

n = 10000, time ms by input order
algorithm              random   ascending  descending
bubble                 120.72        0.01       82.30
selection               23.93       25.11       28.27
insertion               48.96        0.03       74.02
```

Hitungannya identik dengan hitungan C++, digit demi digit. Waktunya menyampaikan dua pelajaran tentang Java. Pertama, pengurutan buatan kita berjalan hampir secepat C++ begitu kompilator *just-in-time* (JIT) sudah menerjemahkannya menjadi kode mesin, tetapi pengukuran awal yang kecil terdistorsi oleh kompilasi itu, sehingga waktu untuk $n = 1000$ tidak menentu. Kedua, pemanggilan pustaka `Arrays.sort` hanya diukur sekali per ukuran, dalam keadaan dingin, dan satu pemanggilan dingin adalah pengukuran yang buruk; pengukuran waktu Java yang andal memerlukan perangkat seperti JMH yang menjalankan kode itu berkali-kali lebih dahulu. Mengurutkan objek `Integer` beberapa kali lebih lambat daripada mengurutkan `int`, karena setiap elemen adalah objek terpisah yang dicapai melalui referensi, dan membandingkan dua di antaranya berarti mengikuti dua referensi. Satu keanehan: *bubble sort* Java *lebih cepat* pada masukan menurun daripada pada masukan acak. Pada masukan menurun setiap perbandingan berkata "tukar", prosesor meramalkannya dengan sempurna, dan tidak pernah harus membuang kerja yang telah dimulainya secara spekulatif; pada masukan acak tebakannya salah separuh waktu.

Kode 3.36 adalah versi Python-nya. Python murni mengeksekusi setiap langkah dasar melalui interpreter, kira-kira dua puluh sampai delapan puluh kali lebih lambat daripada kode terkompilasi, sehingga pengurutan kuadratik berhenti pada $n = 10^4$.

Kode 3.36 `sort_benchmark.py`: tolok ukur dalam Python; pengurutan kuadratik berhenti pada sepuluh ribu elemen.

```
# sort_benchmark.py - hitungan dan waktu setiap pengurutan di bab ini
import time

# perbandingan, pertukaran atau pemindahan
cmp = mov = 0
state = 2026


def next_random(bound):
    global state
    state = (state * 6364136223846793005 + 1442695040888963407) % 2**64
    return (state >> 33) % bound


def bubble(a):
    global cmp, mov
    n = len(a)
    for pass_no in range(1, n):
        swapped = False
        for j in range(n - pass_no):
            cmp += 1
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
                mov += 1
                swapped = True
        if not swapped:
            break


def selection(a):
    global cmp, mov
    n = len(a)
    for i in range(n - 1):
        pos = i
        for j in range(i + 1, n):
            cmp += 1
            if a[j] < a[pos]:
                pos = j
        if pos != i:
            a[i], a[pos] = a[pos], a[i]
            mov += 1
```

Kode 3.36 (Lanjutan) *sort_benchmark.py: tolok ukur dalam Python; pengurutan kuadratik berhenti pada sepuluh ribu elemen.*

```
def gapped_insertion(a, gap):
    global cmp, mov
    for i in range(gap, len(a)):
        key, j = a[i], i - gap
        while j >= 0:
            cmp += 1
            if a[j] <= key:
                break
            a[j + gap] = a[j]
            mov += 1
            j -= gap
        a[j + gap] = key


def insertion(a):
    gapped_insertion(a, 1)


# celah Ciura, diperpanjang dengan x2,25
def shell(a):
    gaps = [1, 4, 10, 23, 57, 132, 301, 701, 1750]
    while gaps[-1] * 9 // 4 < len(a):
        gaps.append(gaps[-1] * 9 // 4)
    for gap in reversed(gaps):
        if gap < len(a):
            gapped_insertion(a, gap)


def merge_rec(a, t, lo, hi):
    global cmp, mov
    if hi - lo < 2:
        return
    mid = lo + (hi - lo) // 2
    merge_rec(a, t, lo, mid)
    merge_rec(a, t, mid, hi)
    i, j, k = lo, mid, lo
    while i < mid and j < hi:
        cmp += 1
        if a[i] <= a[j]:
            t[k] = a[i]
            i += 1
        else:
            t[k] = a[j]
            j += 1
        k += 1
    t[k:hi] = a[i:mid] + a[j:hi]
```

Kode 3.36 (Lanjutan) *sort_benchmark.py: tolok ukur dalam Python; pengurutan kuadratik berhenti pada sepuluh ribu elemen.*

```
    a[lo:hi] = t[lo:hi]
    mov += hi - lo


def merge_sort(a):
    merge_rec(a, [0] * len(a), 0, len(a))


def quick_rec(a, lo, hi):
    global cmp, mov
    while lo < hi:
        mid = lo + (hi - lo) // 2        # median dari tiga ke a[hi]
        if a[mid] < a[lo]:
            a[mid], a[lo] = a[lo], a[mid]
        if a[hi] < a[lo]:
            a[hi], a[lo] = a[lo], a[hi]
        if a[mid] < a[hi]:
            a[mid], a[hi] = a[hi], a[mid]
        cmp += 3
        pivot, i = a[hi], lo - 1
        for j in range(lo, hi):
            cmp += 1
            if a[j] <= pivot:
                i += 1
                if i != j:
                    a[i], a[j] = a[j], a[i]
                    mov += 1
        a[i + 1], a[hi] = a[hi], a[i + 1]
        mov += 1
        p = i + 1
        if p - lo < hi - p:              # rekursi pada sisi yang lebih kecil
            quick_rec(a, lo, p - 1)
            lo = p + 1
        else:
            quick_rec(a, p + 1, hi)
            hi = p - 1


def quick(a):
    quick_rec(a, 0, len(a) - 1)
```

Kode 3.36 (Lanjutan) *sort_benchmark.py: tolok ukur dalam Python; pengurutan kuadratik berhenti pada sepuluh ribu elemen.*

```
def sift_down(a, i, size):
    global cmp, mov
    while True:
        largest, l, r = i, 2 * i + 1, 2 * i + 2
        if l < size:
            cmp += 1
            if a[l] > a[largest]:
                largest = l
        if r < size:
            cmp += 1
            if a[r] > a[largest]:
                largest = r
        if largest == i:
            return
        a[i], a[largest] = a[largest], a[i]
        mov += 1
        i = largest


def heap(a):
    global mov
    n = len(a)
    for i in range(n // 2 - 1, -1, -1):
        sift_down(a, i, n)
    for end in range(n - 1, 0, -1):
        a[0], a[end] = a[end], a[0]
        mov += 1
        sift_down(a, 0, end)


# basis 256, empat lintasan, tanpa perbandingan
def radix(a):
    global mov
    for shift in (0, 8, 16, 24):
        count = [0] * 257
        for x in a:
            count[((x >> shift) & 255) + 1] += 1
        for v in range(256):
            count[v + 1] += count[v]
        out = [0] * len(a)
        for x in a:
            d = (x >> shift) & 255
            out[count[d]] = x
            count[d] += 1
            mov += 1
        a[:] = out
```

Kode 3.36 (Lanjutan) *sort_benchmark.py: tolok ukur dalam Python; pengurutan kuadratik berhenti pada sepuluh ribu elemen.*

```
def list_sort(a):  # Timsort dalam C: tak ada yang bisa dihitung dari sini
    global cmp, mov
    cmp = mov = -1
    a.sort()


def cell(v):
    return "-" if v < 0 else str(v)


def run(fn, data):
    global cmp, mov
    a = data[:]
    cmp = mov = 0
    t0 = time.perf_counter()
    fn(a)
    t1 = time.perf_counter()
    if any(a[i - 1] > a[i] for i in range(1, len(a))):
        print("  NOT SORTED!")
    return (t1 - t0) * 1000


ALGOS = [("bubble", bubble, 10000), ("selection", selection, 10000),
         ("insertion", insertion, 10000), ("shell", shell, 1000000),
         ("merge", merge_sort, 1000000), ("quick", quick, 1000000),
         ("heap", heap, 1000000), ("radix", radix, 1000000),
         ("list.sort", list_sort, 1000000)]

print(f"{'algorithm':<17} {'n':>8} {'comparisons':>15} {'swaps/moves':>15} "
      f"{'time ms':>11}")
n = 1000
while n <= 1000000:
    state = 2026
    data = [next_random(1000000000) for _ in range(n)]
    for name, fn, max_n in ALGOS:
        if n <= max_n:
            ms = run(fn, data)
            print(f"{name:<17} {n:>8} {cell(cmp):>15} {cell(mov):>15} "
                  f"{ms:>11.2f}")
    n *= 10
```

Kode 3.36 (Lanjutan) *sort_benchmark.py: tolok ukur dalam Python; pengurutan kuadratik berhenti pada sepuluh ribu elemen.*

```
print("\nn = 10000, time ms by input order")
print(f"{'algorithm':<17} {'random':>11} {'ascending':>11} {'descending':>11}")
n = 10000
state = 2026
rnd = [next_random(1000000000) for _ in range(n)]
asc = list(range(n))
desc = [n - i for i in range(n)]
for name, fn, _ in ALGOS[:3]:
    t1, t2, t3 = run(fn, rnd), run(fn, asc), run(fn, desc)
    print(f"{name:<17} {t1:>11.2f} {t2:>11.2f} {t3:>11.2f}")
```

```
algorithm                n     comparisons     swaps/moves     time ms
bubble                1000          499290          252221       70.93
selection             1000          499500             987       18.72
insertion             1000          253216          252221       20.38
shell                 1000           13187            6893        1.22
merge                 1000            8723            9976        1.09
quick                 1000           10666            3682        0.82
heap                  1000           16846            9095        1.63
radix                 1000               0            4000        0.60
list.sort             1000               -               -        0.08
bubble               10000        49994994        24649368     4596.55
selection            10000        49995000            9982     2151.18
insertion            10000        24659362        24649368     2077.68
shell                10000          190702          101054       16.41
merge                10000          120445          133616       15.29
quick                10000          151336           62176       19.83
heap                 10000          235396          124322       41.10
radix                10000               0           40000        5.17
list.sort            10000               -               -        1.07
shell               100000         2554967         1384924      388.77
merge               100000         1536335         1668928      170.69
quick               100000         1881706          828252      146.43
heap                100000         3019684         1574735      297.72
radix               100000               0          400000       61.88
list.sort           100000               -               -       14.49
shell              1000000        32014545        17456447     7896.72
merge              1000000        18675220        19951424     3778.53
quick              1000000        22507428        10026671     4650.59
heap               1000000        36794517        19047984    14914.96
radix              1000000               0         4000000     1513.46
list.sort          1000000               -               -      307.26

n = 10000, time ms by input order
algorithm              random   ascending  descending
bubble                7287.39        0.46     7513.43
selection             2645.77     3273.60     3412.94
insertion             3371.85        0.74     6455.52
```

Hitungannya lagi-lagi sama persis dengan C++. Baris yang mendidik adalah baris terakhir: `list.sort` bawaan Python, yang ditulis dalam C, mengurutkan sejuta bilangan dalam sekitar 0.3 s, sedangkan *merge sort* Python murni milik kita, dengan pertumbuhan $n \lg n$ yang sama, memerlukan hampir 4 s. Dalam Python aturannya gamblang: selalu pakai pengurutan bawaan, dan curahkan kode Anda sendiri pada fungsi kuncinya.

3.14 Pengurutan Pustaka Masa Kini

Anda perlu memahami setiap algoritma dalam bab ini, dan Anda hampir tidak pernah perlu menulis salah satunya dalam program yang diandalkan orang lain. Pengurutan dalam pustaka standar lebih cepat, lebih teruji, dan direkayasa lebih cermat daripada apa pun yang ditulis dalam satu sore, dan semuanya merupakan hibrida yang memadukan beberapa algoritma yang kini Anda kenal. **Tabel 3.6** merangkumnya per September 2026.

Tabel 3.6 Pengurutan standar C++, Java, dan Python per September 2026, algoritma di dalamnya, dan jaminannya.

Bahasa dan pemanggilan	Algoritma	Stabil	Kasus terburuk
C++ `std::sort`	introsort: quicksort median dari tiga, heap sort bila terlalu dalam, insertion sort pada potongan kecil	tidak	$O(n\lg n)$
C++ `std::stable_sort`	merge sort (insertion sort pada run kecil)	ya	$O(n\lg n)$ dengan penyangga, $O(n\lg^2 n)$ tanpa
C++ `std::partial_sort`	seleksi k terkecil berbasis heap	tidak	$O(n\lg k)$
Java `Arrays.sort(int[])` dan tipe primitif lain	quicksort dua-pivot dengan insertion sort, penggabungan run, cadangan heap sort; counting sort untuk `byte`, `short`, `char`	tidak diperlukan	$O(n\lg n)$
Java `Arrays.sort(Object[])`, `List.sort`	TimSort	ya	$O(n\lg n)$
Python `list.sort`, `sorted`	Timsort dengan kebijakan penggabungan Powersort (sejak 3.11)	ya	$O(n\lg n)$

3.14.1 C++: introsort dan merge sort

Standar C++ tidak menyebut nama algoritma; ia menyatakan persyaratan. `std::sort` harus melakukan $O(n\lg n)$ perbandingan, dan sejak C++11 batas itu berlaku untuk kasus *terburuk*; sebelum koreksi yang dikenal sebagai LWG 713, yang disyaratkan hanya kasus rata-rata, sehingga *quicksort* biasa pun diperbolehkan [10]. Setiap pustaka utama memenuhi persyaratan itu dengan **introsort**, yang ditemukan Musser pada tahun 1997 [11]. *Introsort* adalah *quicksort* yang mengawasi kedalaman rekursinya sendiri: bila kedalaman melampaui batas yang sebanding dengan $\lg n$, masukan sedang berperilaku buruk, dan potongan itu diselesaikan dengan *heap sort*, yang kasus terburuknya $O(n\lg n)$. Dalam pustaka GCC (libstdc++) batasnya $2\lfloor\lg n\rfloor$, pivotnya median dari tiga, partisi berhenti pada potongan berisi 16 elemen atau kurang, dan satu *insertion sort* terakhir atas seluruh larik menuntaskan semuanya sekaligus: pada saat itu setiap elemen berjarak paling jauh 16 tempat dari rumahnya, sehingga *insertion sort* yang adaptif hanya berbiaya $O(n)$ [12]. Tiga algoritma dalam bab ini, masing-masing mengerjakan apa yang paling dikuasainya.

`std::stable_sort` adalah *merge sort*, itulah sebabnya ia stabil [13]. Standar mensyaratkan $O(n\lg n)$ perbandingan bila ia dapat memperoleh penyangga, dan mengizinkan $O(n\lg^2 n)$ bila ia harus menggabung di tempat. Versi GCC mula-mula mengurutkan potongan berisi 7 elemen dengan *insertion sort*, lalu menggabungkannya. `std::partial_sort(first, middle, last)` menaruh `middle - first` elemen terkecil, dalam keadaan terurut, di depan dengan memakai *heap*, alat yang tepat untuk "sepuluh teratas" tanpa mengurutkan semuanya.

3.14.2 Java: quicksort dua-pivot dan TimSort

Untuk larik bertipe primitif (`int`, `double`, dan seterusnya), `Arrays.sort` memakai **quicksort dua-pivot** (*dual-pivot quicksort*) karya Yaroslavskiy, Bentley, dan Bloch, yang mempartisi di sekitar *dua* pivot menjadi tiga bagian; dokumentasinya menjanjikan $O(n\lg n)$ pada semua himpunan data [14]. Implementasi saat ini, yang ditulis ulang untuk JDK 14, juga beralih ke *insertion sort* untuk bagian-bagian kecil, mengenali dan menggabungkan *run* yang sudah terurut, beralih ke *heap sort* bila rekursinya terlalu dalam, dan memakai *counting sort* untuk larik `byte`, `short`, dan `char`, yang

kuncinya paling banyak mempunyai 65,536 nilai [15]. Kestabilan tidak penting untuk tipe primitif: dua `int` yang sama tidak dapat dibedakan.

Untuk larik objek serta untuk `List.sort` dan `Collections.sort`, kestabilan penting, dan spesifikasinya mewajibkannya. Sejak Java 7 algoritmanya adalah **TimSort**, alih bahasa dari pengurutan list Python [16]: *merge sort* iteratif yang stabil dan adaptif yang "memerlukan jauh lebih sedikit dari $n \lg n$ perbandingan bila dijalankan pada larik yang sebagian terurut". Ia mencari *run* yang sudah menaik (atau menurun tegas, yang lalu dibaliknya), memperpanjang *run* yang pendek sampai panjang minimum dengan *binary insertion sort*, lalu menggabungkan *run-run* itu; panjang penggabungan minimumnya 32 elemen.

3.14.3 Python: Timsort dan Powersort

`list.sort` dan `sorted` milik Python selalu stabil, dan dokumentasinya menjaminnya, sehingga pengurutan multikunci dengan pengurutan berturut-turut (Subbab 3.12.1) didukung secara resmi [17]. Algoritmanya dirancang Tim Peters pada tahun 2002 dan diuraikan dalam catatannya `listsort.txt` di kode sumber CPython [18]. Seperti TimSort, ia memindai *run* alami, memperpanjang yang pendek dengan *binary insertion sort*, lalu menggabungkannya, dengan memakai *galloping* (pencarian eksponensial yang diikuti pencarian biner) ketika satu *run* terus-menerus menang. Yang berubah pada **Python 3.11** adalah *kebijakan penggabungannya* (*merge policy*), aturan yang memutuskan *run* bertetangga mana yang digabung dan kapan. Aturan lama mempunyai kasus-kasus langka yang penggabungannya buruk; sejak 3.11 CPython memakai aturan **Powersort** karya Munro dan Wild [19], [20], yang terbukti hanya selisih suku tambahan kecil dari urutan penggabungan terbaik yang mungkin untuk panjang-panjang *run* yang diberikan. Dampaknya pada data biasa kecil; dampaknya pada kasus buruk besar. Catatan yang sama merekam penyempurnaan lanjutan yang digabungkan ke cabang pengembangan CPython pada tahun 2025, setelah pembekuan fitur 3.14: panjang *run* minimum kini sedikit divariasikan dari *run* ke *run* agar pohon penggabungannya seimbang sempurna [18].

`sort` milik Python menerima **fungsi kunci** (*key function*) alih-alih perbandingan: `key=lambda b: b.category` menghitung kunci setiap elemen sekali saja, yang lebih cepat sekaligus lebih mudah dibuat benar daripada membandingkan, dan `reverse=True` mengurutkan secara menurun sambil *tetap* stabil.

3.14.4 Memakainya

Kode 3.37, **Kode 3.38**, dan **Kode 3.39** mengerjakan lima pekerjaan yang sama dengan pustaka masing-masing bahasa: menaik, menurun, tiga teratas dari daftar skor, pengurutan stabil katalog menurut kategori, dan pengurutan menurut judul.

Kode 3.37 `library_sorts.cpp`: `std::sort` dengan dan tanpa komparator, `std::partial_sort`, dan `std::stable_sort` dengan *lambda*.

```cpp
// library_sorts.cpp - pengurutan pustaka standar, dengan komparator
#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct Book {
    string id, title, category;
};

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    sort(data.begin(), data.end());                    // introsort, menaik
    cout << "ascending:    " << show(data) << "\n";
    // komparator menentukan urutan
    sort(data.begin(), data.end(), greater<int>());
    cout << "descending:   " << show(data) << "\n";

    vector<int> scores = {71, 94, 58, 88, 94, 63, 79};
    partial_sort(scores.begin(), scores.begin() + 3, scores.end(),
                 greater<int>());
    vector<int> top3(scores.begin(), scores.begin() + 3);
    cout << "top 3 scores: " << show(top3) << "\n";

    vector<Book> catalog = {
        {"B1001", "Laskar Pelangi", "Fiksi"},
        {"B1002", "Bumi Manusia", "Fiksi"},
        {"B1003", "Filosofi Kopi", "Fiksi"},
        {"B1004", "Sejarah Nusantara", "Sejarah"},
        {"B1005", "Algoritma & Struktur Data", "Teknologi"},
        {"B1006", "Cerita Rakyat Nusantara", "Anak-anak"},
        {"B1007", "Fisika Dasar", "Sains"},
        {"B1008", "Negeri 5 Menara", "Fiksi"},
    };
    auto byCategory = [](const Book& a, const Book& b) {
        return a.category < b.category;
    };
    stable_sort(catalog.begin(), catalog.end(), byCategory);
```

Kode 3.37 (Lanjutan) *library_sorts.cpp: std::sort dengan dan tanpa komparator, std::partial_sort, dan std::stable_sort dengan lambda.*

```cpp
    cout << "by category, id order kept inside each category:\n";
    for (const Book& b : catalog)
        cout << "  " << b.category << " " << b.id << "\n";

    sort(catalog.begin(), catalog.end(),
         [](const Book& a, const Book& b) { return a.title < b.title; });
    cout << "by title:\n";
    for (const Book& b : catalog) cout << "  " << b.title << "\n";
    return 0;
}
```

Kode 3.38 `LibrarySorts.java`: `Arrays.sort` pada `int[]` dan pada `Integer[]` dengan `Collections.reverseOrder()`, serta `List.sort` dengan `Comparator.comparing`.

```java
// LibrarySorts.java - pengurutan pustaka standar, dengan komparator
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class LibrarySorts {
    record Book(String id, String title, String category) { }

    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        Arrays.sort(data);  // quicksort dua-pivot untuk int[]
        System.out.println("ascending:    " + show(data));
        Integer[] boxed = {22, 10, 15, 3, 8, 2};
        // objek: TimSort, dengan komparator
        Arrays.sort(boxed, Collections.reverseOrder());
        System.out.println("descending:   "
                + Arrays.toString(boxed).replace(",", ""));

        Integer[] scores = {71, 94, 58, 88, 94, 63, 79};
        Arrays.sort(scores, Collections.reverseOrder());
        System.out.println("top 3 scores: "
                + Arrays.toString(Arrays.copyOf(scores, 3)).replace(",", ""));

        List<Book> catalog = new ArrayList<>(List.of(
            new Book("B1001", "Laskar Pelangi", "Fiksi"),
            new Book("B1002", "Bumi Manusia", "Fiksi"),
            new Book("B1003", "Filosofi Kopi", "Fiksi"),
            new Book("B1004", "Sejarah Nusantara", "Sejarah"),
            new Book("B1005", "Algoritma & Struktur Data", "Teknologi"),
            new Book("B1006", "Cerita Rakyat Nusantara", "Anak-anak"),
            new Book("B1007", "Fisika Dasar", "Sains"),
            new Book("B1008", "Negeri 5 Menara", "Fiksi")));
        // List.sort selalu stabil
        catalog.sort(Comparator.comparing(Book::category));
        System.out.println("by category, id order kept inside each category:");
        for (Book b : catalog) System.out.println("  " + b.category() + " "
                + b.id());
```

Kode 3.38 (Lanjutan) *LibrarySorts.java: Arrays.sort pada int[] dan pada Integer[] dengan Collections.reverseOrder(), serta List.sort dengan Comparator.comparing.*

```java

        catalog.sort(Comparator.comparing(Book::title));
        System.out.println("by title:");
        for (Book b : catalog) System.out.println("  " + b.title());
    }
}
```

Kode 3.39 `library_sorts.py`: `list.sort`, `sorted` dengan kunci, dan `heapq.nlargest`.

```python
# library_sorts.py - pengurutan pustaka standar, dengan komparator
import heapq
from dataclasses import dataclass


@dataclass
class Book:
    id: str
    title: str
    category: str


def show(a):
    return "[" + " ".join(map(str, a)) + "]"


data = [22, 10, 15, 3, 8, 2]
data.sort()                                 # Timsort, di tempat, menaik
print(f"ascending:    {show(data)}")
data.sort(reverse=True)                     # tetap stabil, hanya urutannya dibalik
print(f"descending:   {show(data)}")

scores = [71, 94, 58, 88, 94, 63, 79]
print(f"top 3 scores: {show(heapq.nlargest(3, scores))}")

catalog = [
    Book("B1001", "Laskar Pelangi", "Fiksi"),
    Book("B1002", "Bumi Manusia", "Fiksi"),
    Book("B1003", "Filosofi Kopi", "Fiksi"),
    Book("B1004", "Sejarah Nusantara", "Sejarah"),
    Book("B1005", "Algoritma & Struktur Data", "Teknologi"),
    Book("B1006", "Cerita Rakyat Nusantara", "Anak-anak"),
    Book("B1007", "Fisika Dasar", "Sains"),
    Book("B1008", "Negeri 5 Menara", "Fiksi"),
]
catalog.sort(key=lambda b: b.category)   # setiap pengurutan Python stabil
print("by category, id order kept inside each category:")
for b in catalog:
    print(f"  {b.category} {b.id}")

print("by title:")
# sorted() mengembalikan list baru
for b in sorted(catalog, key=lambda b: b.title):
    print(f"  {b.title}")
```

```
ascending:    [2 3 8 10 15 22]
descending:   [22 15 10 8 3 2]
top 3 scores: [94 94 88]
by category, id order kept inside each category:
  Anak-anak B1006
  Fiksi B1001
  Fiksi B1002
  Fiksi B1003
  Fiksi B1008
  Sains B1007
  Sejarah B1004
  Teknologi B1005
by title:
  Algoritma & Struktur Data
  Bumi Manusia
  Cerita Rakyat Nusantara
  Filosofi Kopi
  Fisika Dasar
  Laskar Pelangi
  Negeri 5 Menara
  Sejarah Nusantara
```

Dalam C++ komparator harus berupa **strict weak ordering** (urutan lemah tegas): ia menjawab "apakah a tegas sebelum b?", sehingga ia memakai `<`, tidak pernah `<=`. Komparator yang mengembalikan benar untuk elemen yang sama merusak asumsi pustaka, dan `std::sort` lalu bisa membaca di luar larik. Java tidak mempunyai pengurutan parsial, sehingga **Kode 3.38** mengurutkan ketujuh skor dan mengambil tiga yang pertama; `heapq.nlargest` milik Python memakai *heap* persis seperti `std::partial_sort`.

3.15 Profitina dalam Praktik

Profitina adalah platform visibilitas AI: ia memindai situs web sebuah merek, menilai seberapa siap halaman-halamannya dikutip oleh mesin penjawab AI, menjelaskan setiap sinyal yang diukurnya, dan membantu pemilik memperbaiki apa yang ditemukannya. Aplikasi webnya ditulis dalam Python dengan FastAPI dan menyimpan datanya di PostgreSQL. Pengurutan muncul di setiap lapisannya, biasanya tanpa terlihat.

Daftar sinyal. Sebuah pemindaian menghasilkan daftar temuan, masing-masing dengan tingkat keparahan dan taksiran dampak terhadap skor. Pemilik merek yang bukan orang teknik seharusnya melihat perbaikan terpenting lebih dahulu, sehingga daftar itu diurutkan menurut dampak, yang tertinggi dahulu. Dua temuan sering mempunyai dampak yang sama; urutan di antara keduanya tidak boleh berubah setiap kali halaman dimuat ulang, karena daftar yang mengacak dirinya sendiri pada setiap kunjungan tampak tidak dapat dipercaya. Pengurutan stabil, yang diterapkan pada temuan yang datang dalam urutan tetap (menurut jenis sinyal), memberikan persis hal itu: dampak yang sama mempertahankan urutan yang dapat diduga. Dalam Python ini cukup satu baris, `sorted(findings, key=lambda f: f.impact, reverse=True)`, karena `reverse=True` mempertahankan kunci yang sama dalam urutan aslinya (Subbab 3.14.3).

Tren mingguan. Dasbor historis memperlihatkan skor setiap merek minggu demi minggu, dan penjadwal menambahkan skor baru setelah setiap pemindaian ulang otomatis. Karena itu datanya datang dalam keadaan *hampir terurut menurut waktu*: satu *run* menaik yang panjang dengan, sesekali, hasil yang terlambat dari pemindaian yang tertunda di ujungnya. Itulah kasus terbaik pengurutan adaptif. Timsort mendeteksi *run* panjang itu dalam satu lintasan dan menggabungkan beberapa yang tercecer ke dalamnya, sehingga mengurutkan tren itu berbiaya mendekati n perbandingan alih-alih $n \lg n$; itulah sebabnya mengurutkan ulang deret itu setiap kali sebuah titik ditambahkan hampir tidak berbiaya.

Papan peringkat. Pengguna memperoleh PROFIT dengan memperbaiki jawaban AI, dan sebuah papan peringkat mengurutkan mereka. Mengurutkan menurut kunci utama (poin perolehan) dengan penentu seri (siapa yang lebih dahulu mencapai total itu) adalah pengurutan multikunci pada Subbab 3.12.1, dan halamannya hanya memerlukan beberapa entri teratas, pekerjaan untuk sebuah *heap*: `heapq.nlargest(10, …)` dalam Python, `std::partial_sort` dalam C++, $O(n \lg k)$ alih-alih $O(n \lg n)$.

Di dalam basis data. Ketika sebuah kueri menyatakan `ORDER BY`, PostgreSQL mengurutkan. Keluaran `EXPLAIN ANALYZE`-nya menyebutkan metode yang dipilihnya: `quicksort` bila baris-barisnya muat dalam memori yang diizinkan untuk pengurutan, `top-N heapsort` bila kueri mempunyai `LIMIT` dan hanya baris-baris pertama yang diperlukan, dan `external merge` bila baris-barisnya tidak muat di memori sehingga diurutkan per potongan di disk lalu digabungkan [21]. Itulah *quicksort*, *heap sort*, dan *merge sort*, ketiga algoritma $n \lg n$ dalam bab ini, dipilih secara otomatis menurut persis pertimbangan pada **Tabel 3.2**. Mengenal ketiganya membuat

pengembang mampu membaca rencana kueri yang lambat dan melihat, misalnya, bahwa menambahkan indeks akan membuat basis data melewati pengurutan sama sekali.

Bagian yang dapat dijalankan dari subbab ini memakai **Profitina Laundry**, usaha laundry kecil yang menjadi studi kasus buku ini. Dua belas pesanannya dalam satu hari datang menurut urutan waktu, seperti pada Bab 2. Pada awal giliran mencuci, petugas menginginkan antrean dengan muatan terberat lebih dahulu, dan di antara muatan yang sama beratnya, pelanggan yang datang lebih dahulu dilayani lebih dahulu; di meja depan mereka menginginkan daftar pengambilan yang alfabetis. Permintaan pertama adalah pengurutan stabil menurut berat, menurun.

Kode 3.40 `laundry_orders.cpp`: antrean cuci Profitina Laundry dengan `std::stable_sort`, dan daftar pengambilannya dengan `std::sort`.

```cpp
// laundry_orders.cpp - Profitina Laundry: terberat dulu, seri menurut
// kedatangan
#include <algorithm>
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

struct Order {
    int number;
    string time, customer;
    double kg;
};

void print(const char* heading, const vector<Order>& orders) {
    printf("%s\n", heading);
    for (const Order& o : orders)
        printf("  #%d %s %-5s %4.1f kg\n", o.number, o.time.c_str(),
               o.customer.c_str(), o.kg);
}

int main() {
    vector<Order> orders = {  // menurut urutan kedatangan
        {1001, "07:45", "Sari", 3.5}, {1002, "08:10", "Budi", 5.0},
        {1003, "08:55", "Dewi", 2.0}, {1004, "09:30", "Andi", 7.5},
        {1005, "10:05", "Rina", 4.0}, {1006, "10:40", "Joko", 6.0},
        {1007, "11:15", "Maya", 3.0}, {1008, "11:50", "Tono", 8.0},
        {1009, "12:20", "Lina", 2.5}, {1010, "13:05", "Agus", 5.5},
        {1011, "14:30", "Fitri", 4.5}, {1012, "15:10", "Hadi", 3.0},
    };
    vector<Order> byWeight = orders;
    stable_sort(byWeight.begin(), byWeight.end(),
                [](const Order& a, const Order& b) { return a.kg > b.kg; });
    print("Washing queue, heaviest first (equal weights keep arrival order):",
          byWeight);

    vector<Order> byName = orders;
    sort(byName.begin(), byName.end(), [](const Order& a, const Order& b) {
        return a.customer < b.customer;
    });
    printf("Pickup list:");
    for (const Order& o : byName) printf(" %s", o.customer.c_str());
    printf("\n");
    return 0;
}
```

Kode 3.41 `LaundryOrders.java`: antrean yang sama dengan `List.sort`, yang stabil, dan `Comparator.comparingDouble(...).reversed()`.

```java
// LaundryOrders.java - Profitina Laundry: terberat dulu, seri menurut
// kedatangan
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class LaundryOrders {
    record Order(int number, String time, String customer, double kg) { }

    static void print(String heading, List<Order> orders) {
        System.out.println(heading);
        for (Order o : orders)
            System.out.printf("  #%d %s %-5s %4.1f kg%n",
                    o.number(), o.time(), o.customer(), o.kg());
    }

    public static void main(String[] args) {
        List<Order> orders = List.of(  // menurut urutan kedatangan
            new Order(1001, "07:45", "Sari", 3.5),
            new Order(1002, "08:10", "Budi", 5.0),
            new Order(1003, "08:55", "Dewi", 2.0),
            new Order(1004, "09:30", "Andi", 7.5),
            new Order(1005, "10:05", "Rina", 4.0),
            new Order(1006, "10:40", "Joko", 6.0),
            new Order(1007, "11:15", "Maya", 3.0),
            new Order(1008, "11:50", "Tono", 8.0),
            new Order(1009, "12:20", "Lina", 2.5),
            new Order(1010, "13:05", "Agus", 5.5),
            new Order(1011, "14:30", "Fitri", 4.5),
            new Order(1012, "15:10", "Hadi", 3.0));

        List<Order> byWeight = new ArrayList<>(orders);
        // stabil
        byWeight.sort(Comparator.comparingDouble(Order::kg).reversed());
        print("Washing queue, heaviest first (equal weights keep arrival "
                + "order):", byWeight);

        List<Order> byName = new ArrayList<>(orders);
        byName.sort(Comparator.comparing(Order::customer));
        StringBuilder sb = new StringBuilder("Pickup list:");
        for (Order o : byName) sb.append(' ').append(o.customer());
        System.out.println(sb);
    }
}
```

Kode 3.42 `laundry_orders.py`: antrean yang sama dengan `sorted(..., reverse=True)`.

```python
# laundry_orders.py - Profitina Laundry: terberat dulu, seri menurut kedatangan
# menurut urutan kedatangan
ORDERS = [(1001, "07:45", "Sari", 3.5), (1002, "08:10", "Budi", 5.0),
          (1003, "08:55", "Dewi", 2.0), (1004, "09:30", "Andi", 7.5),
          (1005, "10:05", "Rina", 4.0), (1006, "10:40", "Joko", 6.0),
          (1007, "11:15", "Maya", 3.0), (1008, "11:50", "Tono", 8.0),
          (1009, "12:20", "Lina", 2.5), (1010, "13:05", "Agus", 5.5),
          (1011, "14:30", "Fitri", 4.5), (1012, "15:10", "Hadi", 3.0)]

# stabil, meski dibalik
by_weight = sorted(ORDERS, key=lambda o: o[3], reverse=True)
print("Washing queue, heaviest first (equal weights keep arrival order):")
for o in by_weight:
    print("  #%d %s %-5s %4.1f kg" % o)

by_name = sorted(ORDERS, key=lambda o: o[2])
print("Pickup list:", " ".join(o[2] for o in by_name))
```

Kode 3.41 dan **Kode 3.42** mencetak persis apa yang dicetak **Kode 3.40**:

```
Washing queue, heaviest first (equal weights keep arrival order):
  #1008 11:50 Tono   8.0 kg
  #1004 09:30 Andi   7.5 kg
  #1006 10:40 Joko   6.0 kg
  #1010 13:05 Agus   5.5 kg
  #1002 08:10 Budi   5.0 kg
  #1011 14:30 Fitri  4.5 kg
  #1005 10:05 Rina   4.0 kg
  #1001 07:45 Sari   3.5 kg
  #1007 11:15 Maya   3.0 kg
  #1012 15:10 Hadi   3.0 kg
  #1009 12:20 Lina   2.5 kg
  #1003 08:55 Dewi   2.0 kg
Pickup list: Agus Andi Budi Dewi Fitri Hadi Joko Lina Maya Rina Sari Tono
```

Dua pesanan beratnya 3.0 kg: milik Maya, yang datang pukul 11.15, dan milik Hadi, pukul 15.10. Dalam antrean cuci pesanan Maya lebih dahulu, karena ketiga program memakai pengurutan stabil dan masukannya dalam urutan kedatangan. Dengan `std::sort` pada **Kode 3.40** urutan itu tidak dijamin. Komparator C++ `a.kg > b.kg` tetap merupakan *strict weak ordering*: "lebih berat daripada" itu tegas, sehingga berat yang sama dinilai tidak sebelum dan tidak sesudah.

3.16 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. Menukar tanpa variabel sementara. `a = b; b = a;` membuat kedua variabel berisi `b` yang lama. *Perbaikan:* simpan nilai pertama sebelum menimpanya (Subbab 3.2.4), atau pakai `std::swap` dalam C++ dan penugasan tuple dalam Python.

2. Menulis fungsi swap C++ atau Java yang menerima nilai. `void swap(int a, int b)` menukar salinannya sendiri dan tidak mengubah apa pun bagi pemanggil. *Perbaikan:* dalam C++ teruskan pointer atau referensi; dalam Java tukar dua slot larik yang diterima metode.

3. Menjalankan gelung dalam satu langkah terlalu jauh. *Bubble sort* membandingkan `a[j]` dengan `a[j+1]`, jadi `j` harus berhenti pada `n - pass - 1`; gelung sampai `n - 1` membaca `a[n]`, melewati ujung larik. *Perbaikan:* untuk setiap gelung yang membaca `a[j+1]`, periksa nilai `j` terbesar yang dicapainya.

4. Menguji elemen sebelum indeks. `while (a[j] > key && j >= 0)` membaca `a[-1]` bila kunci seharusnya berada di depan. *Perbaikan:* uji indeksnya lebih dahulu, `j >= 0 && a[j] > key`, dan andalkan evaluasi hubung-singkat (Subbab 3.6.2). Dalam Python urutan yang salah tidak menyebabkan program berhenti; ia diam-diam membandingkan dengan elemen terakhir.

5. Melupakan keluar dini, atau meletakkannya di tempat yang salah. *Bubble sort* tanpa penanda `swapped` tidak pernah adaptif. Penanda yang hanya di-*reset* sekali, sebelum gelung luar, tidak akan pernah bernilai salah lagi. *Perbaikan:* setel `swapped = false` pada awal *setiap* lintasan dan ujilah pada akhir lintasan itu.

6. Kehilangan kestabilan karena satu karakter. *Bubble sort* dengan `>=`, *insertion sort* dengan `>=` pada uji pergeserannya, atau penggabungan dengan `<` alih-alih `<=` semuanya menukar kunci yang sama dan membuat algoritma yang stabil menjadi tidak stabil. *Perbaikan:* geser atau tukar hanya pada ketaksamaan *tegas*, dan ambil dari *run* kiri bila sama.

7. Mengalokasikan penyangga penggabungan di dalam setiap penggabungan. `new` atau `vector` di dalam `merge` mengalokasikan dan membebaskan memori $n - 1$ kali, yang dapat mendominasi waktu jalan. *Perbaikan:* alokasikan satu penyangga berukuran n sekali saja dan teruskan ke bawah, seperti pada **Pseudokode 3.6**.

8. Menghitung titik tengah sebagai `(lo + hi) / 2`. Untuk larik yang sangat besar jumlahnya meluapi `int`. *Perbaikan:* `lo + (hi - lo) / 2`.

9. Mencampur dua skema partisi. Partisi Lomuto mengembalikan posisi pivot, dan pemanggilan rekursifnya mengecualikan pivot itu: `(lo, p - 1)` dan `(p + 1, hi)`. Partisi Hoare mengembalikan titik belah, dan pemanggilannya `(lo, j)` dan `(j + 1, hi)`. Memakai pemanggilan Lomuto dengan partisi Hoare menghilangkan elemen atau berekursi tanpa henti. *Perbaikan:* tulis partisi dan kedua pemanggilan rekursifnya bersama-sama, sebagai satu kesatuan.

10. Memakai elemen pertama atau terakhir sebagai pivot pada data yang tidak Anda kendalikan. Masukan terurut, masukan yang paling lazim, menjadi kasus terburuk: waktu $\Theta(n^2)$ dan rekursi n tingkat. *Perbaikan:* median dari tiga atau pivot acak; berekursi ke sisi yang lebih kecil; atau panggil pustaka, yang mengerjakan keduanya dan lebih lagi.

11. Menganggap pengurutan kuadratik tidak apa-apa "karena komputer cepat". Pada 10^5 elemen *bubble sort* kita memerlukan 28 detik dan `std::sort` 6 milidetik. *Perbaikan:* pakai **Tabel 3.2** sebelum memilih, dan ukurlah.

12. Menganggap algoritma tercepat adalah yang perbandingannya paling sedikit. *Merge sort* paling sedikit membandingkan, tetapi *quicksort* lebih cepat, dan *heap sort*, dengan biaya asimtotik yang sama, dua kali lebih lambat. *Perbaikan:* ingat bahwa lalu lintas memori dan perilaku *cache* sama pentingnya dengan perbandingan.

13. Komparator yang bukan *strict weak ordering*. Dalam C++, `[](int a, int b) { return a <= b; }` memberi tahu `std::sort` bahwa sebuah elemen datang sebelum dirinya sendiri; akibatnya perilaku tak terdefinisi, kadang program berhenti mendadak. *Perbaikan:* komparator memakai `<` atau `>`, tidak pernah `<=` atau `>=`. Dalam Java metode `compare` harus mengembalikan bilangan negatif, nol, atau positif secara konsisten.

14. Mencacah dengan `int`. *Bubble sort* pada 10^5 elemen melakukan sekitar 5×10^9 perbandingan, lebih dari yang dapat ditampung `int` 32-bit. *Perbaikan:* cacah dengan `long long` dalam C++ dan `long` dalam Java, seperti pada setiap program dalam bab ini.

15. Menulis pengurutan sendiri dalam kode produksi. Pengurutan buatan tangan lebih lambat, kurang teruji, dan biasanya tidak stabil ketika kestabilan diandaikan. *Perbaikan:* panggil `std::sort`/`std::stable_sort`, `Arrays.sort`/`List.sort`, atau `sorted`/`list.sort`, dan curahkan tenaga Anda pada komparator atau kuncinya.

3.17 Ringkasan Bab

1. **Pengurutan** menata ulang butir-butir sehingga kuncinya berurutan (Persamaan (3.1)). Ia mengubah pencarian, pendeteksian duplikat, pengelompokan, penggabungan, dan pemeringkatan menjadi operasi yang murah, itulah sebabnya ia termasuk masalah yang paling banyak dikaji dalam komputasi.
2. **Setiap pengurutan bertumpu pada penukaran nilai yang aman**: tiga penugasan dan satu variabel sementara. C++ dapat menulis fungsi penukar dengan pointer atau referensi; Java hanya dapat menukar slot larik; Python memakai penugasan tuple.
3. **Perbandingan, pertukaran, dan pemindahan** mengukur pengurutan tanpa bergantung pada mesin. Banyaknya **inversi** (Persamaan (3.2)) mengukur seberapa tidak terurut sebuah larik; setiap algoritma yang hanya menukar tetangga melakukan tepat satu pertukaran per inversi (Persamaan (3.4)), rata-rata $n(n-1)/4$ (Persamaan (3.5)), sehingga pasti kuadratik.

4. **Bubble sort** membawa elemen terbesar ke ujung pada setiap lintasan; dengan **keluar dini** ia adaptif, $n-1$ perbandingan pada masukan terurut dan $n(n-1)/2$ pada kasus terburuk (Persamaan (3.3)). **Exchange sort** membandingkan satu posisi dengan semua posisi sesudahnya dan selalu kuadratik. Keduanya di tempat; *bubble sort* stabil, *exchange sort* tidak.

5. **Selection sort** mencari minimum dan melakukan satu pertukaran per putaran: selalu $n(n-1)/2$ perbandingan, paling banyak $n-1$ pertukaran, tidak stabil, tidak adaptif.

6. **Insertion sort** menumbuhkan awalan terurut dengan menggeser; biayanya sebanding dengan n ditambah banyaknya inversi (Persamaan (3.6)). Ia stabil, di tempat, adaptif, dan cepat pada larik kecil atau hampir terurut, itulah sebabnya pengurutan pustaka memakainya untuk menuntaskan pekerjaan.

7. **Shell sort** menjalankan *insertion sort* dengan celah yang makin kecil. Lompatan jauh di awal menghapus banyak inversi sekaligus; dengan celah Knuth (Persamaan (3.7)) kasus terburuknya $O(n^{3/2})$, dan celah Ciura terbaik dalam praktik. Ia tidak stabil.

8. **Merge sort** membagi di tengah dan menggabung; Persamaan (3.8) terselesaikan menjadi $\Theta(n\lg n)$ pada setiap kasus (Persamaan (3.9)). Ia stabil tetapi memerlukan memori tambahan sebanyak n elemen.

9. **Quicksort** mempartisi di sekitar pivot (satu penyusuran Lomuto atau dua indeks Hoare yang saling mendekat) dan berekursi tanpa penggabungan. Ia $\Theta(n\lg n)$ pada kasus rata-rata dengan sekitar $1.39\,n\lg n$ perbandingan (Persamaan (3.11)) tetapi $\Theta(n^2)$ pada kasus terburuk (Persamaan (3.10)), yang terkena oleh pivot tetap pada masukan terurut. Pivot median dari tiga atau acak, dan berekursi ke sisi yang lebih kecil, membuatnya tangguh. Dalam praktik ia pengurutan berbasis perbandingan yang tercepat.

10. **Heap sort** menyimpan *max-heap* di dalam larik (Persamaan (3.12)) dan berulang kali memindahkan akar ke ujung: $O(n\lg n)$ pada kasus terburuk dan di tempat, tetapi tidak stabil dan tidak ramah *cache*. Bab 14 membangun *heap* secara lengkap.

11. **Tidak ada pengurutan berbasis perbandingan yang mengalahkan** $\Omega(n\lg n)$ pada kasus terburuk (Persamaan (3.13)). **Counting sort** (Persamaan (3.14)) dan **radix sort** (Persamaan (3.15)) menghindari perbandingan dengan memakai nilai kunci sebagai indeks, dan linear untuk kunci bilangan bulat kecil; keduanya stabil, dan *radix sort* bergantung pada kestabilan itu.

12. Pengurutan **stabil** mempertahankan kunci yang sama dalam urutan masukan; itu memungkinkan pengurutan multikunci dengan mengurutkan kunci yang paling tidak penting lebih dahulu. Pengurutan **di tempat** hanya memerlukan memori tambahan konstan. Pengurutan **adaptif** bekerja lebih sedikit pada masukan yang hampir terurut. **Tabel 3.2** mendaftar ketiganya untuk setiap algoritma.

13. **Pengukuran** menegaskan rumus-rumusnya: pengurutan kuadratik tumbuh seratus kali (atau lebih, akibat *cache*) untuk setiap pertumbuhan n sepuluh kali, pengurutan $n\lg n$ sekitar dua belas kali; pada sejuta elemen selisihnya faktor sekitar 25,000.

14. **Pengurutan pustaka adalah hibrida**: `std::sort` adalah *introsort* (*quicksort*, *heap sort*, *insertion sort*), `std::stable_sort` adalah *merge sort*, Java mengurutkan tipe primitif dengan *quicksort* dua-pivot dan objek dengan TimSort, dan `list.sort` Python adalah Timsort dengan kebijakan penggabungan Powersort sejak 3.11. Pakailah semuanya.

3.17.1 Menatap ke depan

Bab 4 mengerjakan sekumpulan soal evaluasi yang diambil dari Bab 1 sampai 3, beberapa di antaranya meminta Anda mengurutkan *string* dan rekaman dengan komparator buatan sendiri. Bab 5 memulai struktur berantai dengan *stack* dan *queue*. Rekursi, yang dipakai *merge sort* dan *quicksort* di sini tanpa banyak upacara, mendapat bab tersendiri pada Bab 10, tempat pohon-pohon rekursi dari bab ini digambar dan diselesaikan secara umum. *Heap* kembali pada Bab 14 sebagai antrean prioritas, dan *hashing* pada Bab 15 menjawab pertanyaan yang dibiarkan terbuka oleh pengurutan: bagaimana menemukan sebuah butir dalam waktu harapan $O(1)$ tanpa menjaga apa pun tetap berurutan.

3.18 Soal Latihan

3.1 Lacak *bubble sort* dengan keluar dini pada [5,1,4,2,8]. Tunjukkan larik setelah setiap lintasan, dan hitung perbandingan serta pertukarannya. Berapa banyak inversi masukan itu?

3.2 Jalankan *selection sort* dengan tangan pada larik terbalik [6,5,4,3,2,1]. Berapa perbandingan dan berapa pertukaran yang dilakukannya? Bandingkan dengan *bubble sort* pada masukan yang sama.

3.3 Jalankan *insertion sort* pada [3,1,2,5,4] dan hitung perbandingan serta pergeserannya. Periksa hitungan Anda terhadap Persamaan (3.6).

3.4 *(Pemrograman.) Bubble sort* memindahkan elemen kecil yang berada di dekat ujung larik hanya satu tempat ke kiri per lintasan. **Cocktail shaker sort** menyelang-nyelingkan lintasan kiri-ke-kanan, yang membawa elemen terbesar ke kanan, dengan lintasan kanan-ke-kiri, yang membawa elemen terkecil ke kiri. Implementasikan dalam C++, Java, dan Python, jalankan pada [2,3,8,10,15,22,1], dan bandingkan perbandingannya dengan *bubble sort* berketentuan keluar dini pada masukan yang sama.

3.5 Buktikan bahwa *bubble sort* sebagaimana ditulis pada **Pseudokode 3.1** stabil. Kemudian tunjukkan, dengan contoh dua elemen, bahwa mengganti `>` dengan `>=` pada baris 4 membuatnya tidak stabil.

3.6 Jalankan *exchange sort* pada [4,3,2,1]. Berapa pertukaran yang dilakukannya? Berapa yang akan dilakukan *selection sort*? Jelaskan perbedaannya.

3.7 Celah apa saja yang dihasilkan barisan Knuth Persamaan (3.7), sebagaimana dipakai dalam **Kode 3.16**, untuk $n = 100$ dan untuk $n = 1000$?

3.8 Gabungkan *run* terurut [1,4,7,9] dan [2,3,8,10,12] seperti yang dilakukan **Pseudokode 3.6**. Berapa perbandingan yang dilakukan? Berapa banyak perbandingan terbesar yang mungkin diperlukan untuk menggabungkan dua *run* berpanjang 4 dan 5, dan masukan apa yang memerlukannya?

3.9 Terapkan partisi Lomuto (**Pseudokode 3.8**) pada [7,2,9,4,3,8,5]. Tunjukkan larik dan nilai i setelah setiap nilai j, dan sebutkan indeks yang dikembalikan.

3.10 Apa yang terjadi pada *quicksort* dengan partisi Lomuto bila semua n kuncinya sama? Apa yang terjadi dengan partisi Hoare? Berikan banyaknya perbandingan pada masing-masing kasus sebagai fungsi dari n.

3.11 Bangun *max-heap* dari [4,1,3,2,16,9,10] dengan fase pembangunan pada **Pseudokode 3.10**. Tunjukkan larik setelah setiap pemanggilan `SIFT-DOWN`.

3.12 Urutkan [3,0,2,3,1,0,2] dengan *counting sort*, $k = 4$. Tunjukkan `count` setelah gelung pencacahan dan setelah jumlah kumulatif, serta keluarannya.

3.13 Urutkan [170,45,75,90,802,24,2,66] dengan *radix sort* LSD berbasis 10. Tunjukkan larik setelah setiap lintasan.

3.14 *(Pemrograman.)* Ubah *merge sort* sehingga ia juga mengembalikan banyaknya inversi larik, dalam waktu $O(n \lg n)$. Ujilah terhadap hitungan $O(n^2)$ pada A, pada larik terurut dan terbalik, dan pada [5,1,4,2,3], dalam ketiga bahasa.

3.15 Dengan memakai Persamaan (3.13), tentukan batas bawah banyaknya perbandingan yang diperlukan pengurutan berbasis perbandingan mana pun, pada kasus terburuk, untuk mengurutkan 5 elemen, dan untuk $n = 1000$.

3.16 Tunjukkan dengan contoh tiga elemen bahwa *quicksort* dengan partisi Lomuto tidak stabil.

3.17 Profitina Laundry ingin pesanan hari itu didaftar menurut nama pelanggan dan, untuk pelanggan yang sama, yang terberat lebih dahulu. Bila hanya memakai pengurutan stabil, kunci mana yang harus diurutkan lebih dahulu? Jelaskan mengapa urutan sebaliknya gagal.

3.18 Untuk setiap situasi, pilih metode pengurutan dari bab ini atau pemanggilan pustaka, dan berikan alasannya dalam satu atau dua kalimat: (a) 20 bilangan pada mikrokontroler kecil dengan memori sangat sedikit; (b) berkas log sejuta baris yang terurut menurut waktu kecuali beberapa ratus entri yang terlambat; (c) 10^8 pembacaan sensor 32-bit; (d) rekaman pelanggan yang harus mempertahankan urutan sebelumnya di antara kunci yang sama; (e) sistem waktu nyata yang kasus terburuknya harus $O(n \lg n)$ dan tidak ada memori tambahan.

3.19 Temukan tiga metode pengurutan yang tidak dibahas dalam bab ini. Untuk masing-masing, uraikan gagasannya, tuliskan dalam pseudokode, dan berikan waktu jalan terbaik, rata-rata, dan terburuknya serta apakah ia stabil dan di tempat.

3.19 Penyelesaian Soal Latihan

3.1 Lintasan 1: $5 > 1$ tukar, $5 > 4$ tukar, $5 > 2$ tukar, $5 < 8$: [1,4,2,5,8], 4 perbandingan, 3 pertukaran. Lintasan 2: $1 < 4$, $4 > 2$ tukar, $4 < 5$: [1,2,4,5,8], 3 perbandingan, 1 pertukaran. Lintasan 3: $1 < 2$, $2 < 4$: tidak ada pertukaran, sehingga keluar dini menghentikan pengurutan setelah 2 perbandingan. Total: 9 perbandingan dan 4 pertukaran. Inversinya adalah (5,1), (5,4), (5,2), dan (4,2): empat, sama dengan banyaknya pertukaran, sebagaimana diharuskan Persamaan (3.4).

3.2 Putaran 1 menemukan 1 di indeks 5 dan menukarnya dengan 6: [1,5,4,3,2,6]. Putaran 2 menemukan 2 di indeks 4: [1,2,4,3,5,6]. Putaran 3 menemukan 3 di indeks 3: [1,2,3,4,5,6]. Putaran 4 dan 5 mendapati minimumnya sudah di tempat. Perbandingan: $5 + 4 + 3 + 2 + 1 = 15$; pertukaran: 3. *Bubble sort* pada masukan yang sama melakukan 15 perbandingan dan 15 pertukaran, satu per inversi. Pada masukan terbalik keunggulan *selection sort* dalam hal pertukaran paling besar.

3.3 Kunci 1: $3 > 1$ geser, sampai di depan: 1 perbandingan, 1 pergeseran, [1,3,2,5,4]. Kunci 2: $3 > 2$ geser, $1 \le 2$ berhenti: 2 perbandingan, 1 pergeseran. Kunci 5: $3 \le 5$ berhenti: 1 perbandingan. Kunci 4: $5 > 4$ geser, $3 \le 4$ berhenti: 2 perbandingan, 1 pergeseran. Total 6 perbandingan dan 3 pergeseran. Inversinya (3,1), (3,2), (5,4), sehingga $\text{inv} = 3$, dan Persamaan (3.6) memberikan $4 \le 6 \le 3 + 4 = 7$; hitungan tepatnya $\text{inv} + (n - 1) - 1 = 6$, karena satu kunci (1) berjalan sampai ke depan.

3.4 Kode 3.43 adalah penyelesaian dalam C++; setiap putaran adalah lintasan kiri-ke-kanan yang diikuti lintasan kanan-ke-kiri, dan rentangnya menyusut dari kedua ujung.

Kode 3.43 `cocktail_sort.cpp`: *cocktail shaker sort*, penyelesaian Soal 3.4.

```cpp
// cocktail_sort.cpp - bubble sort dua arah (cocktail shaker sort)
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

void cocktailSort(vector<int>& a) {
    int lo = 0, hi = static_cast<int>(a.size()) - 1;
    long long comparisons = 0;
    bool swapped = true;
    while (swapped && lo < hi) {
        swapped = false;
        for (int j = lo; j < hi; ++j) {  // kiri ke kanan: terbesar ke a[hi]
            ++comparisons;
            if (a[j] > a[j + 1]) { swap(a[j], a[j + 1]); swapped = true; }
        }
        --hi;
        for (int j = hi; j > lo; --j) {  // kanan ke kiri: terkecil ke a[lo]
            ++comparisons;
            if (a[j - 1] > a[j]) { swap(a[j - 1], a[j]); swapped = true; }
        }
        ++lo;
        cout << "  round: " << show(a) << "\n";
    }
    cout << "  comparisons = " << comparisons << "\n";
}

int main() {
    // satu nilai kecil tersangkut di ujung
    vector<int> turtle = {2, 3, 8, 10, 15, 22, 1};
    cout << "Cocktail sort on " << show(turtle) << "\n";
    cocktailSort(turtle);
    return 0;
}
```

Kode 3.44 `CocktailSort.java`: *cocktail shaker sort* dalam Java.

```java
// CocktailSort.java - bubble sort dua arah (cocktail shaker sort)
public class CocktailSort {
    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static void swap(int[] a, int i, int j) {
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }

    static void cocktailSort(int[] a) {
        int lo = 0, hi = a.length - 1;
        long comparisons = 0;
        boolean swapped = true;
        while (swapped && lo < hi) {
            swapped = false;
            for (int j = lo; j < hi; j++) {  // kiri ke kanan: terbesar ke a[hi]
                comparisons++;
                if (a[j] > a[j + 1]) { swap(a, j, j + 1); swapped = true; }
            }
            hi--;
            for (int j = hi; j > lo; j--) {  // kanan ke kiri: terkecil ke a[lo]
                comparisons++;
                if (a[j - 1] > a[j]) { swap(a, j - 1, j); swapped = true; }
            }
            lo++;
            System.out.println("  round: " + show(a));
        }
        System.out.println("  comparisons = " + comparisons);
    }

    public static void main(String[] args) {
        // satu nilai kecil tersangkut di ujung
        int[] turtle = {2, 3, 8, 10, 15, 22, 1};
        System.out.println("Cocktail sort on " + show(turtle));
        cocktailSort(turtle);
    }
}
```

Kode 3.45 `cocktail_sort.py`: *cocktail shaker sort* dalam Python.

```python
# cocktail_sort.py - bubble sort dua arah (cocktail shaker sort)
def show(a):
    return "[" + " ".join(map(str, a)) + "]"


def cocktail_sort(a):
    lo, hi = 0, len(a) - 1
    comparisons = 0
    swapped = True
    while swapped and lo < hi:
        swapped = False
        for j in range(lo, hi):          # kiri ke kanan: terbesar ke a[hi]
            comparisons += 1
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
                swapped = True
        hi -= 1
        for j in range(hi, lo, -1):      # kanan ke kiri: terkecil ke a[lo]
            comparisons += 1
            if a[j - 1] > a[j]:
                a[j - 1], a[j] = a[j], a[j - 1]
                swapped = True
        lo += 1
        print(f"  round: {show(a)}")
    print(f"  comparisons = {comparisons}")


turtle = [2, 3, 8, 10, 15, 22, 1]        # satu nilai kecil tersangkut di ujung
print(f"Cocktail sort on {show(turtle)}")
cocktail_sort(turtle)
```

Kode 3.44 dan **Kode 3.45** mencetak hal yang sama dengan versi C++:

```
Cocktail sort on [2 3 8 10 15 22 1]
  round: [1 2 3 8 10 15 22]
  round: [1 2 3 8 10 15 22]
  comparisons = 18
```

Putaran pertama sudah mengurutkan larik: lintasan kanan-ke-kiri membawa 1 dari ujung ke depan dalam satu sapuan. Putaran kedua tidak menemukan apa pun untuk ditukar dan berhenti, dengan total 18 perbandingan. *Bubble sort* memerlukan enam lintasan, karena 1 bergerak satu tempat per lintasan, dan melakukan $6 + 5 + 4 + 3 + 2 + 1 = 21$ perbandingan. Keduanya melakukan 6 pertukaran yang sama, satu per inversi: *cocktail sort* tetap pengurutan bertukar-tetangga dan tetap kuadratik pada kasus terburuk dan rata-rata.

3.5 *Bubble sort* mengubah urutan relatif dua elemen hanya ketika menukar keduanya, dan ia hanya menukar tetangga dengan `A[j] > A[j+1]`. Dua kunci yang sama tidak pernah berada dalam hubungan itu, sehingga keduanya tidak pernah ditukar satu dengan yang lain. Mungkinkah keduanya saling melewati dengan cara lain? Dua elemen hanya dapat bertukar urutan relatif dengan ditukar satu sama lain, karena setiap pertukaran melibatkan tepat dua tetangga. Jadi kunci yang sama mempertahankan urutan relatifnya: pengurutan itu stabil. Dengan `>=`, masukan $[2_a, 2_b]$ (dua kunci sama, diberi indeks bawah untuk membedakannya) ditukar menjadi $[2_b, 2_a]$, dan lintasan berikutnya menukarnya kembali hanya bila ada lintasan lain yang berjalan; dengan keluar dini dan `>=` penandanya tidak pernah bernilai salah, dan urutan kunci-kunci yang sama pada hasilnya bergantung pada berapa lintasan yang berjalan. Pengurutan itu tidak lagi stabil.

3.6 Putaran 1: $3 < 4$ tukar, [3,4,2,1]; $2 < 3$ tukar, [2,4,3,1]; $1 < 2$ tukar, [1,4,3,2]. Putaran 2: $3 < 4$ tukar, [1,3,4,2]; $2 < 3$ tukar, [1,2,4,3]. Putaran 3: $3 < 4$ tukar. Enam pertukaran, satu untuk setiap perbandingan. *Selection sort* melakukan dua: $4 \leftrightarrow 1$ dan $3 \leftrightarrow 2$. *Exchange sort* membayar satu

pertukaran setiap kali muncul calon yang lebih baik untuk posisi i; *selection sort* hanya mengingat indeksnya dan menukar sekali.

3.7 Gelung itu mengalikan dengan tiga dan menambah satu selama $h < n/3$. Untuk $n = 100$, $n/3 = 33$: h berjalan 1,4,13,40, dan celahnya 40,13,4,1. Untuk $n = 1000$, $n/3 = 333$: h berjalan 1,4,13,40,121,364, dan celahnya 364,121,40,13,4,1.

3.8 $1 < 2$ ambil 1; $4 > 2$ ambil 2; $4 > 3$ ambil 3; $4 < 8$ ambil 4; $7 < 8$ ambil 7; $9 > 8$ ambil 8; $9 < 10$ ambil 9. *Run* kiri habis setelah 7 perbandingan, dan 10,12 disalin. Paling banyak diperlukan $4 + 5 - 1 = 8$ perbandingan: setiap perbandingan mengeluarkan satu elemen, dan elemen terakhir selalu keluar tanpa perbandingan. Kasus terburuk menyelang-nyelingkan kedua *run* sehingga dua elemen terbesar berada di *run* yang berbeda, misalnya [1,3,5,9] dan [2,4,6,8,10]: maka 9 dibandingkan dengan 6, 8, dan 10, dan 8 perbandingan dilakukan sebelum salah satu *run* habis.

3.9 Pivotnya $x = 5$ dan $i = -1$. $j = 0$: $7 > 5$, $i = -1$. $j = 1$: $2 \le 5$, $i = 0$, tukar $A[0], A[1]$: [2,7,9,4,3,8,5]. $j = 2$: $9 > 5$. $j = 3$: $4 \le 5$, $i = 1$: [2,4,9,7,3,8,5]. $j = 4$: $3 \le 5$, $i = 2$: [2,4,3,7,9,8,5]. $j = 5$: $8 > 5$. Akhirnya tukar $A[3]$ dengan pivot: [2,4,3,5,9,8,7], dan partisi mengembalikan 3 setelah 6 perbandingan.

3.10 Dengan partisi Lomuto setiap `A[j] ≤ x` bernilai benar, sehingga i maju bersama j, pivot mendarat di ujung kanan, dan bagian kiri berisi $m - 1$ elemen: kasus terburuk Persamaan (3.10), $n(n-1)/2$ perbandingan dan rekursi n tingkat. Dengan partisi Hoare kedua indeks berhenti pada *setiap* elemen (masing-masing $\ge x$ sekaligus $\le x$), setiap penghentian menghasilkan pertukaran, dan kedua indeks bertemu di tengah, sehingga setiap partisi membelah secara seimbang dan totalnya $\Theta(n \lg n)$, sekitar $n \lg n$ perbandingan ditambah penghentian-penghentiannya. Inilah salah satu alasan pustaka diturunkan dari skema Hoare; perbaikan lain adalah partisi tiga arah menjadi wilayah $<$, $=$, dan $>$.

3.11 $n = 7$, sehingga pembangunan memanggil `SIFT-DOWN` untuk $i = 2,1,0$. $i = 2$: 3 mempunyai anak 9 dan 10; tukar dengan 10: [4,1,10,2,16,9,3]. $i = 1$: 1 mempunyai anak 2 dan 16; tukar dengan 16: [4,16,10,2,1,9,3]; indeks 4 tidak mempunyai anak. $i = 0$: 4 mempunyai anak 16 dan 10; tukar dengan 16: [16,4,10,2,1,9,3]; di indeks 1, 4 lebih besar daripada anak-anaknya 2 dan 1, dan *heap* sudah terbangun.

3.12 Hitungan: `count = [2, 1, 2, 2]` (dua 0, satu 1, dua 2, dua 3). Jumlah kumulatif: `[2, 3, 5, 7]`. Penempatan dari kanan ke kiri: 2 yang terakhir ke indeks 4, 0 yang terakhir ke indeks 1, 1 ke indeks 2, 3 yang terakhir ke indeks 6, 2 yang pertama ke indeks 3, 0 yang pertama ke indeks 0, 3 yang pertama ke indeks 5. Keluaran: [0,0,1,2,2,3,3].

3.13 Menurut digit satuan: [170,90,802,2,24,45,75,66]. Menurut digit puluhan: [802,2,24,45,66,170,75,90] (perhatikan bahwa 802 dan 2, keduanya berdigit puluhan 0, mempertahankan urutan lintasan pertama). Menurut digit ratusan: [2,24,45,66,75,90,170,802].

3.14 Ketika `MERGE` mengambil elemen dari *run* kanan sementara $mid - i$ elemen masih tersisa di *run* kiri, elemen itu lebih kecil daripada setiap elemen tersebut dan datang sesudah semuanya: tepat $mid - i$ inversi, yang semuanya terhapus sekaligus. Menjumlahkan besaran itu selama penggabungan, ditambah inversi di dalam setiap paruh (dihitung secara rekursif), memberikan totalnya. Kerja tambahannya satu penjumlahan per langkah penggabungan, sehingga waktunya tetap $\Theta(n \lg n)$.

Kode 3.46 `count_inversions.cpp`: inversi dihitung dengan *merge sort*, diperiksa terhadap hitungan kuadratik menurut definisi. Penyelesaian Soal 3.14.

```cpp
// count_inversions.cpp - menghitung inversi dengan merge sort dalam O(n log n)
#include <iostream>
#include <vector>
using namespace std;

// Mengurutkan a[lo..hi-1] dan mengembalikan banyak pasangan i < j dengan a[i] >
// a[j].
long long sortCount(vector<int>& a, vector<int>& t, int lo, int hi) {
    if (hi - lo < 2) return 0;
    int mid = lo + (hi - lo) / 2;
    long long inv = sortCount(a, t, lo, mid) + sortCount(a, t, mid, hi);
    int i = lo, j = mid, k = lo;
    while (i < mid && j < hi) {
        if (a[i] <= a[j]) t[k++] = a[i++];
        else {
            inv += mid - i;  // a[j] melompati setiap a[i..mid-1]
            t[k++] = a[j++];
        }
    }
    while (i < mid) t[k++] = a[i++];
    while (j < hi) t[k++] = a[j++];
    for (k = lo; k < hi; ++k) a[k] = t[k];
    return inv;
}

long long bruteForce(const vector<int>& a) {         // pemeriksaan O(n^2)
    long long inv = 0;
    for (size_t i = 0; i < a.size(); ++i)
        for (size_t j = i + 1; j < a.size(); ++j)
            if (a[i] > a[j]) ++inv;
    return inv;
}

int main() {
    vector<vector<int>> tests = {{22, 10, 15, 3, 8, 2}, {2, 3, 8, 10, 15, 22},
                                 {22, 15, 10, 8, 3, 2}, {5, 1, 4, 2, 3}};
    for (vector<int> a : tests) {
        long long slow = bruteForce(a);
        vector<int> t(a.size());
        long long fast = sortCount(a, t, 0, static_cast<int>(a.size()));
        cout << "inversions: merge sort " << fast << ", brute force " << slow
             << "\n";
    }
    return 0;
}
```

Kode 3.47 `CountInversions.java`: penghitungan inversi dalam Java.

```java
// CountInversions.java - menghitung inversi dengan merge sort dalam O(n log n)
public class CountInversions {
    // Mengurutkan a[lo..hi-1] dan mengembalikan banyak pasangan i < j dengan
    // a[i] > a[j].
    static long sortCount(int[] a, int[] t, int lo, int hi) {
        if (hi - lo < 2) return 0;
        int mid = lo + (hi - lo) / 2;
        long inv = sortCount(a, t, lo, mid) + sortCount(a, t, mid, hi);
        int i = lo, j = mid, k = lo;
        while (i < mid && j < hi) {
            if (a[i] <= a[j]) t[k++] = a[i++];
            else {
                inv += mid - i;  // a[j] melompati setiap a[i..mid-1]
                t[k++] = a[j++];
            }
        }
        while (i < mid) t[k++] = a[i++];
        while (j < hi) t[k++] = a[j++];
        for (k = lo; k < hi; k++) a[k] = t[k];
        return inv;
    }

    static long bruteForce(int[] a) {                // pemeriksaan O(n^2)
        long inv = 0;
        for (int i = 0; i < a.length; i++)
            for (int j = i + 1; j < a.length; j++)
                if (a[i] > a[j]) inv++;
        return inv;
    }

    public static void main(String[] args) {
        int[][] tests = {{22, 10, 15, 3, 8, 2}, {2, 3, 8, 10, 15, 22},
                         {22, 15, 10, 8, 3, 2}, {5, 1, 4, 2, 3}};
        for (int[] a : tests) {
            long slow = bruteForce(a);
            long fast = sortCount(a, new int[a.length], 0, a.length);
            System.out.println("inversions: merge sort " + fast
                    + ", brute force " + slow);
        }
    }
}
```

Kode 3.48 `count_inversions.py`: penghitungan inversi dalam Python.

```python
# count_inversions.py - menghitung inversi dengan merge sort dalam O(n log n)
def sort_count(a, t, lo, hi):
    """Urutkan a[lo..hi-1]; hasil: banyak pasangan i < j dengan a[i] > a[j]."""
    if hi - lo < 2:
        return 0
    mid = lo + (hi - lo) // 2
    inv = sort_count(a, t, lo, mid) + sort_count(a, t, mid, hi)
    i, j, k = lo, mid, lo
    while i < mid and j < hi:
        if a[i] <= a[j]:
            t[k] = a[i]
            i += 1
        else:
            inv += mid - i              # a[j] melompati setiap a[i..mid-1]
            t[k] = a[j]
            j += 1
        k += 1
    t[k:hi] = a[i:mid] + a[j:hi]        # run mana pun yang tersisa
    a[lo:hi] = t[lo:hi]
    return inv


def brute_force(a):                     # pemeriksaan O(n^2)
    n = len(a)
    return sum(1 for i in range(n) for j in range(i + 1, n) if a[i] > a[j])


TESTS = ([22, 10, 15, 3, 8, 2], [2, 3, 8, 10, 15, 22],
         [22, 15, 10, 8, 3, 2], [5, 1, 4, 2, 3])
for a in TESTS:
    slow = brute_force(a)
    fast = sort_count(a, [0] * len(a), 0, len(a))
    print(f"inversions: merge sort {fast}, brute force {slow}")
```

Kode 3.46, **Kode 3.47**, dan **Kode 3.48** mencetak:

```
inversions: merge sort 13, brute force 13
inversions: merge sort 0, brute force 0
inversions: merge sort 15, brute force 15
inversions: merge sort 6, brute force 6
```

Ke-13 inversi A adalah yang dihitung dengan tangan pada Subbab 3.2.5; larik terbalik mempunyai semua $6 \cdot 5/2 = 15$.

3.15 Untuk $n = 5$: $5! = 120$ dan $\lg 120 \approx 6.91$, sehingga paling sedikit 7 perbandingan pada kasus terburuk (dan 7 dapat dicapai, dengan metode cerdik dalam [1]). Untuk $n = 1000$: $\lg(1000!) \approx 8530$, sehingga paling sedikit 8530 perbandingan; batas yang lebih kasar pada Persamaan (3.13) memberikan $500 \lg 500 \approx 4483$. Kasus terburuk *merge sort*, $n\lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1 = 10\,000 - 1024 + 1 = 8977$, hanya selisih 6% dari minimum yang sesungguhnya.

3.16 Ambil $[2_a, 2_b, 1]$. Pivotnya 1; $2_a > 1$ dan $2_b > 1$, sehingga i tetap -1, dan pivot ditukar dengan $A[0] = 2_a$: $[1, 2_b, 2_a]$. Kedua kunci yang sama telah bertukar urutan, dan sublarik $[2_b, 2_a]$ sudah "terurut", sehingga larik terurut tetap menyimpan keduanya dalam keadaan terbalik.

3.17 Urutkan menurut berat (terberat dahulu) lebih dahulu, lalu secara stabil menurut nama pelanggan. Pengurutan terakhir menentukan urutan utama (menurut nama), dan kestabilan mempertahankan, di antara pesanan pelanggan yang sama, urutan yang dihasilkan pengurutan pertama (menurut berat). Mengurutkan menurut nama lebih dahulu dan menurut berat terakhir akan menghasilkan daftar yang terurut menurut berat, dengan nama hanya sebagai penentu seri.

3.18 (a) *Insertion sort*: kodenya kecil, tanpa memori tambahan, dan tercepat dari semuanya untuk 20 elemen. (b) *Insertion sort* atau Timsort pustaka: masukannya mempunyai sedikit inversi,

sehingga pengurutan adaptif hampir linear; *merge sort* biasa atau *heap sort* akan mengabaikan urutan yang sudah ada. (c) *Radix sort* berbasis 256, empat lintasan: waktu linear, seperti diperlihatkan **Kode 3.34**, dengan biaya larik kedua berisi 10^8 elemen. (d) Pengurutan stabil: *merge sort*, `std::stable_sort`, `List.sort`, atau `sorted`. (e) *Heap sort*: dijamin $O(n\lg n)$ dan di tempat; *introsort* juga cocok, karena ia beralih ke *heap sort*.

3.19 Banyak jawaban yang mungkin; tiga di antaranya disketsakan di sini. **Comb sort** adalah *bubble sort* dengan celah: bandingkan elemen yang berjarak `gap`, perkecil celahnya dengan faktor sekitar 1.3 pada setiap lintasan, dan tutup dengan *bubble sort* biasa pada celah 1; seperti *Shell sort*, ia menghapus inversi jarak jauh sejak awal; kasus terburuk $O(n^2)$, dalam praktik biasanya mendekati $n\lg n$; di tempat, tidak stabil. **Gnome sort** menjaga satu posisi `p`: bila `A[p-1] ≤ A[p]`, melangkah ke kanan, bila tidak, tukar keduanya dan melangkah ke kiri; ia adalah *insertion sort* yang dikerjakan dengan pertukaran bertetangga, terbaik $O(n)$, rata-rata dan terburuk $O(n^2)$, di tempat dan stabil. **Bucket sort** untuk kunci yang tersebar merata pada [0,1) memasukkan setiap kunci ke salah satu dari n ember menurut `⌊n·key⌋`, mengurutkan setiap ember dengan *insertion sort*, lalu menyambung ember-ember itu; harapan $O(n)$ untuk kunci seragam, kasus terburuk $O(n^2)$ bila semua kunci jatuh ke satu ember, ruang tambahan $O(n)$, stabil bila pengurutan embernya stabil. *Tree sort*, yang disebut pada Subbab 3.2.3, menyisipkan setiap kunci ke pohon pencarian biner lalu membacanya kembali secara berurutan; ia muncul pada Bab 11.

Daftar Pustaka

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

[2] O. Astrachan, "Bubble sort: An archaeological algorithmic analysis," in *Proc. 34th SIGCSE Technical Symposium on Computer Science Education*, 2003, pp. 1–5.

[3] D. L. Shell, "A high-speed sorting procedure," *Communications of the ACM*, vol. 2, no. 7, pp. 30–32, 1959.

[4] M. Ciura, "Best increments for the average case of Shellsort," in *Fundamentals of Computation Theory (FCT 2001)*, LNCS 2138. Berlin, Germany: Springer, 2001, pp. 106–117.

[5] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 2, 6, 7, 8.

[7] J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, 1964.

[8] R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, vol. 7, no. 12, p. 701, 1964.

[9] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011, ch. 2.

[10] cppreference.com, "std::sort." [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/sort (accessed Sep. 2026).

[11] D. R. Musser, "Introspective sorting and selection algorithms," *Software: Practice and Experience*, vol. 27, no. 8, pp. 983–993, 1997.

[12] Free Software Foundation, GNU C++ Library (libstdc++), GCC 13, source file `bits/stl_algo.h` (`__introsort_loop`, `__final_insertion_sort`, `_S_threshold = 16`, `_S_chunk_size = 7`).

[13] cppreference.com, "std::stable_sort." [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/stable_sort (accessed Sep. 2026).

[14] Oracle, "Class Arrays," *Java SE 25 & JDK 25 API Specification*, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Arrays.html (accessed Sep. 2026).

[15] V. Yaroslavskiy, J. Bentley, J. Bloch, and D. Lea, `DualPivotQuicksort.java`, OpenJDK. [Online]. Available: https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/DualPivotQuicksort.java (accessed Sep. 2026).

[16] OpenJDK, `TimSort.java`, and issue JDK-6804124, "Replace 'modified mergesort' in java.util.Arrays.sort with timsort." [Online]. Available: https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/TimSort.java and https://bugs.openjdk.org/browse/JDK-6804124 (accessed Sep. 2026).

[17] Python Software Foundation, "Sorting Techniques," *Python 3.14 Documentation, HOWTOs*. [Online]. Available: https://docs.python.org/3/howto/sorting.html (accessed Sep. 2026).

[18] T. Peters *et al.*, `Objects/listsort.txt`, CPython source repository. [Online]. Available: https://github.com/python/cpython/blob/main/Objects/listsort.txt (accessed Sep. 2026).

[19] J. I. Munro and S. Wild, "Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs," in *Proc. 26th European Symposium on Algorithms (ESA 2018)*, LIPIcs vol. 112, 2018, pp. 63:1–63:16.

[20] T. Peters, "bpo-34561: Switch to Munro & Wild 'powersort' merge strategy," CPython pull request #28108, Sep. 2021 (released in Python 3.11); and S. Wild, "Powersort in official Python 3.11 release." [Online]. Available: https://github.com/python/cpython/pull/28108 and https://www.wild-inter.net/posts/powersort-in-python-3.11 (accessed Sep. 2026).

[21] The PostgreSQL Global Development Group, "Using EXPLAIN," *PostgreSQL 18 Documentation*, sec. 14.1, and source file `src/backend/utils/sort/tuplesort.c`. [Online]. Available: https://www.postgresql.org/docs/current/using-explain.html (accessed Sep. 2026).

Bab 4

Evaluasi dan Pembahasan I: Array, Pencarian, dan Pengurutan, Diselesaikan Langkah demi Langkah

Bab ini tidak menambahkan struktur data yang baru. Sebaliknya, bab ini mengambil semua yang telah diberikan Bab 1 sampai Bab 3 (larik di dalam memori, empat cara mencari, dan keluarga algoritma pengurutan), lalu memakainya untuk menyelesaikan sebelas soal yang utuh. Empat di antaranya adalah soal-soal pertama yang klasik dalam pembelajaran struktur data: mengurutkan kata menurut urutan kamus, menemukan karakter pertama sebuah string yang berulang, mencari berapa banyak uang yang dimiliki seseorang, dan mengurutkan pesan menurut tingkat kepentingannya. Tujuh lainnya baru: menghitung alamat elemen larik dengan tangan, menghitung berapa kali sebuah nilai muncul dalam larik terurut, pencarian interpolasi pada kode buku sebuah perpustakaan, menelusuri tiga algoritma pengurutan dasar lintasan demi lintasan, menggabungkan dua daftar terurut, menghapus nilai dari larik di tempat, dan menemukan dua layanan binatu yang harganya tepat sama dengan nilai sebuah voucher. Setiap soal diselesaikan dari nol, perlahan-lahan, dan selalu dengan urutan yang sama: soal dinyatakan ulang dengan kata-kata sederhana; contoh dikerjakan dengan tangan; rancangan digambar dan ditelusuri di atas kertas; program C++ yang lengkap ditulis; program yang sama ditulis dalam Java dan Python, lengkap dengan keluaran nyata ketiganya; sebuah *penyelesaian keliru yang menggoda* disajikan bersama cara persis kegagalannya; dan sebuah varian *uji diri* diberikan beserta jawabannya. Di akhir bab ini Anda akan memiliki sebuah metode untuk menggarap soal struktur data kecil apa pun, dan sebelas contoh yang dikerjakan tuntas dengan metode itu sebagai pegangan.

4.1 Dari Mana Kita Berangkat

Sebelum menyelesaikan apa pun, mari kita gelar dulu perkakasnya. Setiap soal dalam bab ini meraih beberapa perkakas dari tiga bab pertama, dan menyebutkan perkakas yang Anda perlukan sudah merupakan separuh pekerjaan.

Dari **Bab 1** Anda memiliki meja kerja dan model memori. Anda dapat mengompilasi program C++ dengan `g++`, menjalankan kelas Java dengan `java` dan skrip Python dengan `python3`, lalu membaca apa yang dicetaknya. Yang lebih penting, Anda tahu apa itu *larik* (*array*): sederet kotak berukuran sama yang bersebelahan di dalam memori, bernomor mulai dari 0, sehingga komputer dapat melompat ke kotak nomor i dalam satu langkah dengan menghitung alamatnya. Anda juga telah bertemu larik dua dimensi, yang oleh C++ disimpan baris demi baris.

Dari **Bab 2** Anda memiliki *pencarian* (*searching*). *Pencarian sekuensial* (disebut juga pencarian linear) memeriksa butir satu per satu dari depan dan tidak memerlukan urutan sama sekali; varian *sentinel*-nya menaruh kunci di ujung sehingga gelungnya cukup memakai satu uji. *Pencarian biner* memerlukan data terurut dan membagi dua bagian yang masih harus dicari pada setiap langkah. *Pencarian interpolasi* juga memerlukan data terurut, dan menebak posisi kunci dari nilainya, seperti Anda membuka buku telepon di bagian belakang ketika mencari nama yang berawalan W. Bab 2 juga memperkenalkan bahasa untuk membandingkan algoritma: banyaknya

langkah sebagai fungsi ukuran masukan n, yang ditulis dalam notasi Big-O seperti $O(n)$ dan $O(\log n)$.

Dari **Bab 3** Anda memiliki *pengurutan* (*sorting*). Bubble sort dan exchange sort membandingkan tetangga atau pasangan lalu menukarnya; selection sort berulang kali memilih butir terkecil yang tersisa; insertion sort menumbuhkan bagian terurut di sebelah kiri satu butir demi satu butir; merge sort dan quicksort membagi pekerjaan menjadi dua bagian; dan Anda telah mempelajari kata *stabil* (*stable*), yang menggambarkan pengurutan yang mempertahankan urutan semula butir-butir berkunci sama. Bab 3 ditutup dengan pengamatan bahwa setiap pengurutan berbasis perbandingan memerlukan aturan yang menyatakan mana dari dua butir yang didahulukan, dan bahwa mengganti aturan ini saja sudah mengubah arti "terurut" [1], [2]. Soal 4.1 dan Soal 4.4 tumbuh langsung dari pengamatan itu.

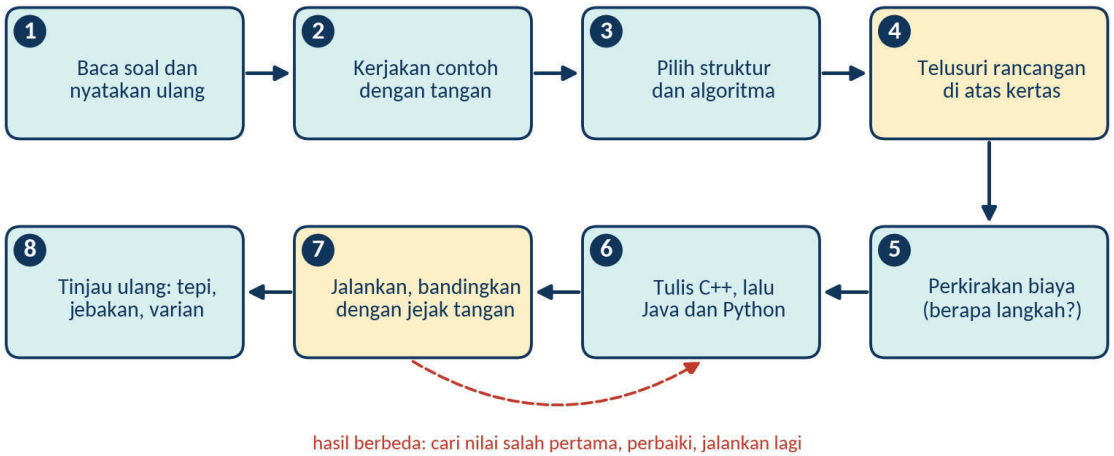
Tabel 4.1 menunjukkan perkakas mana yang paling banyak dilatih oleh setiap soal dalam bab ini. Pakailah tabel ini sebagai peta: jika Anda belum yakin dengan pencarian biner, mulailah dari Soal 4.3 dan Soal 4.6; jika pengurutan masih terasa sekadar hafalan dan belum dipahami, mulailah dari Soal 4.8.

Tabel 4.1 Kesebelas soal dalam bab ini dan perkakas dari Bab 1 sampai Bab 3 yang paling banyak dilatih oleh masing-masing soal.

Soal	Topik	Perkakas utama dari Bab 1–3
4.1	Mengurutkan kata menurut abjad	insertion sort, aturan perbandingan untuk string
4.2	Karakter pertama yang berulang	larik pencacah berindeks karakter, dua lintasan
4.3	Mencari uang menurut nama	larik rekaman, pencarian sekuensial
4.4	Mengurutkan menurut kepentingan	aturan perbandingan dua kunci, kestabilan
4.5	Alamat larik dengan tangan	larik satu dan dua dimensi di dalam memori
4.6	Menghitung kunci dalam larik terurut	pencarian biner untuk sebuah batas
4.7	Pencarian interpolasi pada kode buku	pencarian interpolasi, dibandingkan dengan pencarian biner
4.8	Menelusuri pengurutan dasar	bubble, selection, dan insertion sort, menghitung langkah
4.9	Menggabungkan dua daftar terurut	langkah *merge* dari merge sort
4.10	Menghapus nilai di tempat	penghapusan pada larik, indeks baca dan indeks tulis
4.11	Dua harga yang cocok dengan voucher	pengurutan, lalu jalan dua indeks

4.2 Cara Menggarap Soal Struktur Data

Seorang pemula yang diberi soal biasanya langsung membuka penyunting dan mulai mengetik `#include <iostream>`. Seorang pemrogram berpengalaman biasanya tidak menyentuh papan ketik selama beberapa menit pertama. Bedanya bukan bakat, melainkan kebiasaan kerja. Matematikawan George Pólya menguraikan kebiasaan semacam itu untuk soal matematika (pahami soalnya, susun rencana, jalankan rencana itu, tinjau ulang) [3], dan keempat gerakan yang sama, yang direntangkan menjadi delapan langkah kecil, sama bergunanya bagi pemrogram. **Gambar 4.1** menunjukkan delapan langkah yang kita ikuti untuk setiap soal dalam bab ini.



Gambar 4.1 Delapan langkah yang dipakai untuk setiap soal dalam bab ini; panah putus-putus adalah putaran antara menjalankan program dan memperbaikinya.

4.2.1 Delapan langkah

Langkah 1: Baca soal dan nyatakan ulang. Bacalah pernyataan soal dua kali. Lalu tuliskan, dalam satu atau dua kalimat Anda sendiri, apa yang *masuk* dan apa yang harus *keluar*. Jika Anda tidak sanggup melakukannya, berarti Anda belum memahami soalnya, dan sebanyak apa pun kode tidak akan menolong. Garis bawahi setiap contoh dalam soal; contoh adalah uji pertama bagi program Anda.

Langkah 2: Kerjakan contoh dengan tangan. Ambil contoh-contoh dalam soal dan hitunglah jawabannya dengan pensil dan kertas *sebelum* menulis kode apa pun, sambil mencatat setiap nilai antara. Di sinilah Anda mengetahui apakah Anda sungguh memahami soalnya. Tambahkan satu contoh buatan sendiri di sebuah tepi: masukan kosong, satu butir saja, kunci yang tidak ada, kunci di posisi paling awal atau paling akhir, dua butir yang seri.

Langkah 3: Pilih struktur dan algoritma. Ajukan dua pertanyaan. *Di mana data akan tinggal?* Dalam bab ini jawabannya hampir selalu sebuah larik (atau kerabatnya di Java dan Python), tetapi *bagaimana* larik itu ditata ternyata penting: terurut atau tidak, satu baris atau sebuah kisi, satu larik rekaman atau dua larik sejajar. *Algoritma yang sudah dikenal manakah yang mengerjakan tugas ini, atau hampir mengerjakannya?* Kebanyakan soal adalah algoritma yang sudah dikenal dalam penyamaran. Menemukan penyamaran itu adalah inti langkah ini: "karakter pertama yang berulang" adalah soal mencacah; "berapa kali 2001 muncul" adalah pencarian biner untuk dua batas; "dua harga yang jumlahnya sama dengan voucher" adalah pengurutan yang disusul jalan dari kedua ujung.

Langkah 4: Telusuri rancangan di atas kertas. Gambarlah larik sebagai sederet kotak dengan indeks di bawahnya, lalu gerakkan algoritma melewatinya dengan tangan, satu baris per langkah, sambil mencatat nilai setiap variabel indeks. Setiap soal dalam bab ini memiliki gambar semacam itu. Penelusuran di atas kertas menemukan sebagian besar kesalahan *off-by-one* (meleset satu) sebelum kesalahan itu sempat ada, dan memberi Anda, baris demi baris, nilai-nilai yang harus dihasilkan kembali oleh program pada langkah 7.

Langkah 5: Perkirakan biayanya. Hitunglah, secara kasar, berapa banyak langkah dasar yang diperlukan rancangan itu sebagai fungsi ukuran masukan n. **Tabel 4.2** mengubah hitungan semacam itu menjadi rasa tentang waktu. Rencana yang memerlukan n^2 langkah sama sekali tidak masalah untuk seratus kata dan mustahil untuk sepuluh juta harga; rencana yang

memerlukan $n\log_2 n$ langkah baik untuk keduanya. Ketika sebuah soal menyatakan "penyelesaian optimalnya $O(n)$", soal itu sedang memberi tahu baris tabel mana yang harus Anda tuju.

Langkah 6: Tulis kodenya, C++ lebih dulu. Tulislah program dalam potongan-potongan kecil: data lebih dulu, dikompilasi dan dicetak; lalu satu fungsi, dikompilasi dan diuji; lalu fungsi berikutnya. Program yang tumbuh dalam langkah-langkah kecil yang teruji tidak akan pernah menyodorkan lima puluh galat sekaligus. Setelah versi C++ berjalan, terjemahkan ke Java dan Python. Penerjemahan itu sendiri adalah pemeriksaan yang berguna: baris yang tidak dapat Anda terjemahkan sering kali adalah baris yang belum sepenuhnya Anda pahami.

Langkah 7: Jalankan, dan bandingkan dengan jejak tangan. Jalankan program pada contoh-contoh dari langkah 2. Ketika sebuah hasil berbeda, jangan menebak. Cetaklah variabel-variabel pada setiap langkah (atau jalankan program langkah demi langkah dalam *debugger*) dan bandingkan dengan jejak tangan Anda sampai Anda menemukan nilai *pertama* yang berbeda. Perbedaan itulah kesalahannya. Perbaiki, lalu jalankan semua contoh sekali lagi; inilah panah putus-putus pada **Gambar 4.1**.

Langkah 8: Tinjau ulang. Setelah semuanya cocok, ajukan tiga pertanyaan. Masukan mana yang belum saya coba? Apa cara keliru yang paling *menggoda* untuk menyelesaikan soal ini, dan mengapa cara itu gagal? Dapatkah saya menyelesaikan versi yang sedikit berbeda? Ketiga pertanyaan inilah yang menjadi tiga bagian terakhir setiap soal dalam bab ini.

> **MENGAPA JEJAK DIDAHULUKAN DARIPADA KODE.** Sebuah program hanya dapat sebenar pemahaman Anda terhadap soalnya. Jika Anda tidak sanggup menggerakkan indeks-indeks pencarian biner melewati sembilan kotak di atas kertas, Anda juga tidak sanggup memeriksa program yang mengaku melakukannya. Setiap keluaran yang diharapkan dalam bab ini lebih dulu dihitung dengan tangan pada bagian "Memahaminya dengan tangan" dan "Merancangnya di atas kertas" dari soal yang bersangkutan.

4.2.2 Memperkirakan biaya sebelum menulis kode

Langkah 5 layak dibahas lebih dekat, karena langkah inilah yang paling sering dilewati pemula dan yang paling menentukan begitu data bertambah besar. Komputer modern mengerjakan kira-kira 10^8 sampai 10^9 langkah sederhana (sebuah penjumlahan, sebuah perbandingan, satu kali lewat badan gelung yang pendek) per detik. Itu hanya orde besaran, tetapi orde besaran sudah cukup bagi kita. **Tabel 4.2** menerapkannya pada laju pertumbuhan dari Bab 2 dan Bab 3.

Tabel 4.2 Perkiraan kasar banyak langkah untuk laju pertumbuhan dari Bab 2 dan Bab 3; dengan sekitar 10^8 sampai 10^9 langkah sederhana per detik, 10^{12} langkah memakan waktu bermenit-menit dan 10^{16} langkah memakan waktu bertahun-tahun.

Langkah yang diperlukan	n=100	n=10^4	n=10^6	n=10^8
$\log_2 n$ (pencarian biner)	7	14	20	27
n (satu lintasan)	10^2	10^4	10^6	10^8
$n\log_2 n$ (merge sort)	sekitar 700	sekitar $1.3 \cdot 10^5$	sekitar $2 \cdot 10^7$	sekitar $2.7 \cdot 10^9$
n^2 (dua gelung bersarang)	10^4	10^8	10^{12}	10^{16}

Dua hitungan muncul berulang kali dalam bab ini, jadi kita memberinya nama. Yang pertama adalah banyaknya perbandingan yang dilakukan dua gelung bersarang ketika gelung dalam berjalan atas butir-butir di sebelah kanan butir gelung luar, seperti pada bubble sort, selection sort, dan pencarian pasangan secara *brute force*. Untuk n butir, gelung dalam berjalan $n - 1$ kali,

lalu $n-2$ kali, dan seterusnya sampai sekali, sehingga totalnya adalah jumlah pada Persamaan (4.1).

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2} \tag{4.1}$$

Mengapa jumlah itu sama dengan $\frac{n(n-1)}{2}$? Tulislah jumlah itu dua kali, sekali maju dan sekali mundur, lalu jumlahkan kedua baris itu kolom demi kolom: $(n-1)+1$, $(n-2)+2$, dan seterusnya; setiap kolom berjumlah n, dan ada $n-1$ kolom. Jadi jumlah yang digandakan itu $n(n-1)$, dan jumlah aslinya separuhnya. Untuk $n=7$ nilainya $\frac{7\cdot 6}{2}=21$, bilangan yang akan kita jumpai pada Soal 4.8, dan untuk $n=8$ nilainya 28, yang muncul lagi pada Soal 4.11.

Hitungan kedua adalah banyaknya pemeriksaan (*probe*) pencarian biner. Setiap pemeriksaan sedikitnya membagi dua bagian larik yang masih mungkin memuat kunci, sehingga setelah k pemeriksaan tersisa paling banyak $\frac{n}{2^k}$ butir. Pencarian harus berakhir ketika yang tersisa kurang dari satu butir, sehingga banyaknya pemeriksaan paling banyak sebesar bilangan pada Persamaan (4.2).

$$\lfloor \log_2 n \rfloor + 1 \tag{4.2}$$

Lambang $\lfloor x \rfloor$ berarti "x dibulatkan ke bawah menjadi bilangan bulat", dan $\log_2 n$ adalah pangkat yang harus dikenakan pada 2 agar menghasilkan n; jadi $\log_2 8=3$ karena $2^3=8$. Untuk $n=8$ kode buku, pencarian biner melakukan paling banyak $\lfloor \log_2 8 \rfloor + 1 = 4$ pemeriksaan, dan untuk sejuta butir paling banyak 20. Itulah seluruh alasan mengapa data terurut layak dimiliki.

4.2.3 Susunan setiap soal, dan tempat kodenya

Setiap subbab soal memiliki tujuh bagian yang sama, agar Anda selalu tahu sedang berada di mana:

1. **Soalnya**: pernyataan soal, di dalam kotak.
2. **Memahaminya dengan tangan**: soal dinyatakan ulang dengan kata-kata sederhana dan contoh-contohnya dihitung di atas kertas (langkah 1 dan 2).
3. **Merancangnya di atas kertas**: pilihan struktur dan algoritma, gambar yang ditelusuri dengan tangan, pseudokodenya, dan biayanya (langkah 3 sampai 5).
4. **Penyelesaian C++**: program lengkap, apa yang dicetaknya, dan pembahasan baris-baris pentingnya (langkah 6 dan 7).
5. **Penyelesaian Java dan Python**: rancangan yang sama dalam dua bahasa lainnya; ketiga program mencetak keluaran yang persis sama, bita demi bita.
6. **Penyelesaian keliru yang menggoda**: program yang tampak benar, keluaran nyatanya, dan mengapa ia gagal (langkah 8).
7. **Uji diri**: sebuah varian soal, beserta jawabannya.

Semua program ada di folder kode bab ini: `Chapter04/cpp`, `Chapter04/java`, dan `Chapter04/python`, masing-masing dengan subfolder `tempting` yang berisi penyelesaian-penyelesaian keliru yang menggoda, serta folder bersama `Chapter04/input` yang berisi berkas-berkas masukan. Program C++ dikompilasi dengan `g++ -std=c++17 -O2 -Wall -Wextra` tanpa satu peringatan pun dan juga dijalankan di bawah AddressSanitizer dan UndefinedBehaviorSanitizer (`-fsanitize=address,undefined`), yang melaporkan setiap operasi

baca atau tulis di luar larik. Program Java dikompilasi dengan `javac` 21 dan program Python dijalankan dengan Python 3.11; keduanya juga berjalan tanpa perubahan pada JDK 25 dan Python 3.14. Program yang membaca masukan dijalankan dengan berkas masukan yang dialihkan, misalnya `./sort_words < ../input/words.txt` di Linux dan macOS, atau `Get-Content ..\input\words.txt | .\sort_words.exe` di Windows PowerShell; README folder itu mencantumkan perintah untuk setiap program.

> **SATU ATURAN UNTUK KESEBELAS SOAL.** Setiap program dalam bab ini memakai larik biasa serta pencarian dan pengurutan yang ditulis sendiri, bahkan ketika pustaka standar sudah menyediakan fungsi siap pakai. Ini disengaja: tujuannya adalah melihat struktur data itu bekerja. Bagian C++ setiap soal ditutup dengan menyebutkan fungsi pustaka yang mengerjakan tugas yang sama (`std::sort`, `std::lower_bound`, `std::merge`, dan seterusnya), agar Anda tahu apa yang dapat diraih setelah memahami apa yang dikerjakannya.

4.3 Soal 4.1: Mengurutkan Kata Menurut Abjad

4.3.1 Soalnya

> **SOAL 4.1.** Create a program that sorts the words in alphabetical order.

Dengan kata lain: buatlah program yang mengurutkan kata-kata menurut abjad.

Itulah seluruh pernyataan soalnya: satu kalimat, tanpa contoh, tanpa format masukan. Pernyataan yang pendek justru paling berbahaya, karena setiap keputusan yang tidak diambil oleh soal harus Anda ambil sendiri, dan harus Anda tuliskan. Di sini kita mengambil tiga keputusan. *Masukan:* bilangan pertama n menyatakan berapa kata yang menyusul, lalu datang n kata yang dipisahkan spasi atau pergantian baris; sebuah kata tidak memuat spasi. *Keluaran:* kata-kata itu sekali sebagaimana dibaca, lalu kata-kata yang sudah terurut, satu per baris. *Urutan:* "menurut abjad" berarti urutan kamus, yang menganggap *apple* dan *Apple* sebagai kata yang sama dan menaruh keduanya sebelum *banana*; bila dua kata hanya berbeda pada huruf kapitalnya, kata yang berhuruf kapital kita dahulukan, agar hasilnya selalu sama.

4.3.2 Memahaminya dengan tangan

Dengan kata-kata sederhana: kita menerima sebuah daftar kata dan harus mencetaknya menurut urutan kamus. Contoh kita adalah berkas masukan `words.txt`:

```
8
mango Banana apple Cherry durian banana Apple grape
```

Dengan tangan, tanpa memedulikan huruf kapital, urutan kamus kedelapan kata itu adalah: apple, Apple, banana, Banana, Cherry, durian, grape, mango. Dua pasang kata seri bila huruf kapital diabaikan (*apple*/*Apple* dan *banana*/*Banana*), dan keputusan ketiga kita mendahulukan kata berhuruf kapital pada setiap pasangan. Maka jawaban yang diharapkan adalah

```
Apple apple Banana banana Cherry durian grape mango
```

Mengapa kita memerlukan keputusan tentang huruf kapital? Karena komputer tidak membandingkan huruf; ia membandingkan bilangan. Setiap karakter disimpan sebagai bilangan bulat kecil yang disebut *kode karakter*, dan dalam kode ASCII yang disepakati bersama oleh C++, Java, dan Python untuk huruf-huruf ini, huruf kapital `A` sampai `Z` berkode 65 sampai 90 dan huruf kecil `a` sampai `z` berkode 97 sampai 122 [4]. Ketika sebuah program membandingkan dua string dengan `<` biasa, ia membandingkan karakter pertama keduanya menurut kodenya; jika sama, karakter kedua; dan seterusnya. Urutan ini disebut urutan *leksikografis*. Karena setiap huruf kapital berkode lebih kecil daripada setiap huruf kecil, `<` biasa menyatakan bahwa `"Cherry" <`

`"apple"`, sebab 67 lebih kecil daripada 97. Itu bukan urutan kamus, dan itulah jebakan Subbab 4.3.6.

Contoh tepi buatan kita sendiri: daftar kosong ($n = 0$) tidak mencetak apa pun setelah `Sorted:`; daftar berisi satu kata mencetak kata itu; daftar yang sudah terurut harus tetap seperti semula.

4.3.3 Merancangnya di atas kertas

Strukturnya adalah larik string, `words[0..n-1]`. **Algoritmanya** adalah pengurutan mana pun dari Bab 3, dengan satu perubahan: uji "apakah `a` sebelum `b`?" tidak lagi memakai `<` biasa, melainkan sebuah fungsi kecil buatan kita, `comesBefore(a, b)`. Aturannya, jika dituliskan:

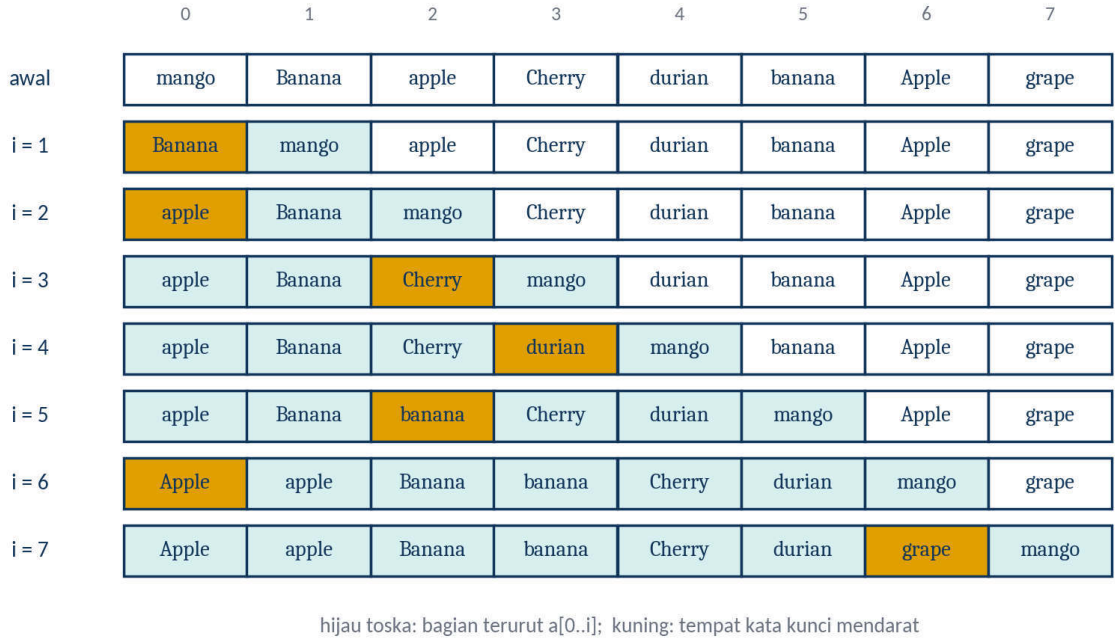
1. Buat salinan `a` dan `b` yang setiap huruf kapitalnya diubah menjadi huruf kecil (dalam ASCII, tambahkan $97 - 65 = 32$ pada kodenya).
2. Jika kedua salinan berbeda, `a` didahulukan tepat ketika salinannya lebih kecil.
3. Jika kedua salinan sama, kedua kata paling banyak berbeda pada huruf kapital; maka `a` didahulukan tepat ketika `a < b` menurut urutan kode biasa, yang menaruh `Apple` sebelum `apple`.

Untuk pengurutannya kita memilih *insertion sort*, karena tiga alasan: pendek, stabil (Soal 4.4 akan memerlukannya), dan cepat pada daftar yang hampir terurut. **Pseudokode 4.1** adalah pseudokodenya; satu-satunya yang membedakannya dari insertion sort di Bab 3 adalah nama perbandingan pada baris 4.

Pseudokode 4.1 Insertion sort untuk kata; perbandingan COMES-BEFORE mewujudkan urutan kamus.

```
INSERTION-SORT-WORDS(words, n)
1  for i = 1 to n - 1
2      key = words[i]
3      j = i - 1
4      while j >= 0 and COMES-BEFORE(key, words[j])
5          words[j + 1] = words[j]        // geser satu tempat ke kanan
6          j = j - 1
7      words[j + 1] = key
```

Gambar 4.2 menelusuri algoritma itu pada kedelapan kata kita. Baris `i = 3`, misalnya, berangkat dari `apple Banana mango Cherry …`: kuncinya `Cherry`; ia didahulukan dari `mango` (karena *cherry* < *mango*), sehingga `mango` bergeser ke kanan; ia tidak didahulukan dari `Banana` (karena *banana* < *cherry*), sehingga gelung berhenti dan `Cherry` mendarat di indeks 2. Ikutilah setidaknya baris `i = 5` dan `i = 6` dengan pensil: pada baris 6 kunci `Apple` berjalan sampai ke indeks 0, melewati `apple`, karena kedua kata seri dalam huruf kecil dan `Apple` < `apple` menurut urutan kode.



Gambar 4.2 Insertion sort atas delapan kata: setelah langkah i, $i+1$ kata pertama (hijau toska) sudah menurut urutan kamus, dan sel kuning adalah tempat kata kunci mendarat.

Biayanya. Insertion sort melakukan paling banyak $\frac{n(n-1)}{2}$ perbandingan (Persamaan (4.1)), masing-masing membandingkan dua kata yang panjangnya paling banyak L karakter, sehingga pekerjaannya $O(n^2L)$ pada kasus terburuk dan $O(nL)$ pada daftar yang sudah terurut. Untuk beberapa ratus kata itu tidak berarti apa-apa. Untuk sejuta kata kita akan beralih ke pengurutan $O(n\log n)$, dan fungsi perbandingannya tetap persis sama.

4.3.4 Penyelesaian C++

Kode 4.1 adalah program lengkapnya.

Kode 4.1 `sort_words.cpp`: insertion sort dengan perbandingan menurut urutan kamus.

```cpp
// sort_words.cpp - Soal 4.1: urutkan kata menurut abjad (urutan kamus)
#include <iostream>
#include <string>
using namespace std;

const int MAX_WORDS = 100;

// Kata dalam huruf kecil, agar "Banana" dan "banana" tampak sama.
string lowerCopy(const string& w) {
    string s = w;
    for (size_t i = 0; i < s.size(); i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] = s[i] - 'A' + 'a';
        }
    }
    return s;
}

// Benar bila kata a harus berada sebelum kata b di kamus.
bool comesBefore(const string& a, const string& b) {
    string la = lowerCopy(a), lb = lowerCopy(b);
    if (la != lb) {
        return la < lb;          // kata berbeda: urutan kamus
    }
    return a < b;                // huruf sama: kapital dulu
}

// Insertion sort (Bab 3): bagian terurut di kiri tumbuh kata demi kata.
void insertionSort(string words[], int n) {
    for (int i = 1; i < n; i++) {
        string key = words[i];
        int j = i - 1;
        while (j >= 0 && comesBefore(key, words[j])) {
            words[j + 1] = words[j];   // geser kata yang lebih besar ke kanan
            j--;
        }
        words[j + 1] = key;
    }
}
```

Kode 4.1 (Lanjutan) *sort_words.cpp: insertion sort dengan perbandingan menurut urutan kamus.*

```cpp
int main() {
    int n;
    string words[MAX_WORDS];
    cin >> n;
    if (n < 0 || n > MAX_WORDS) {
        cout << "n must be between 0 and " << MAX_WORDS << '\n';
        return 1;
    }
    for (int i = 0; i < n; i++) {
        cin >> words[i];
    }
    cout << "Read " << n << " words:";
    for (int i = 0; i < n; i++) {
        cout << ' ' << words[i];
    }
    cout << "\nSorted:\n";
    insertionSort(words, n);
    for (int i = 0; i < n; i++) {
        cout << words[i] << '\n';
    }
    return 0;
}
```

Baris-baris pentingnya, satu per satu:

- `lowerCopy` menerima kata itu melalui `const string&`, sebuah referensi yang berjanji tidak akan mengubahnya (Bab 6 membahas referensi secara lengkap), lalu mengembalikan salinan yang sudah diubah. Uji `s[i] >= 'A' && s[i] <= 'Z'` menanyakan "apakah ini huruf kapital?", dan `s[i] - 'A' + 'a'` mengubah, misalnya, `'C'` (67) menjadi $67 - 65 + 97 = 99$, yaitu `'c'`.
- `comesBefore` adalah aturan Subbab 4.3.3 yang ditulis dalam C++. Perhatikan bahwa fungsi ini mengembalikan `la < lb` hanya bila kedua salinan *berbeda*; `return` yang kedua menangani keadaan seri.
- `insertionSort` adalah **Pseudokode 4.1** baris demi baris. Syarat `j >= 0 && comesBefore(key, words[j])` bergantung pada aturan *short-circuit* (hubung-singkat) dari `&&`: ketika `j` sudah menjadi `-1`, sisi kiri bernilai salah dan C++ tidak pernah mengevaluasi `words[-1]`. Tukarlah kedua sisinya, maka program membaca di luar larik.
- `main` memeriksa bahwa n muat dalam larik berukuran `MAX_WORDS` kotak sebelum membaca. Larik biasa berukuran tetap, dan menulis melewati ujungnya tidak ditangkap oleh C++ (Bab 1); pemeriksaan di pintu masuk adalah asuransi yang murah.

Dijalankan dengan masukan dari Subbab 4.3.2, program mencetak:

```
Read 8 words: mango Banana apple Cherry durian banana Apple grape
Sorted:
Apple
apple
Banana
banana
Cherry
durian
grape
mango
```

Inilah persis urutan yang kita hitung dengan tangan. Dalam C++ sehari-hari Anda akan menulis `std::sort(words, words + n, comesBefore)` atau, bila urutan butir yang seri penting, `std::stable_sort` [5]; keduanya menerima `comesBefore` yang sama sebagai argumen ketiga.

4.3.5 Penyelesaian Java dan Python

Kode 4.2 dan **Kode 4.3** mengikuti rancangan yang sama.

Kode 4.2 `SortWords.java`: insertion sort yang sama dalam Java.

```java
// SortWords.java - Soal 4.1: urutkan kata menurut abjad (urutan kamus)
import java.util.Scanner;

public class SortWords {
    // Benar bila kata a harus berada sebelum kata b di kamus.
    static boolean comesBefore(String a, String b) {
        String la = a.toLowerCase(), lb = b.toLowerCase();
        if (!la.equals(lb)) {
            return la.compareTo(lb) < 0;   // kata berbeda: urutan kamus
        }
        return a.compareTo(b) < 0;           // huruf sama: kapital dulu
    }

    // Insertion sort (Bab 3): bagian terurut di kiri terus tumbuh.
    static void insertionSort(String[] words, int n) {
        for (int i = 1; i < n; i++) {
            String key = words[i];
            int j = i - 1;
            while (j >= 0 && comesBefore(key, words[j])) {
                words[j + 1] = words[j];     // geser kata yang lebih besar ke kanan
                j--;
            }
            words[j + 1] = key;
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        String[] words = new String[n];
        for (int i = 0; i < n; i++) {
            words[i] = in.next();
        }
        System.out.print("Read " + n + " words:");
        for (int i = 0; i < n; i++) {
            System.out.print(" " + words[i]);
        }
        System.out.println();
        System.out.println("Sorted:");
        insertionSort(words, n);
        for (int i = 0; i < n; i++) {
            System.out.println(words[i]);
        }
    }
}
```

Kode 4.3 `sort_words.py`: insertion sort yang sama dalam Python.

```python
# sort_words.py - Soal 4.1: urutkan kata menurut abjad (urutan kamus)
import sys


def comes_before(a, b):
    """Benar bila kata a harus berada sebelum kata b di kamus."""
    la, lb = a.lower(), b.lower()
    if la != lb:
        return la < lb          # kata berbeda: urutan kamus
    return a < b                # huruf sama: kapital dulu


def insertion_sort(words):
    """Insertion sort (Bab 3): bagian terurut di kiri terus tumbuh."""
    for i in range(1, len(words)):
        key = words[i]
        j = i - 1
        while j >= 0 and comes_before(key, words[j]):
            words[j + 1] = words[j]     # geser kata yang lebih besar ke kanan
            j -= 1
        words[j + 1] = key


def main():
    tokens = sys.stdin.read().split()
    n = int(tokens[0])
    words = tokens[1:1 + n]
    print(f"Read {n} words:", " ".join(words))
    print("Sorted:")
    insertion_sort(words)
    for w in words:
        print(w)


main()
```

Keduanya mencetak, karakter demi karakter, keluaran program C++. Tiga perbedaan antarbahasa patut diperhatikan. Java membandingkan string dengan `compareTo`, yang mengembalikan bilangan negatif, nol, atau bilangan positif; `a.compareTo(b) < 0` berarti "`a` sebelum `b`", dan `!la.equals(lb)` adalah cara Java menanyakan "apakah isinya berbeda?" (dengan `!=` ia akan menanyakan apakah keduanya *objek* yang berbeda). `toLowerCase` pada Java dan `lower` pada Python menggantikan `lowerCopy` buatan kita; keduanya juga mengenal huruf beraksen dan huruf non-Latin, yang tidak dikenal oleh versi kita yang hanya untuk ASCII. Dan Python membaca seluruh masukan sekaligus dengan `sys.stdin.read().split()`, yang memecah pada campuran spasi dan pergantian baris apa pun. Jalan pintas pustakanya adalah `Arrays.sort(words, comparator)` di Java [6] dan `sorted(words, key=lambda w: (w.lower(), w))` di Python [7]; *key* Python itu membentuk, untuk setiap kata, pasangan (salinan berhuruf kecil, kata asli), dan pasangan dibandingkan lebih dulu menurut unsur pertamanya lalu menurut unsur keduanya, yang persis sama dengan aturan kita.

4.3.6 Penyelesaian keliru yang menggoda

Jalan pintas yang paling menggoda adalah mempertahankan pengurutannya dan membandingkan kata-kata dengan `<` biasa yang sudah disediakan string. **Kode 4.4** melakukan persis itu.

Kode 4.4 `wrong_sort_words.cpp`: insertion sort yang sama dengan `<` biasa pada string.

```cpp
// wrong_sort_words.cpp - penyelesaian keliru yang menggoda untuk Soal 4.1:
// kata dibandingkan dengan < biasa, yang membandingkan kode karakter.
#include <iostream>
#include <string>
using namespace std;

int main() {
    int n;
    string words[100];
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> words[i];
    }
    for (int i = 1; i < n; i++) {          // insertion sort yang sama ...
        string key = words[i];
        int j = i - 1;
        while (j >= 0 && key < words[j]) { // ... tetapi dengan < biasa
            words[j + 1] = words[j];
            j--;
        }
        words[j + 1] = key;
    }
    for (int i = 0; i < n; i++) {
        cout << words[i] << '\n';
    }
    return 0;
}
```

```
Apple
Banana
Cherry
apple
banana
durian
grape
mango
```

Program itu berjalan, dan keluarannya terurut, tetapi menurut kode karakter, bukan urutan kamus: semua kata berhuruf kapital datang lebih dulu, lalu semua kata lainnya, karena setiap huruf kapital berkode lebih kecil daripada setiap huruf kecil. *Cherry* berdiri sebelum *apple*. Perhatikan bahwa kesalahan ini tidak tampak pada masukan yang seluruhnya ditulis dengan huruf kecil; inilah sebabnya langkah 2 meminta contoh buatan sendiri, dan mengapa contoh kita sengaja mencampur huruf kapital.

> **UJI DIRI.** Ubahlah program itu agar mengurutkan menurut urutan kamus *terbalik* (mango lebih dulu). Fungsi mana saja yang Anda ubah, dan apa baris pertama keluaran yang baru setelah `Sorted:`?
>
> *Jawaban.* Hanya `comesBefore` yang berubah: kembalikan `la > lb` bila kedua salinan berbeda, dan `a > b` bila seri. Tidak ada yang berubah di `insertionSort`, karena pengurutan itu hanya pernah bertanya "apakah ini didahulukan dari itu?". Daftar terurutnya diawali `mango` dan diakhiri `apple`, lalu `Apple`, karena membalik seluruh urutan juga membalik aturan pemutus seri.

4.4 Soal 4.2: Karakter Pertama yang Berulang

4.4.1 Soalnya

> **SOAL 4.2.** Find the first duplicate character in a string. *Input:* a string. *Output:* a single character which satisfies the condition. *Example 1.* Input `abbcbbca`, output `a`. *Example 2.* Input `abbcc`, output `b`. *Tip:* the optimal solution is $O(n)$.

Dengan kata lain: temukan karakter duplikat pertama dalam sebuah string. Masukannya sebuah string, keluarannya satu karakter yang memenuhi syarat itu; untuk `abbcbbca` jawabannya `a`, untuk `abbcc` jawabannya `b`, dan penyelesaian optimalnya berjalan dalam waktu linear.

4.4.2 Memahaminya dengan tangan

Pernyataan ini tampak sederhana tetapi menyembunyikan teka-teki sungguhan: apa arti "karakter duplikat pertama"? Ada dua tafsiran yang wajar.

- **Tafsiran A:** berjalanlah dari kiri ke kanan dan berhentilah pada karakter pertama yang *sudah pernah Anda lihat*. Pada `abbcbbca` Anda melihat `a`, lalu `b`, lalu `b` lagi, sehingga jawabannya `b`.
- **Tafsiran B:** di antara karakter-karakter yang muncul *lebih dari sekali*, ambillah karakter yang *kemunculan pertamanya* paling kiri. Pada `abbcbbca` huruf `a` muncul dua kali (di indeks 0 dan 7), `b` empat kali, dan `c` dua kali; yang paling kiri di antaranya adalah `a` di indeks 0, sehingga jawabannya `a`.

Contoh-contohnya menuntaskan pertanyaan itu: Contoh 1 menyatakan `a`, jadi soal itu memaksudkan Tafsiran B. Periksalah Contoh 2 dengan tafsiran ini: pada `abbcc` huruf `a` muncul sekali, `b` dua kali, dan `c` dua kali; karakter paling kiri yang muncul lebih dari sekali adalah `b`, yaitu keluaran yang diharapkan. Tafsiran A juga akan memberi `b` pada Contoh 2, dan hal ini menunjukkan mengapa Anda harus selalu memeriksa *setiap* contoh, bukan hanya yang pertama: Contoh 2 saja tidak mampu membedakan kedua tafsiran.

Dinyatakan ulang dengan kata-kata sederhana: *untuk setiap karakter, hitunglah berapa kali ia muncul di seluruh string; jawabannya adalah karakter paling kiri yang hitungannya sedikitnya 2.* Contoh-contoh buatan kita sendiri, dikerjakan dengan aturan ini:

- `abcdef`: setiap hitungan bernilai 1, jadi tidak ada jawaban. Soal tidak menyatakan apa yang harus dicetak; kita memutuskan untuk mencetak `no repeated character`.
- `Profitina`: hitungannya P 1, r 1, o 1, f 1, i 2, t 1, n 1, a 1; jawabannya `i`. Perhatikan bahwa `P` dan `p` adalah karakter yang berbeda, karena kodenya berbeda.
- `hello world`: h 1, e 1, l 3; jawabannya `l`. Spasi adalah karakter seperti yang lain dan muncul sekali.
- `xyzzyx`: x 2, y 2, z 2; jawabannya `x` menurut Tafsiran B, dan `z` menurut Tafsiran A. Inilah contoh yang paling baik membedakan kedua tafsiran.

4.4.3 Merancangnya di atas kertas

Rancangan yang paling jelas membandingkan setiap karakter dengan setiap karakter sesudahnya: untuk i dari 0 sampai $n-1$, untuk j dari $i+1$ sampai $n-1$, jika `s[i] == s[j]` maka `s[i]` adalah jawabannya. Rancangan itu benar untuk Tafsiran B, karena ia mencoba posisi i dari kiri ke kanan dan berhenti pada i pertama yang memiliki pasangan. Tetapi ia melakukan sampai $\frac{n(n-1)}{2}$ perbandingan (Persamaan (4.1)); untuk string sepanjang sejuta karakter itu sekitar $5 \cdot$

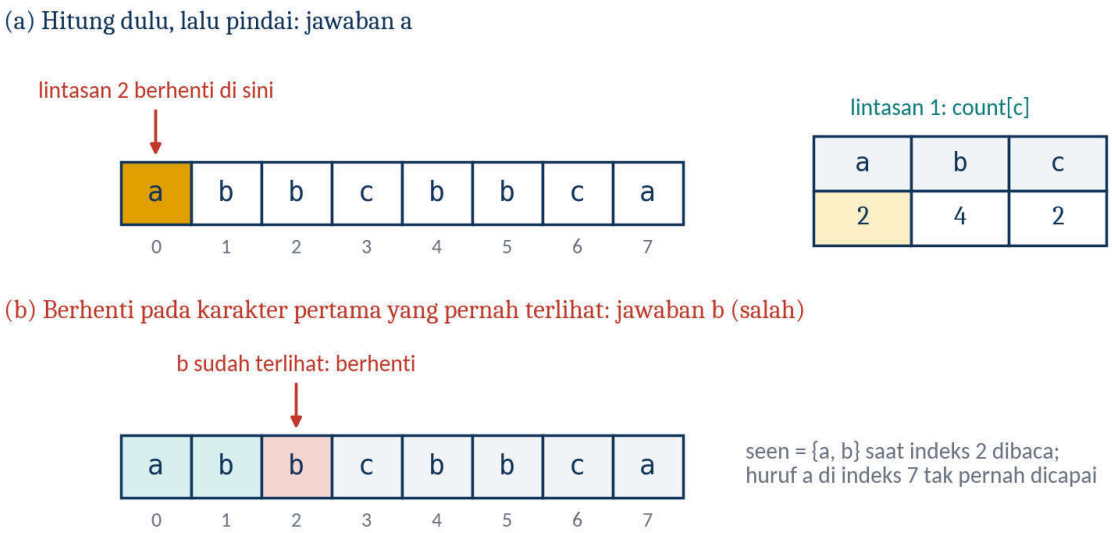
10^{11}, dan **Tabel 4.2** memberi tahu kita bahwa itu berarti bermenit-menit. Petunjuk soal meminta $O(n)$.

Rancangan $O(n)$ menukar sedikit memori dengan banyak waktu. Dalam C++, sebuah karakter adalah satu bita, bilangan dari 0 sampai 255 setelah dibaca sebagai `unsigned char`. Jadi kita dapat menyimpan *larik 256 pencacah*, satu untuk setiap karakter yang mungkin, dan memakai karakter itu sendiri sebagai indeks: `count['a']` adalah pencacah untuk `a`, yaitu `count[97]`. Inilah gagasan terpenting soal ini, yang akan Anda jumpai lagi sebagai *tabel hash* di Bab 15: *nilai yang Anda cari memberi tahu di mana harus mencarinya*, sehingga tidak diperlukan pencarian sama sekali. Rancangan ini memerlukan dua lintasan atas string itu, seperti ditunjukkan **Pseudokode 4.2**.

Pseudokode 4.2 Karakter pertama yang berulang dalam dua lintasan, dengan satu pencacah untuk setiap karakter yang mungkin.

```
FIRST-REPEATED(s)
1  let count[0..255] be a new array, all zeros
2  for i = 0 to length(s) - 1          // lintasan 1: mencacah
3      count[s[i]] = count[s[i]] + 1
4  for i = 0 to length(s) - 1          // lintasan 2: paling kiri dengan hitungan >= 2
5      if count[s[i]] >= 2
6          return i
7  return -1                             // tidak ada karakter yang berulang
```

Gambar 4.3(a) menelusurinya pada `abbcbbca`. Lintasan 1 mengunjungi kedelapan karakter dan meninggalkan `count['a'] = 2`, `count['b'] = 4`, `count['c'] = 2`. Lintasan 2 mulai lagi dari indeks 0, melihat `a`, membaca `count['a'] = 2`, dan langsung berhenti: jawabannya `a`. Bagian (b) gambar itu menunjukkan apa yang akan dilakukan Tafsiran A, yang menjadi penyelesaian keliru yang menggoda di Subbab 4.4.6.



Gambar 4.3 (a) Lintasan 1 mengisi pencacah, lintasan 2 berhenti pada karakter pertama yang pencacahnya sedikitnya 2; (b) berhenti pada karakter pertama yang pernah terlihat memberi jawaban keliru b.

Biayanya. Setiap lintasan memeriksa masing-masing dari n karakter sekali dan mengerjakan sejumlah tetap pekerjaan di sana, dan membuat 256 pencacah juga merupakan pekerjaan dalam jumlah tetap. Totalnya $2n + 256$ langkah, yaitu $O(n)$. Memorinya 256 pencacah, berapa pun panjang string itu: ruang tambahan $O(1)$.

4.4.4 Penyelesaian C++

Kode 4.5 membaca satu string per baris dan menjawab untuk masing-masing.

Kode 4.5 `first_repeat.cpp`: dua lintasan atas string dengan 256 pencacah.

```
// first_repeat.cpp - Soal 4.2: karakter pertama sebuah string yang muncul
// lebih dari sekali, ditemukan dalam waktu O(n) dengan tabel hitungan.
#include <iostream>
#include <string>
using namespace std;

// Mengembalikan posisi karakter pertama yang muncul sedikitnya dua kali,
// atau -1 bila setiap karakter hanya muncul sekali.
int firstRepeated(const string& s) {
    int count[256] = {0};                  // satu pencacah per karakter
    for (size_t i = 0; i < s.size(); i++) {  // lintasan 1: hitung semua
        count[(unsigned char)s[i]]++;
    }
    for (size_t i = 0; i < s.size(); i++) {  // lintasan 2: hitungan >= 2 pertama
        if (count[(unsigned char)s[i]] >= 2) {
            return (int)i;
        }
    }
    return -1;
}

int main() {
    string line;
    while (getline(cin, line)) {
        if (!line.empty() && line.back() == '\r') {
            line.pop_back();               // akhir baris gaya Windows
        }
        int pos = firstRepeated(line);
        if (pos < 0) {
            cout << line << " -> no repeated character\n";
        } else {
            cout << line << " -> " << line[pos] << '\n';
        }
    }
    return 0;
}
```

- `int count[256] = {0};` membuat 256 pencacah dan mengisi semuanya dengan nol. Tanpa `= {0}`, larik lokal dalam C++ berisi apa saja yang kebetulan ada di memori itu sebelumnya (Bab 1), dan hitungannya akan berisi sampah.
- `count[(unsigned char)s[i]]++` adalah jantung program. Pada kebanyakan komputer, `char` biasa bersifat *bertanda* (*signed*), sehingga bita seperti bita pertama huruf é dalam UTF-8 bernilai negatif, dan indeks negatif akan menulis di luar larik. Konversi ke `unsigned char` mengubah setiap bita menjadi bilangan dari 0 sampai 255.
- `getline(cin, line)` membaca satu baris utuh, termasuk spasinya, itulah sebabnya `hello world` tiba sebagai satu string. Dua baris sesudahnya membuang karakter *carriage return* `'\r'` yang dibawa berkas buatan Windows di akhir setiap baris.
- Fungsi itu mengembalikan *posisi*, bukan karakter, agar "tidak ada jawaban" dapat dinyatakan dengan `-1`. Mengembalikan karakter akan memaksa kita memilih suatu karakter untuk berarti "tidak ada", padahal karakter itu bisa saja jawaban yang sesungguhnya.

Dengan berkas masukan `strings.txt`,

```
abbcbbca
abbcc
abcdef
Profitina
hello world
xyzzyx
```

program itu mencetak:

```
abbcbbca -> a
abbcc -> b
abcdef -> no repeated character
Profitina -> i
hello world -> l
xyzzyx -> x
```

Setiap baris cocok dengan hitungan tangan kita, termasuk kedua contoh dalam soal.

4.4.5 Penyelesaian Java dan Python

Kode 4.6 `FirstRepeat.java`: dua lintasan yang sama dalam Java, dengan satu pencacah per `char` Java.

```java
// FirstRepeat.java - Soal 4.2: karakter pertama sebuah string yang muncul
// lebih dari sekali, ditemukan dalam waktu O(n) dengan tabel hitungan.
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class FirstRepeat {
    // Mengembalikan posisi karakter pertama yang muncul sedikitnya dua kali,
    // atau -1 bila setiap karakter hanya muncul sekali.
    static int firstRepeated(String s) {
        int[] count = new int[65536];          // satu pencacah per char Java
        for (int i = 0; i < s.length(); i++) { // lintasan 1: hitung semua
            count[s.charAt(i)]++;
        }
        for (int i = 0; i < s.length(); i++) { // lintasan 2: hitungan >= 2 pertama
            if (count[s.charAt(i)] >= 2) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in =
                new BufferedReader(new InputStreamReader(System.in));
        String line;
        while ((line = in.readLine()) != null) {
            int pos = firstRepeated(line);
            if (pos < 0) {
                System.out.println(line + " -> no repeated character");
            } else {
                System.out.println(line + " -> " + line.charAt(pos));
            }
        }
    }
}
```

Kode 4.7 `first_repeat.py`: dua lintasan yang sama dalam Python, dengan kamus pencacah.

```python
# first_repeat.py - Soal 4.2: karakter pertama sebuah string yang muncul
# lebih dari sekali, ditemukan dalam waktu O(n) dengan tabel hitungan.
import sys


def first_repeated(s):
    """Posisi karakter pertama yang muncul sedikitnya dua kali, atau -1."""
    count = {}                          # karakter -> berapa kali ia muncul
    for ch in s:                        # lintasan 1: hitung setiap karakter
        count[ch] = count.get(ch, 0) + 1
    for i, ch in enumerate(s):          # lintasan 2: hitungan >= 2 pertama
        if count[ch] >= 2:
            return i
    return -1


for line in sys.stdin:
    line = line.rstrip("\r\n")
    pos = first_repeated(line)
    if pos < 0:
        print(f"{line} -> no repeated character")
    else:
        print(f"{line} -> {line[pos]}")
```

Kode 4.6 dan **Kode 4.7** mencetak keenam baris yang persis sama. Versi Java memerlukan 65.536 pencacah, bukan 256, karena `char` Java adalah bilangan 16 bit (satu unit kode UTF-16); lariknya tetap berukuran tetap, sehingga ruang tambahannya tetap $O(1)$, hanya dengan konstanta yang lebih besar. Versi Python sama sekali tidak memakai larik, melainkan *kamus* (*dictionary*) `count` yang memetakan setiap karakter ke hitungannya; `count.get(ch, 0)` berarti "hitungan `ch`, atau 0 bila `ch` belum pernah terlihat". Kamus adalah tabel hash (Bab 15), dan kelebihannya adalah bekerja untuk setiap karakter yang dikenal Python, bukan hanya untuk 256 karakter. Python juga menyediakan `collections.Counter(s)`, yang mengerjakan lintasan 1 dalam satu panggilan [7].

4.4.6 Penyelesaian keliru yang menggoda

Tafsiran A menghasilkan program yang lebih pendek, hanya satu lintasan, dan keliru. **Kode 4.8** menyimpan larik `seen` berisi 256 nilai Boolean dan berhenti pada karakter pertama yang sudah pernah terlihat.

Kode 4.8 `wrong_first_repeat.cpp`: berhenti pada karakter pertama yang sudah pernah terlihat.

```cpp
// wrong_first_repeat.cpp - penyelesaian keliru yang menggoda untuk Soal 4.2:
// berhenti pada karakter pertama yang SUDAH PERNAH TERLIHAT.
#include <iostream>
#include <string>
using namespace std;

int main() {
    string line;
    while (getline(cin, line)) {
        if (!line.empty() && line.back() == '\r') {
            line.pop_back();
        }
        bool seen[256] = {false};
        int pos = -1;
        for (size_t i = 0; i < line.size(); i++) {
            unsigned char c = line[i];
            if (seen[c]) {                  // c sudah muncul sebelumnya: berhenti
                pos = (int)i;
                break;
            }
            seen[c] = true;
        }
        if (pos < 0) {
            cout << line << " -> no repeated character\n";
        } else {
            cout << line << " -> " << line[pos] << '\n';
        }
    }
    return 0;
}
```

```
abbcbbca -> b
abbcc -> b
abcdef -> no repeated character
Profitina -> i
hello world -> l
xyzzyx -> z
```

Empat dari enam jawaban sama dengan program yang benar, termasuk Contoh 2, dan justru itulah yang membuat penyelesaian ini begitu menggoda: jika Anda hanya menguji dengan `abbcc`, program itu lolos. Tetapi pada Contoh 1 ia menjawab `b`, bukan `a`, dan pada `xyzzyx` ia menjawab `z`, bukan `x`, persis seperti yang diramalkan **Gambar 4.3**(b). Program itu menemukan karakter yang kemunculan *kedua*-nya paling dulu, sedangkan soal menanyakan karakter yang kemunculan *pertama*-nya paling dulu. Program satu lintasan tidak mungkin tahu, ketika membaca `a` yang pertama, bahwa masih ada `a` lain yang menunggu di paling ujung; itulah sebabnya rancangan yang benar perlu mencacah semuanya sebelum memutuskan.

Jebakan kedua layak disebut dalam satu kalimat karena begitu lazim dalam C++. Sangat menggoda untuk menulis gelungnya sebagai `for (int i = 0; i <= s.length(); i++)` dan memberi indeks dengan `s[i] - 97`, dengan anggapan string itu hanya berisi huruf kecil. Tanda `<=` membuat gelung mengunjungi satu posisi terlalu banyak, tempat string C++ menyimpan karakter penutup tak kasatmata berkode 0; lalu `0 - 97 = -97` menjadi indeks, dan program menulis ke memori 97 pencacah *sebelum* larik itu. Pada banyak komputer program itu tetap mencetak jawaban yang benar, dan hal itu membuat kesalahannya makin buruk, bukan makin baik. Menjalankannya dengan `-fsanitize=address` langsung melaporkan operasi tulis di luar batas itu, dan inilah alasan setiap program C++ dalam bab ini dijalankan di bawah *sanitizer*.

> **UJI DIRI.** Ubahlah soalnya menjadi “temukan karakter pertama yang muncul *tepat sekali*” (karakter pertama yang tidak berulang). Apa hasil keenam baris masukan itu, dan baris mana dari **Pseudokode 4.2** yang berubah?

Jawaban. Hanya baris 5 yang berubah, menjadi `if count[s[i]] == 1`. Jawabannya: `abbcbbca` tidak memiliki karakter seperti itu (setiap huruf muncul sedikitnya dua kali); `abbcc` memberi `a`; `abcdef` memberi `a`; `Profitina` memberi `P`; `hello world` memberi `h`; `xyzzyx` tidak memilikinya. Biayanya tetap $O(n)$.

4.5 Soal 4.3: Mencari Uang Menurut Nama

4.5.1 Soalnya

SOAL 4.3. Make a program in C++ that can search the amounts of money owned by persons. Print the amount of money if the person exists in the data, and print "Not found" otherwise. *Data:* Puspita 50, Bejo 35, Sarita 20, Pujo 45, Lestari 60. *Example 1.* Input `Pujo`, output `45`. *Example 2.* Input `Komeng`, output `Not found`.

Dengan kata lain: buatlah program C++ yang mencari jumlah uang milik seseorang. Cetak jumlah uangnya bila orang itu ada dalam data, dan cetak "Not found" bila tidak ada; untuk `Pujo` keluarannya `45`, untuk `Komeng` keluarannya `Not found`.

4.5.2 Memahaminya dengan tangan

Dengan kata-kata sederhana: kita memegang tabel kecil berisi lima orang, masing-masing dengan sejumlah uang, dan untuk setiap nama yang ditanyakan kita mencetak jumlahnya, atau `Not found`. Tabel itu adalah datanya; nama-nama yang diketik pengguna adalah *kueri*-nya. Dengan tangan, menjawab `Pujo` berarti menelusurkan jari ke bawah kolom nama: Puspita? bukan. Bejo? bukan. Sarita? bukan. Pujo? ya, jumlahnya 45. Menjawab `Komeng` berarti menelusurkan jari ke bawah kelima baris lalu jatuh dari dasar tabel: `Not found`.

Kita menambahkan tiga kueri buatan sendiri, masing-masing dipilih untuk menguji satu tepi:

- `Lestari` ada di baris *terakhir*, kasus terburuk bagi pencarian dari atas: jawabannya 60 setelah lima perbandingan.
- `Puspita` ada di baris *pertama*, kasus terbaik: 50 setelah satu perbandingan.
- `pujo` dengan `p` kecil tidak ada dalam tabel, karena perbandingan string bersifat persis dan `p` (112) bukan `P` (80). Jawabannya `Not found`. Apakah *seharusnya* demikian adalah pertanyaan bagi pembuat soal; kita mencatat keputusan kita (kecocokan persis) dan memegangnya.

Jadi untuk kelima kueri `Pujo`, `Komeng`, `Lestari`, `pujo`, `Puspita`, jawaban yang diharapkan adalah 45, Not found, 60, Not found, 50.

4.5.3 Merancangnya di atas kertas

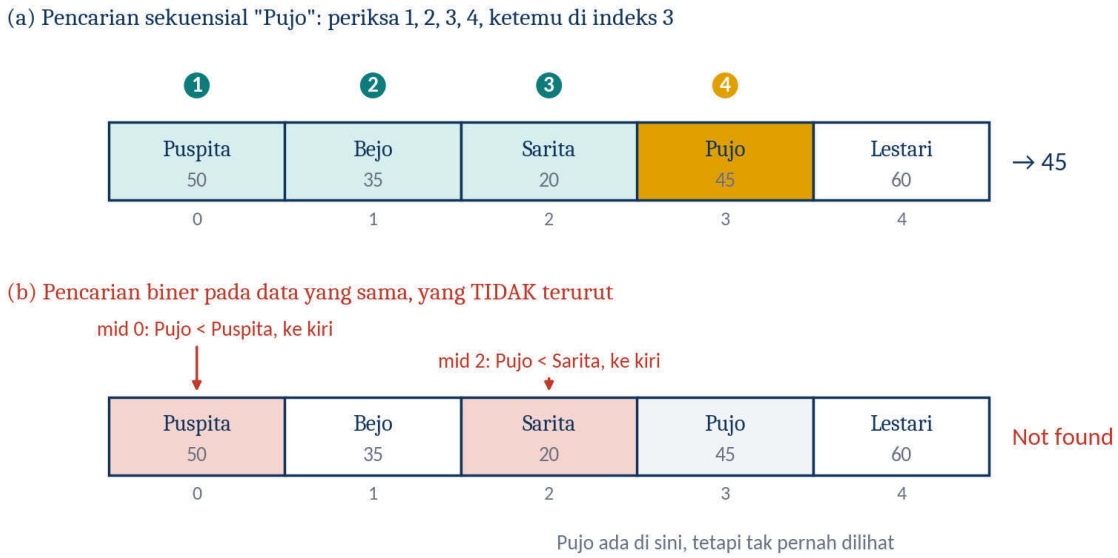
Strukturnya. Setiap orang adalah satu *rekaman* (*record*) dengan dua *field*, sebuah nama dan sebuah jumlah, sehingga kita mendeklarasikan `struct Person { string name; int money; }` dan menyimpan kelima rekaman dalam satu larik, `people[0..4]`. Alternatifnya, dua *larik sejajar* `names[]` dan `money[]` yang indeks i pada satu larik menjadi pasangan indeks i pada larik lainnya, juga dapat dipakai tetapi memiliki kelemahan: jika suatu saat Anda mengurutkan satu larik dan lupa larik yang lain, setiap jumlah uang menjadi milik orang yang salah. Rekaman menjaga kedua nilai tetap bersama dengan sendirinya.

Algoritmanya. Data itu tidak terurut menurut nama (Puspita, Bejo, Sarita, … tidak menurut abjad), sehingga satu-satunya pencarian dari Bab 2 yang berlaku tanpa persiapan adalah *pencarian sekuensial*, seperti pada **Pseudokode 4.3**.

Pseudokode 4.3 Pencarian sekuensial pada larik rekaman; hasilnya sebuah indeks, atau −1 bila nama itu tidak ada.

```
SEQUENTIAL-SEARCH(people, n, name)
1  for i = 0 to n - 1
2      if people[i].name == name
3          return i                  // ketemu: langsung berhenti
4  return -1                         // semua n baris sudah dilihat: tidak ada
```

Gambar 4.4(a) menelusuri pencarian `Pujo`: empat perbandingan, yang keempat berhasil, dan jawabannya `people[3].money`, yaitu 45. Bagian (b) gambar itu milik Subbab 4.5.6.



Gambar 4.4 (a) Pencarian sekuensial menemukan Pujo pada pemeriksaan keempat; (b) pencarian biner pada data yang sama yang tidak terurut membuang bagian yang memuat Pujo dan melaporkan Not found.

Biayanya. Pencarian nama di indeks i memerlukan $i + 1$ perbandingan; pencarian nama yang tidak ada memerlukan n. Jadi satu kueri berbiaya $O(n)$ pada kasus terburuk, dan q kueri berbiaya $O(qn)$. Untuk lima orang dan lima kueri itu paling banyak 25 perbandingan. Untuk bank dengan sejuta nasabah dan sejuta kueri sehari angkanya 10^{12}, dan saat itulah data perlu diurutkan sekali, dalam $O(n \log n)$, agar setiap kueri dapat memakai pencarian biner dalam $O(\log n)$; uji diri subbab ini dan Soal 4.6 menempuh jalan itu.

4.5.4 Penyelesaian C++

Kode 4.9 adalah program lengkapnya.

Kode 4.9 `search_money.cpp`: pencarian sekuensial pada larik rekaman `Person`.

```cpp
// search_money.cpp - Soal 4.3: cari berapa banyak uang milik seseorang
// dengan pencarian sekuensial (Bab 2) pada larik rekaman.
#include <iostream>
#include <string>
using namespace std;

struct Person {
    string name;
    int money;
};

// Mengembalikan indeks orang bernama name, atau -1 bila tidak ada.
int sequentialSearch(const Person people[], int n, const string& name) {
    for (int i = 0; i < n; i++) {
        if (people[i].name == name) {
            return i;                          // ketemu: langsung berhenti
        }
    }
    return -1;                                 // semua n sudah dilihat: tidak ada
}

int main() {
    const int N = 5;
    Person people[N] = {
        {"Puspita", 50}, {"Bejo", 35}, {"Sarita", 20},
        {"Pujo", 45}, {"Lestari", 60}
    };
    string name;
    while (cin >> name) {                      // satu nama per kueri
        int i = sequentialSearch(people, N, name);
        if (i >= 0) {
            cout << people[i].money << '\n';
        } else {
            cout << "Not found\n";
        }
    }
    return 0;
}
```

- `Person people[N] = { {"Puspita", 50}, … };` mengisi larik rekaman dengan *inisialisasi kurung kurawal*: setiap pasangan kurung kurawal di dalam mengisi satu rekaman, *field* demi *field*, menurut urutan deklarasi *field*-nya.
- `sequentialSearch` menerima larik itu sebagai `const Person people[]`. Larik yang dikirim ke fungsi dalam C++ sebenarnya dikirim sebagai alamat kotak pertamanya (Bab 6 menjelaskan alasannya), sehingga fungsi itu tidak dapat mengetahui berapa banyak kotaknya; itulah sebabnya `n` ikut dikirim.
- `people[i].name == name` membandingkan dua objek `std::string` menurut isinya. (Di Java, `==` yang sama akan membandingkan *referensi*; lihat Subbab 4.5.5.)
- `return i;` di dalam gelung adalah jalan keluar dini yang membuat kasus terbaiknya hanya satu perbandingan. Versi yang memasang penanda lalu terus berputar sampai akhir memang benar, tetapi selalu melakukan n perbandingan.
- `while (cin >> name)` terus membaca nama sampai masukan habis, sehingga satu kali jalan dapat menjawab kueri sebanyak apa pun.

Dengan kueri-kueri dari Subbab 4.5.2 dalam berkas `names.txt`,

```
Pujo
Komeng
Lestari
pujo
Puspita
```

program itu mencetak satu jawaban per kueri:

```
45
Not found
60
Not found
50
```

Jawaban-jawaban itu cocok dengan hitungan tangan kita baris demi baris. Ketika data bertambah besar dan kuerinya banyak, pemrogram C++ menyimpan tabel semacam ini dalam `std::map<string, int>` (pohon pencarian seimbang, terurut menurut nama) atau `std::unordered_map<string, int>` (tabel hash), yang menjawab satu kueri dalam $O(\log n)$ langkah dan, rata-rata, $O(1)$ langkah; Bab 11 dan Bab 15 membangun kedua struktur itu dari nol.

4.5.5 Penyelesaian Java dan Python

Kode 4.10 dan **Kode 4.11** memberikan rancangan yang sama dalam Java dan dalam Python.

Kode 4.10 `SearchMoney.java`: pencarian yang sama dengan kelas kecil `Person`.

```java
// SearchMoney.java - Soal 4.3: cari berapa banyak uang milik seseorang
// dengan pencarian sekuensial (Bab 2) pada larik rekaman.
import java.util.Scanner;

public class SearchMoney {
    static class Person {
        String name;
        int money;

        Person(String name, int money) {
            this.name = name;
            this.money = money;
        }
    }

    // Mengembalikan indeks orang bernama name, atau -1 bila tidak ada.
    static int sequentialSearch(Person[] people, String name) {
        for (int i = 0; i < people.length; i++) {
            if (people[i].name.equals(name)) {   // equals, jangan == pada string
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        Person[] people = {
            new Person("Puspita", 50), new Person("Bejo", 35),
            new Person("Sarita", 20), new Person("Pujo", 45),
            new Person("Lestari", 60)
        };
        Scanner in = new Scanner(System.in);
        while (in.hasNext()) {                   // satu nama per kueri
            String name = in.next();
            int i = sequentialSearch(people, name);
            if (i >= 0) {
                System.out.println(people[i].money);
            } else {
                System.out.println("Not found");
            }
        }
    }
}
```

Kode 4.11 `search_money.py`: pencarian yang sama atas list berisi pasangan (nama, uang).

```
# search_money.py - Soal 4.3: cari berapa banyak uang milik seseorang
# dengan pencarian sekuensial (Bab 2) pada list rekaman.
import sys

PEOPLE = [("Puspita", 50), ("Bejo", 35), ("Sarita", 20),
          ("Pujo", 45), ("Lestari", 60)]


def sequential_search(people, name):
    """Indeks orang bernama name, atau -1 bila tidak ada."""
    for i in range(len(people)):
        if people[i][0] == name:
            return i                    # ketemu: langsung berhenti
    return -1                           # semua n sudah dilihat: tidak ada


for name in sys.stdin.read().split():   # satu nama per kueri
    i = sequential_search(PEOPLE, name)
    if i >= 0:
        print(PEOPLE[i][1])
    else:
        print("Not found")
```

Keduanya mencetak kelima baris yang sama dengan program C++. Di Java, **Kode 4.10** mendeklarasikan `Person` sebagai *kelas bersarang* kecil alih-alih `struct`, dan membandingkan nama dengan `people[i].name.equals(name)`. Inilah kekeliruan Java yang paling lazim menyangkut string: `==` antara dua objek `String` menanyakan apakah keduanya *objek yang sama di memori*, dan nama yang dibaca dari papan ketik adalah objek baru, sehingga `==` akan menjawab "berbeda" meskipun hurufnya sama. `==` pada Python membandingkan isi, seperti pada C++. **Kode 4.11** menyimpan setiap rekaman sebagai *tuple* `("Pujo", 45)`, sehingga `people[i][0]` adalah namanya dan `people[i][1]` jumlahnya. Jawaban bawaan Python untuk seluruh soal ini adalah kamus: `money = {"Puspita": 50, …}` disusul `money.get(name, "Not found")` [7].

4.5.6 Penyelesaian keliru yang menggoda

Bab 2 menegaskan bahwa pencarian biner jauh lebih cepat daripada pencarian sekuensial, sehingga sangat menggoda untuk memakainya di sini. **Kode 4.12** memakainya, pada data persis seperti yang diberikan.

Kode 4.12 `wrong_search_money.cpp`: pencarian biner pada data yang tidak terurut menurut nama.

```cpp
// wrong_search_money.cpp - penyelesaian keliru yang menggoda untuk Soal 4.3:
// pencarian biner pada data yang TIDAK terurut menurut nama.
#include <iostream>
#include <string>
using namespace std;

int main() {
    const int N = 5;
    string names[N] = {"Puspita", "Bejo", "Sarita", "Pujo", "Lestari"};
    int money[N] = {50, 35, 20, 45, 60};
    string name;
    while (cin >> name) {
        int lo = 0, hi = N - 1, found = -1;
        while (lo <= hi) {
            int mid = (lo + hi) / 2;
            if (names[mid] == name) {
                found = mid;
                break;
            } else if (name < names[mid]) {
                hi = mid - 1;                // "pasti ada di kiri"
            } else {
                lo = mid + 1;                // "pasti ada di kanan"
            }
        }
        if (found >= 0) {
            cout << money[found] << '\n';
        } else {
            cout << "Not found\n";
        }
    }
    return 0;
}
```

```
Not found
Not found
Not found
Not found
50
```

Hanya `Puspita` yang ditemukan. **Gambar 4.4**(b) menunjukkan mengapa `Pujo` tidak: pemeriksaan pertama melihat bagian tengah, indeks 2, `Sarita`; karena `Pujo` < `Sarita` menurut abjad, pencarian biner menyimpulkan "Pujo pasti ada di kiri" dan membuang indeks 2 sampai 4, termasuk indeks 3 tempat Pujo sebenarnya berada. Kesimpulan itu sahih pada larik yang *terurut*; pada larik ini kesimpulan itu keliru begitu saja. Pemeriksaan kedua, di indeks 0, menarik kesimpulan keliru yang sejenis, dan pencarian berakhir dengan tangan hampa. Program itu tidak berhenti mendadak, tidak memberi peringatan, ia hanya menjawab `Not found`, dan itulah jenis kesalahan yang paling berbahaya. Pencarian biner memiliki prasyarat, dan prasyarat yang tidak terpenuhi tidak menghasilkan pesan galat, hanya jawaban yang salah.

> **UJI DIRI.** Urutkan dulu kelima rekaman menurut nama (dengan insertion sort, membandingkan `people[j].name`), lalu pakailah pencarian biner. Seperti apa larik yang terurut itu, dan berapa pemeriksaan yang dilakukan pencarian biner untuk `Pujo` dan untuk `Komeng`?
>
> *Jawaban.* Terurut menurut nama: Bejo 35, Lestari 60, Pujo 45, Puspita 50, Sarita 20 (rekaman berpindah secara utuh, sehingga setiap jumlah tetap bersama pemiliknya). Untuk `Pujo`: `lo = 0`, `hi = 4`, `mid = 2` adalah `Pujo`, ketemu dengan satu pemeriksaan, jawabannya 45. Untuk `Komeng`: `mid = 2` (Pujo), `Komeng` < `Pujo`, maka `hi = 1`; `mid = 0` (Bejo), `Komeng` > `Bejo`, maka `lo = 1`; `mid = 1` (Lestari), `Komeng` < `Lestari`, maka `hi = 0`; sekarang `lo > hi`, tiga pemeriksaan,

> `Not found`. Keduanya sesuai dengan Persamaan (4.2): paling banyak $\lfloor \log_2 5 \rfloor + 1 = 3$ pemeriksaan.

4.6 Soal 4.4: Mengurutkan Menurut Kepentingan

4.6.1 Soalnya

> **SOAL 4.4.** You are given an array of strings. Your task is to sort these strings based on their importance. The importance of a string is determined by two factors: (1) the number of `'!'` characters it contains, more `'!'` characters implying higher importance; (2) if two strings have the same number of `'!'` characters, the longer string is considered less important. Write a C++ program to sort the array of strings from most to least important. *Input:* `!important`, `not important at all`, `meh`, `Most Important!!!!`, `how about this?`, `important!`. *Output:* `Most Important!!!!`, `!important`, `important!`, `meh`, `how about this?`, `not important at all`.

Dengan kata lain: diberikan sebuah larik string, urutkan string-string itu menurut tingkat kepentingannya. Kepentingan ditentukan oleh dua faktor: (1) banyaknya karakter `'!'`, makin banyak makin penting; (2) bila banyaknya `'!'` sama, string yang lebih panjang dianggap kurang penting. Tulislah program C++ yang mengurutkan larik itu dari yang terpenting ke yang paling tidak penting.

4.6.2 Memahaminya dengan tangan

Dengan kata-kata sederhana: setiap string mendapat *skor* yang terdiri atas dua bilangan, dan kita mengurutkan menurut skor itu. Bilangan pertama adalah banyaknya tanda seru, dan makin banyak makin baik. Bilangan kedua, yang hanya berperan ketika bilangan pertama seri, adalah panjangnya, dan makin pendek makin baik. Kita menghitung keduanya dengan tangan untuk keenam string, secara cermat, karena panjang itu mencakup spasi dan tanda baca:

Tabel 4.3 Kedua kunci setiap string pada Soal 4.4, dihitung dengan tangan.

String	`'!'`	Panjang
`!important`	1	9
`not important at all`	0	20
`meh`	0	3
`Most Important!!!!`	4	18
`how about this?`	0	15
`important!`	1	10

Mengurutkan menurut kunci pertama saja menghasilkan tiga kelompok: string dengan 4 tanda, lalu dua string dengan 1 tanda, lalu tiga string dengan 0 tanda. Di dalam kelompok bertanda 1, `!important` (9 karakter) mengalahkan `important!` (10). Di dalam kelompok bertanda 0, `meh` (3) mengalahkan `how about this?` (15), yang mengalahkan `not important at all` (20). Jika dituliskan, itulah persis keluaran yang diharapkan soal. Kasus tepi buatan kita sendiri: dua string yang seri pada *kedua* kunci, seperti `a!` dan `b!`. Soal tidak menyatakan apa pun tentang keduanya; kita memutuskan bahwa keduanya mempertahankan urutan saat dibaca. Pengurutan yang menjamin hal ini bersifat *stabil* (Bab 3), dan itulah alasan pilihan algoritma kita.

4.6.3 Merancangnya di atas kertas

Ini adalah Soal 4.1 lagi dengan perbandingan yang berbeda, dan itulah inti catatan Bab 3 bahwa sebuah pengurutan hanya memerlukan aturan "mana yang didahulukan". **Tabel 4.3** dan **Gambar 4.5** menunjukkan kedua kunci dan hasilnya. Aturan `moreImportant(a, b)`, yang menyatakan apakah `a` harus berdiri sebelum `b`, adalah:

1. Hitung tanda serunya: c_a dan c_b. Jika berbeda, `a` didahulukan tepat ketika $c_a > c_b$.
2. Jika tidak, `a` didahulukan tepat ketika ia *benar-benar* lebih pendek daripada `b`.

Kata *benar-benar* (ketat) itu penting. Ketika panjangnya juga sama, aturan itu harus menjawab "tidak" untuk kedua arah, `moreImportant(a, b)` maupun `moreImportant(b, a)`, agar insertion sort membiarkan kedua string di tempatnya; itulah yang menjaga pengurutan tetap stabil. **Pseudokode 4.4** adalah insertion sort yang sama dengan **Pseudokode 4.1** dengan aturan yang baru.

Pseudokode 4.4 Aturan perbandingan dua kunci dan insertion sort stabil yang memakainya.

```
MORE-IMPORTANT(a, b)
1  ca = number of '!' in a;  cb = number of '!' in b
2  if ca != cb
3      return ca > cb                  // kunci pertama: tanda lebih banyak dulu
4  return length(a) < length(b)        // kunci kedua: lebih pendek dulu (ketat)

SORT-BY-IMPORTANCE(lines, n)
5  INSERTION-SORT-WORDS(lines, n) with COMES-BEFORE replaced by MORE-IMPORTANT
```

Urutan masukan

string	'!'	panjang
!important	1	9
not important at all	0	20
meh	0	3
Most Important!!!!	4	18
how about this?	0	15
important!	1	10

Terpenting ke tak penting

string	'!'	panjang
Most Important!!!!	4	18
!important	1	9
important!	1	10
meh	0	3
how about this?	0	15
not important at all	0	20

kunci pertama: '!' lebih banyak dulu; kunci kedua (hanya saat seri): lebih pendek dulu

Gambar 4.5 Keenam string dengan kedua kuncinya, menurut urutan masukan (kiri) dan terurut dari yang terpenting ke yang paling tidak penting (kanan); warna menandai tiga kelompok kunci pertama.

Biayanya. Menghitung tanda seru pada string sepanjang L memerlukan L langkah, dan insertion sort memanggil aturan itu sampai $\frac{n(n-1)}{2}$ kali, sehingga program mengerjakan $O(n^2L)$. Pemrogram yang cermat akan menghitung tanda seru setiap string sekali saja, sebelum pengurutan, dan menyimpan hasilnya dalam larik kedua; dengan begitu biaya penghitungan totalnya $O(nL)$. Untuk enam string kita mempertahankan versi yang lebih sederhana.

4.6.4 Penyelesaian C++

Kode 4.13 adalah program lengkapnya.

Kode 4.13 `importance_sort.cpp`: aturan perbandingan dua kunci di dalam insertion sort yang stabil.

```cpp
// importance_sort.cpp - Soal 4.4: urutkan string dari terpenting ke tak penting.
// Lebih banyak '!' berarti lebih penting; bila '!' sama, yang lebih pendek menang.
#include <iostream>
#include <string>
using namespace std;

const int MAX_LINES = 100;

int countBangs(const string& s) {
    int c = 0;
    for (char ch : s) {
        if (ch == '!') {
            c++;
        }
    }
    return c;
}

// Benar bila a LEBIH penting daripada b, jadi a harus di depan.
bool moreImportant(const string& a, const string& b) {
    int ca = countBangs(a), cb = countBangs(b);
    if (ca != cb) {
        return ca > cb;                     // kunci 1: '!' lebih banyak dulu
    }
    return a.size() < b.size();             // kunci 2: lebih pendek dulu
}

// Insertion sort stabil: string yang seri pada kedua kunci tetap urutannya.
void sortByImportance(string lines[], int n) {
    for (int i = 1; i < n; i++) {
        string key = lines[i];
        int j = i - 1;
        while (j >= 0 && moreImportant(key, lines[j])) {
            lines[j + 1] = lines[j];
            j--;
        }
        lines[j + 1] = key;
    }
}
```

Kode 4.13 (Lanjutan) *importance_sort.cpp: aturan perbandingan dua kunci di dalam insertion sort yang stabil.*

```cpp
int main() {
    string lines[MAX_LINES];
    int n = 0;
    string line;
    while (n < MAX_LINES && getline(cin, line)) {
        if (!line.empty() && line.back() == '\r') {
            line.pop_back();
        }
        lines[n++] = line;
    }
    sortByImportance(lines, n);
    for (int i = 0; i < n; i++) {
        cout << lines[i] << '\n';
    }
    return 0;
}
```

- `for (char ch : s)` adalah *range-based for loop* dari C++11 [8]: gelung itu mengunjungi setiap karakter `s` secara bergiliran, tanpa indeks. Inilah perkakas yang tepat ketika indeksnya sendiri tidak diperlukan.
- `moreImportant` mengembalikan `a.size() < b.size()`: ketat, seperti yang dituntut rancangan.

- `main` membaca baris utuh dengan `getline`, karena string-string ini memuat spasi, dan berhenti pada `MAX_LINES` agar larik tidak meluap.

Dengan keenam string dari soal dalam berkas `messages.txt`,

```
!mportant
not important at all
meh
Most Important!!!!
how about this?
important!
```

keluarannya adalah:

```
Most Important!!!!
!mportant
important!
meh
how about this?
not important at all
```

Inilah persis keluaran yang diminta soal. Dalam C++ cara pustakanya adalah `std::stable_sort(lines, lines + n, moreImportant)` [5]. `std::sort` biasa juga akan memberi jawaban yang benar untuk masukan *ini*, karena tidak ada dua string yang seri pada kedua kunci, tetapi ia tidak menjanjikan bahwa butir-butir yang seri tetap dalam urutan masukannya.

4.6.5 Penyelesaian Java dan Python

Kode 4.14 dan **Kode 4.15** memberikan rancangan yang sama dalam Java dan dalam Python.

Kode 4.14 `ImportanceSort.java`: aturan dan pengurutan yang sama dalam Java.

```java
// ImportanceSort.java - Soal 4.4: urutkan string dari terpenting ke tak penting.
// Lebih banyak '!' berarti lebih penting; bila '!' sama, yang lebih pendek menang.
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;

public class ImportanceSort {
    static int countBangs(String s) {
        int c = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '!') {
                c++;
            }
        }
        return c;
    }

    // Benar bila a LEBIH penting daripada b, jadi a harus di depan.
    static boolean moreImportant(String a, String b) {
        int ca = countBangs(a), cb = countBangs(b);
        if (ca != cb) {
            return ca > cb;                     // kunci 1: '!' lebih banyak dulu
        }
        return a.length() < b.length();         // kunci 2: lebih pendek dulu
    }

    // Insertion sort stabil: yang seri pada kedua kunci tetap urutannya.
    static void sortByImportance(String[] lines) {
        for (int i = 1; i < lines.length; i++) {
            String key = lines[i];
            int j = i - 1;
            while (j >= 0 && moreImportant(key, lines[j])) {
                lines[j + 1] = lines[j];
                j--;
            }
            lines[j + 1] = key;
        }
    }
```

Kode 4.14 (Lanjutan) *ImportanceSort.java: aturan dan pengurutan yang sama dalam Java.*

```java
    public static void main(String[] args) throws IOException {
        BufferedReader in =
                new BufferedReader(new InputStreamReader(System.in));
        ArrayList<String> read = new ArrayList<>();
        String line;
        while ((line = in.readLine()) != null) {
            read.add(line);
        }
        String[] lines = read.toArray(new String[0]);
        sortByImportance(lines);
        for (String s : lines) {
            System.out.println(s);
        }
    }
}
```

Kode 4.15 `importance_sort.py`: aturan dan pengurutan yang sama dalam Python.

```python
# importance_sort.py - Soal 4.4: urutkan string dari terpenting ke tak penting.
# Lebih banyak '!' berarti lebih penting; bila '!' sama, yang lebih pendek menang.
import sys


def more_important(a, b):
    """Benar bila a LEBIH penting daripada b, jadi a harus di depan."""
    ca, cb = a.count("!"), b.count("!")
    if ca != cb:
        return ca > cb                  # kunci 1: '!' lebih banyak dulu
    return len(a) < len(b)              # kunci 2: lebih pendek dulu


def sort_by_importance(lines):
    """Insertion sort stabil: yang seri pada kedua kunci tetap urutannya."""
    for i in range(1, len(lines)):
        key = lines[i]
        j = i - 1
        while j >= 0 and more_important(key, lines[j]):
            lines[j + 1] = lines[j]
            j -= 1
        lines[j + 1] = key


lines = [line.rstrip("\r\n") for line in sys.stdin]
sort_by_importance(lines)
for s in lines:
    print(s)
```

Keduanya mencetak keenam baris yang sama. Program Java lebih dulu mengumpulkan baris-baris itu dalam sebuah `ArrayList`, karena ia tidak dapat tahu sebelumnya berapa banyak baris yang akan datang, lalu mengubah list itu menjadi larik dengan `toArray`. Python menghitung tanda seru dengan metode string `count("!")`. Jalan pintas pustakanya bersifat mendidik. `Arrays.sort` pada Java untuk larik objek dijamin stabil [6], sehingga `Arrays.sort(lines, Comparator.comparingInt((String s) -> -countBangs(s)).thenComparingInt(String::length))` mengerjakan seluruh tugasnya. Pengurutan Python juga stabil, dan sebuah *key* yang mengembalikan pasangan `(-s.count("!"), len(s))` membuat Python membandingkan lebih dulu negatif banyaknya tanda seru lalu panjangnya: `sorted(lines, key=lambda s: (-s.count("!"), len(s)))` [7]. Tanda minus mengubah "tanda seru lebih banyak dulu" menjadi "bilangan lebih kecil dulu", yang memang dikerjakan pengurutan naik.

4.6.6 Penyelesaian keliru yang menggoda

Gagasan yang populer adalah mengurutkan dua kali dengan pengurutan pustaka yang stabil: sekali menurut satu kunci dan sekali menurut kunci lainnya. Gagasan itu baik, tetapi hanya dengan urutan yang benar, dan urutan yang menggoda justru urutan yang keliru. **Kode 4.16** mengurutkan menurut tanda seru lebih dulu dan menurut panjang sesudahnya.

Kode 4.16 `wrong_importance_sort.py`: dua pengurutan stabil yang dijalankan dengan urutan keliru.

```
# wrong_importance_sort.py - penyelesaian keliru yang menggoda untuk Soal 4.4:
# dua pengurutan stabil, tetapi urutannya terbalik, jadi kunci TERAKHIR menentukan.
import sys

lines = [line.rstrip("\r\n") for line in sys.stdin]
lines.sort(key=lambda s: -s.count("!"))    # ke-1: menurut banyaknya '!' ...
lines.sort(key=len)                         # ke-2: ... lalu panjang (menang!)
for s in lines:
    print(s)
```

```
meh
!mportant
important!
how about this?
Most Important!!!!
not important at all
```

String-string itu kini terurut menurut panjang, dari `meh` (3) sampai `not important at all` (20), dan tanda seru hanya berperan di antara string yang sama panjangnya. Alasannya, pengurutan stabil mempertahankan urutan sebelumnya hanya *di antara butir yang seri pada kunci yang sedang dipakai*; pengurutan terakhir karena itu menentukan urutan utama, dan pengurutan sebelumnya hanya menentukan di antara yang seri. Jadi aturan pengurutan berlapis adalah: *urutkan menurut kunci yang paling tidak penting lebih dulu dan menurut kunci yang paling penting paling akhir* [7]. Tukarlah kedua baris pada **Kode 4.16**, dengan mengurutkan menurut `len` lebih dulu dan menurut tanda seru sesudahnya, maka keluarannya menjadi benar. (Program C++ di Subbab 4.6.4 menghindari persoalan ini sepenuhnya dengan menggabungkan kedua kunci dalam satu aturan.)

> **UJI DIRI.** Tambahkan dua baris `Hi!` dan `Yo!` di akhir masukan, dengan urutan itu. Di mana keduanya muncul dalam keluaran program yang benar, dan dengan urutan apa?
>
> *Jawaban.* Keduanya memiliki satu tanda seru dan panjang 3, sehingga termasuk kelompok bertanda satu dan lebih pendek daripada `!mportant` (9). Keduanya ditempatkan tepat setelah `Most Important!!!!`: keluarannya diawali `Most Important!!!!`, `Hi!`, `Yo!`, `!mportant`, `important!`. `Hi!` tetap sebelum `Yo!` karena keduanya seri pada kedua kunci dan insertion sort bersifat stabil.

4.7 Soal 4.5: Alamat Larik dengan Tangan

Empat soal sejauh ini berasal dari sebuah kumpulan soal klasik. Tujuh soal berikutnya baru, dan masing-masing mengisolasi satu gagasan dari Bab 1 sampai Bab 3 yang telah dipakai keempat soal pertama tanpa diamati dari dekat. Kita mulai dari dasar, dari memori itu sendiri.

4.7.1 Soalnya

> **SOAL 4.5.** Sebuah larik `int A[10]` dimulai pada alamat `0x1000`, demikian pula larik dua dimensi `int B[3][5]`. (a) Jika sebuah `int` menempati 4 bita, berapakah alamat `A[7]` dan `B[2][3]`? (b) Pertanyaan yang sama untuk kompilator lama yang `int`-nya menempati 2 bita. (c) Periksalah bagian (a) terhadap memori nyata komputer Anda.

4.7.2 Memahaminya dengan tangan

Dengan kata-kata sederhana: larik adalah sederet kotak yang lebarnya sama, terletak bersebelahan di dalam memori, dan setiap bita memori memiliki nomor, yaitu *alamat*-nya. Alamat biasanya ditulis dalam *heksadesimal* (basis 16) dengan awalan `0x`; angkanya 0 sampai 9 lalu A sampai F untuk sepuluh sampai lima belas, sehingga `0x1000` adalah $1 \cdot 16^3 = 4096$ dan

`0x1C` adalah $1 \cdot 16 + 12 = 28$. Kita ditanya di mana kotak nomor 7 dari `A`, dan kotak baris 2 kolom 3 dari `B`, dimulai.

Untuk `A`: kotak 0 dimulai di `0x1000`, kotak 1 satu `int` kemudian, di `0x1004`, kotak 2 di `0x1008`, dan secara umum kotak i dimulai i kotak berukuran 4 bita setelah awal larik. Jadi `A[7]` dimulai $7 \cdot 4 = 28$ bita kemudian: $4096 + 28 = 4124$, yaitu `0x101C`. Dengan `int` 2 bita, letaknya $7 \cdot 2 = 14$ bita kemudian, `0x100E`.

Untuk `B`: memori adalah satu deret panjang, bukan kisi, sehingga kisi itu harus dibentangkan menjadi satu deret dengan suatu cara. C++ menyimpannya *baris demi baris*: mula-mula kelima kotak baris 0, lalu kelima kotak baris 1, lalu baris 2. Cara ini disebut *urutan row-major* (utama baris). Untuk mencapai `B[2][3]` kita harus melewati dua baris lengkap berisi 5 kotak (10 kotak), lalu 3 kotak lagi di baris 2, semuanya 13 kotak. Dengan 4 bita per kotak itu $13 \cdot 4 = 52$ bita, sehingga alamatnya $4096 + 52 = 4148$, yaitu `0x1034`. Dengan 2 bita per kotak itu 26 bita, `0x101A`.

4.7.3 Merancangnya di atas kertas

Kedua hitungan tangan itu menjadi dua rumus. Untuk larik satu dimensi dengan alamat pertama b (*base*, basis) dan ukuran elemen s bita, elemen i dimulai pada alamat Persamaan (4.3).

$$\text{addr}(A[i]) = b + i \cdot s \tag{4.3}$$

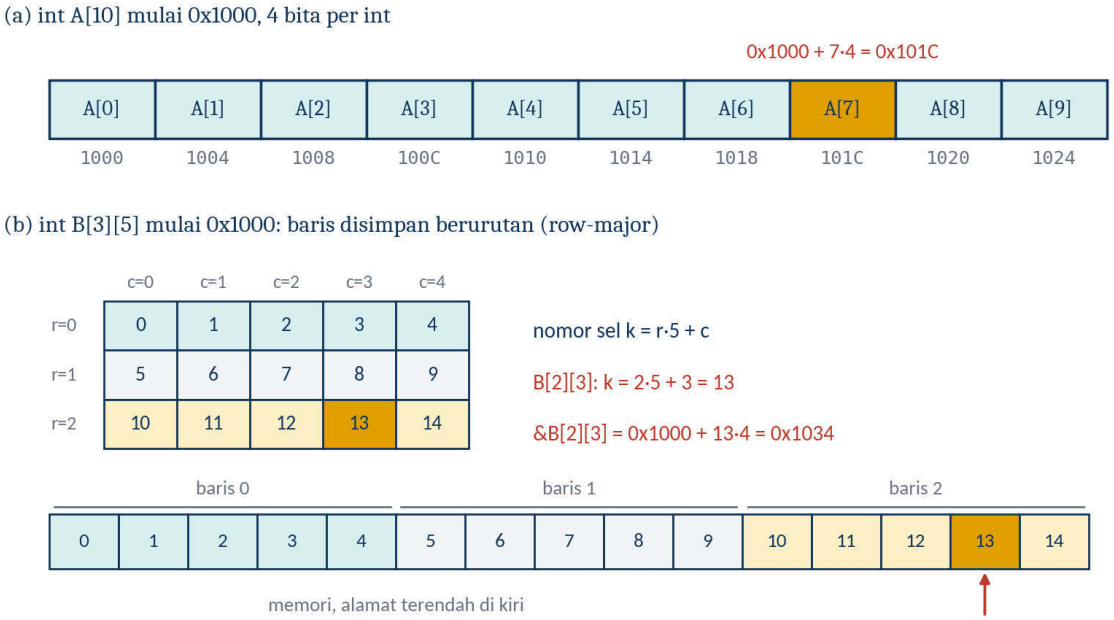
Untuk larik dua dimensi dengan C kolom yang disimpan dalam urutan row-major, elemen (r, c) didahului oleh r baris lengkap berisi C elemen dan oleh c elemen dari barisnya sendiri, yang menghasilkan Persamaan (4.4).

$$\text{addr}(B[r][c]) = b + (r \cdot C + c) \cdot s \tag{4.4}$$

Perhatikan apa yang *tidak* muncul pada Persamaan (4.4): banyaknya baris. Banyaknya baris membatasi seberapa besar r boleh bernilai, tetapi tidak pernah masuk ke dalam alamat. **Gambar 4.6** menggambar kedua larik: bagian (a) menampilkan `A` dengan alamat di bawah setiap kotak, dan bagian (b) memberi nomor kelima belas sel `B` menurut urutan letaknya di memori, sehingga nomor sel $k = r \cdot 5 + c$ dapat langsung dibaca. **Pseudokode 4.5** adalah perhitungannya, yang begitu pendek sehingga nilainya sepenuhnya terletak pada rumus-rumus itu.

Pseudokode 4.5 Alamat elemen larik dalam satu dan dua dimensi (urutan row-major).

```
ADDRESS-1D(b, s, i)
1  return b + i * s
ADDRESS-2D(b, s, C, r, c)
2  return b + (r * C + c) * s        // C = banyaknya kolom
```



Gambar 4.6 (a) Sepuluh kotak A beserta alamatnya; (b) kisi B, sel-selnya diberi nomor menurut urutan di memori, dan sel-sel yang sama sebagaimana sungguh-sungguh terletak di memori.

Biayanya konstan: satu perkalian dan satu penjumlahan dalam satu dimensi, dua dari masing-masing dalam dua dimensi. Lompatan berwaktu konstan ke elemen mana pun inilah sifat *utama* larik, dan alasan larik menjadi struktur data pertama di setiap buku struktur data. Sifat ini pula yang dilepaskan oleh *linked list* di Bab 7.

4.7.4 Penyelesaian C++

Kode 4.17 menghitung kedua jawaban dengan rumus untuk kedua ukuran `int`, lalu bertanya kepada memori nyata komputer.

Kode 4.17 `array_address.cpp`: rumus alamat, diperiksa terhadap jarak di memori nyata.

```cpp
// array_address.cpp - Soal 4.5: alamat sebuah elemen larik, dihitung
// dengan tangan memakai rumus Bab 1 lalu dicocokkan dengan memori nyata.
#include <iostream>
#include <cstdio>
using namespace std;

// Alamat A[i] pada larik satu dimensi yang mulai di base.
long address1D(long base, int size, int i) {
    return base + (long)i * size;
}

// Alamat B[r][c] pada larik dua dimensi dengan cols kolom,
// disimpan baris demi baris (urutan row-major, seperti C++).
long address2D(long base, int size, int cols, int r, int c) {
    return base + ((long)r * cols + c) * size;
}

int main() {
    long base = 0x1000;
    for (int size = 4; size >= 2; size -= 2) {
        printf("int of %d bytes, both arrays starting at 0x%lX\n", size, base);
        printf("  &A[7]    = 0x%lX + 7 * %d = 0x%lX\n",
               base, size, address1D(base, size, 7));
        printf("  &B[2][3] = 0x%lX + (2 * 5 + 3) * %d = 0x%lX\n",
               base, size, address2D(base, size, 5, 2, 3));
    }

    // Pertanyaan yang sama diajukan kepada memori nyata komputer ini.
    int A[10];
    int B[3][5];
    long dA = (long)((char*)&A[7] - (char*)&A[0]);         // jarak dalam bita
    long dB = (long)((char*)&B[2][3] - (char*)&B[0][0]);
    printf("On this computer sizeof(int) = %zu\n", sizeof(int));
    printf("  &A[7] - &A[0]       = %ld bytes\n", dA);
    printf("  &B[2][3] - &B[0][0] = %ld bytes\n", dB);
    return 0;
}
```

```
int of 4 bytes, both arrays starting at 0x1000
  &A[7]    = 0x1000 + 7 * 4 = 0x101C
  &B[2][3] = 0x1000 + (2 * 5 + 3) * 4 = 0x1034
int of 2 bytes, both arrays starting at 0x1000
  &A[7]    = 0x1000 + 7 * 2 = 0x100E
  &B[2][3] = 0x1000 + (2 * 5 + 3) * 2 = 0x101A
On this computer sizeof(int) = 4
  &A[7] - &A[0]       = 28 bytes
  &B[2][3] - &B[0][0] = 52 bytes
```

- `printf` dengan `%lX` mencetak sebuah `long` dalam heksadesimal berhuruf kapital, dan `%zu` mencetak hasil `sizeof`, yang bertipe `size_t`.
- Konversi `(long)i * size` membuat perkalian dikerjakan dalam `long`. Pada larik yang sangat besar, aritmetika `int` dapat meluap; pada larik kita tidak, tetapi kebiasaan ini tidak memakan biaya apa pun.
- Bagian terakhir tidak dapat mencetak *alamat* `A[7]` dan `B[2][3]` secara berguna, karena sistem operasi menempatkan larik di alamat yang berbeda setiap kali program dijalankan. Bagian itu mencetak *jarak* dari awal larik: `(char*)&A[7] - (char*)&A[0]`. Konversi ke `char*` membuat pengurangan itu menghitung dalam bita; tanpanya C++ akan menghitung dalam elemen dan mencetak 7 (Bab 6 menjelaskan aritmetika pointer secara lengkap). Jarak yang terukur, 28 dan 52 bita, tepat sama dengan $7 \cdot 4$ dan $13 \cdot 4$, yang menegaskan Persamaan (4.3) dan Persamaan (4.4) serta tata letak row-major.

4.7.5 Penyelesaian Java dan Python

Java dan Python dengan sengaja menyembunyikan alamat, sehingga **Kode 4.18** dan **Kode 4.19** menghitung rumus-rumusnya tetapi tidak dapat melakukan pemeriksaan bagian (c).

Kode 4.18 `ArrayAddress.java`: rumus alamat dalam Java.

```java
// ArrayAddress.java - Soal 4.5: alamat sebuah elemen larik, dihitung
// dengan tangan memakai rumus Bab 1. (Java menyembunyikan alamat nyata.)
public class ArrayAddress {
    // Alamat A[i] pada larik satu dimensi yang mulai di base.
    static long address1D(long base, int size, int i) {
        return base + (long) i * size;
    }

    // Alamat B[r][c] pada larik ber-cols kolom, disimpan baris demi baris.
    static long address2D(long base, int size, int cols, int r, int c) {
        return base + ((long) r * cols + c) * size;
    }

    public static void main(String[] args) {
        long base = 0x1000;
        for (int size = 4; size >= 2; size -= 2) {
            System.out.printf("int of %d bytes, both arrays starting at 0x%X%n",
                    size, base);
            System.out.printf("  &A[7]    = 0x%X + 7 * %d = 0x%X%n",
                    base, size, address1D(base, size, 7));
            System.out.printf("  &B[2][3] = 0x%X + (2 * 5 + 3) * %d = 0x%X%n",
                    base, size, address2D(base, size, 5, 2, 3));
        }
    }
}
```

Kode 4.19 `array_address.py`: rumus alamat dalam Python.

```python
# array_address.py - Soal 4.5: alamat sebuah elemen larik, dihitung
# dengan tangan memakai rumus Bab 1. (Python menyembunyikan alamat nyata.)


def address_1d(base, size, i):
    """Alamat A[i] pada larik satu dimensi yang mulai di base."""
    return base + i * size


def address_2d(base, size, cols, r, c):
    """Alamat B[r][c] pada larik ber-cols kolom, disimpan baris demi baris."""
    return base + (r * cols + c) * size


base = 0x1000
for size in (4, 2):
    print(f"int of {size} bytes, both arrays starting at 0x{base:X}")
    print(f"  &A[7]    = 0x{base:X} + 7 * {size} = "
          f"0x{address_1d(base, size, 7):X}")
    print(f"  &B[2][3] = 0x{base:X} + (2 * 5 + 3) * {size} = "
          f"0x{address_2d(base, size, 5, 2, 3):X}")
```

Keduanya mencetak enam baris pertama keluaran C++:

```
int of 4 bytes, both arrays starting at 0x1000
  &A[7]    = 0x1000 + 7 * 4 = 0x101C
  &B[2][3] = 0x1000 + (2 * 5 + 3) * 4 = 0x1034
int of 2 bytes, both arrays starting at 0x1000
  &A[7]    = 0x1000 + 7 * 2 = 0x100E
  &B[2][3] = 0x1000 + (2 * 5 + 3) * 2 = 0x101A
```

Rumus-rumus itu tetap penting dalam bahasa-bahasa ini. Sebuah `int[]` Java disimpan persis seperti yang dinyatakan Persamaan (4.3), di dalam sebuah objek yang juga mencatat panjangnya; panjang yang tercatat itulah yang memungkinkan Java melempar `ArrayIndexOutOfBoundsException` alih-alih diam-diam membaca memori tetangga, seperti akan kita lihat pada Soal 4.8. Sebaliknya, `int[3][5]` pada Java *bukan* satu blok: ia adalah larik berisi tiga referensi ke tiga baris terpisah yang masing-masing berisi lima, sehingga Persamaan (4.4) hanya berlaku di dalam satu baris. List Python menyimpan referensi ke objek, bukan bilangan itu sendiri; pustaka numerik NumPy menyimpan lariknya sebagai blok sungguhan, row-major secara bawaan, dan memakai persis Persamaan (4.4).

4.7.6 Penyelesaian keliru yang menggoda

Dua kekeliruan cukup lazim sehingga layak mendapat program sendiri, **Kode 4.20**.

Kode 4.20 `wrong_array_address.py`: melupakan ukuran elemen, dan mengalikan dengan banyaknya baris.

```
# wrong_array_address.py - dua cara keliru yang menggoda untuk menghitung alamat.
base, size = 0x1000, 4
# Kesalahan 1: lupa bahwa satu elemen selebar size bita.
print(f"&A[7]    = 0x{base + 7:X}")
# Kesalahan 2: baris dikali banyaknya BARIS (3), bukan kolom (5).
print(f"&B[2][3] = 0x{base + (2 * 3 + 3) * size:X}")
```

```
&A[7]    = 0x1007
&B[2][3] = 0x1024
```

Baris pertama menambahkan *indeks* 7 pada basis, bukan 7 *elemen* berukuran 4 bita; alamat `0x1007` adalah bita terakhir `A[1]`, bukan awal elemen mana pun. Baris kedua mengalikan nomor baris dengan banyaknya baris (3), bukan banyaknya kolom (5), sehingga menghitung $2 \cdot 3 + 3 = 9$ sel, bukan 13, dan mendarat di `B[1][4]`. Pemeriksaan kewajaran yang cepat menangkap keduanya: alamat elemen mana pun dari larik `int` harus berselisih kelipatan 4 dari basisnya, dan `0x1007` tidak demikian; serta `B[2][3]`, yang terletak di baris terakhir, harus sedikitnya dua baris penuh (40 bita) melewati basis, dan `0x1024` (36 bita) tidak demikian.

> **UJI DIRI.** Sebuah `double` menempati 8 bita. Larik `double M[4][6]` dimulai di `0x2000`. Berapakah alamat `M[3][5]`, elemen yang paling akhir, dan berapakah alamat bita pertama *sesudah* larik itu?
>
> *Jawaban.* Menurut Persamaan (4.4), $k = 3 \cdot 6 + 5 = 23$ dan alamatnya $0\text{x}2000 + 23 \cdot 8 = 0\text{x}2000 + 184$; karena $184 = 11 \cdot 16 + 8$, alamat itu `0x20B8`. Larik itu memuat $4 \cdot 6 = 24$ elemen berukuran 8 bita, seluruhnya 192 bita, sehingga bita pertama sesudahnya adalah \$ + 192 = \$ `0x20C0`, yang juga sama dengan `0x20B8` ditambah satu elemen 8 bita, sebagaimana mestinya.

4.8 Soal 4.6: Menghitung Kunci dalam Larik Terurut

4.8.1 Soalnya

> **SOAL 4.6.** Perpustakaan Pustaka Ceria menyimpan tahun terbit buku-buku pada satu rak dalam larik terurut: 1998, 2001, 2001, 2001, 2005, 2010, 2010, 2019, 2023. Untuk tahun tertentu, berapa buku di rak itu yang terbit pada tahun tersebut? Jawablah dalam waktu $O(\log n)$ untuk tahun 2001, 2010, 2002, 2023, 1990, dan 2030.

4.8.2 Memahaminya dengan tangan

Dengan kata-kata sederhana: kita harus menghitung berapa kali sebuah nilai muncul dalam larik terurut, dan kita tidak boleh memeriksa setiap elemen untuk melakukannya. Dengan tangan hitungannya mudah dibaca: 2001 muncul 3 kali, 2010 dua kali, 2023 sekali, dan 2002, 1990, serta 2030 tidak sama sekali.

Pengamatan kuncinya adalah bahwa dalam larik terurut *nilai-nilai yang sama berdiri bersama dalam satu blok*. Ketiga 2001 menempati indeks 1, 2, dan 3. Jadi, alih-alih menghitungnya satu per satu, kita dapat mencari di mana blok itu *dimulai* (indeks 1) dan di mana elemen pertama *sesudah* blok itu (indeks 4), lalu mengurangkan: $4 - 1 = 3$. Cara yang sama berlaku untuk nilai yang tidak muncul: blok 2002 kosong, "dimulai" di indeks 4 (elemen pertama yang tidak lebih kecil dari 2002 adalah 2005, di indeks 4) dan berakhir di indeks 4, dan $4 - 4 = 0$. Untuk 2030, yang lebih besar dari semuanya, kedua batas terletak *lewat ujung*, di indeks 9, yang bukan kotak larik melainkan posisi tepat sesudah kotak terakhir; hitungannya $9 - 9 = 0$.

4.8.3 Merancangnya di atas kertas

Kita memberi nama kedua batas itu. *Batas bawah* (*lower bound*) sebuah kunci adalah indeks pertama yang elemennya lebih besar daripada atau sama dengan kunci itu; *batas atas* (*upper bound*) adalah indeks pertama yang elemennya benar-benar lebih besar daripada kunci itu. Jika indeks semacam itu tidak ada, batasnya adalah n, posisi tepat sesudah ujung. Maka hitungannya cukup berupa selisih pada Persamaan (4.5).

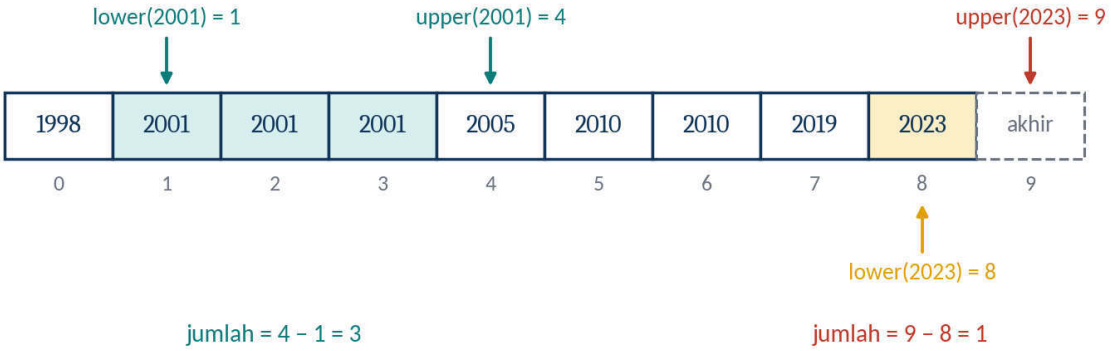
$$\text{count}(key) = \text{upper}(key) - \text{lower}(key) \tag{4.5}$$

Bagaimana menemukan batas bawah dalam $O(\log n)$ langkah? Dengan pencarian biner, tetapi pencarian biner untuk sebuah *batas*, bukan untuk sebuah elemen. Peliharalah rentang `[lo, hi]` yang pasti memuat jawabannya; pada awalnya rentang itu `[0, n]`, termasuk posisi lewat-ujung n, karena jawabannya bisa saja n. Lihatlah elemen tengah `a[mid]`. Jika ia lebih kecil daripada kunci, batasnya ada di sebelah kanannya, maka `lo = mid + 1`. Jika tidak, `a[mid]` sedikitnya sama dengan kunci, sehingga `mid` sendiri mungkin batasnya, dan kita mempertahankannya: `hi = mid`. Ketika `lo` dan `hi` bertemu, rentang itu memuat tepat satu calon, dan itulah jawabannya. Batas atas adalah pencarian yang sama dengan `<` diganti `<=`. **Pseudokode 4.6** memberikan keduanya.

Pseudokode 4.6 Pencarian biner untuk kedua batas sebuah blok kunci yang sama dalam larik terurut.

```
LOWER-BOUND(a, n, key)              // i pertama dengan a[i] >= key, atau n
1  lo = 0;  hi = n
2  while lo < hi
3      mid = lo + (hi - lo) / 2     // pembagian bilangan bulat
4      if a[mid] < key
5          lo = mid + 1
6      else hi = mid
7  return lo
UPPER-BOUND(a, n, key)              // i pertama dengan a[i] > key, atau n
8  the same, with line 4 reading: if a[mid] <= key
```

Gambar 4.7 menandai batas-batas untuk 2001 dan 2023 dan menelusuri `LOWER-BOUND(2001)` langkah demi langkah. Ikutilah tabelnya: rentang `[0, 9]` menyusut menjadi `[0, 4]`, `[0, 2]`, `[0, 1]`, dan akhirnya `[1, 1]`, setelah empat pemeriksaan, sesuai dengan Persamaan (4.2), yang mengizinkan paling banyak $\lfloor \log_2 9 \rfloor + 1 = 4$.



lowerBound(2001), langkah demi langkah

lo	hi	mid	a[mid]	keputusan
0	9	4	2005	hi = mid
0	4	2	2001	hi = mid
0	2	1	2001	hi = mid
0	1	0	1998	lo = mid+1
1	1			lo = hi: berhenti, jawaban 1

Gambar 4.7 Batas bawah dan batas atas 2001 dan 2023, dengan posisi lewat-ujung 9 digambar putus-putus, dan penelusuran LOWER-BOUND(2001) langkah demi langkah.

Mengapa `mid = lo + (hi - lo) / 2` dan bukan `(lo + hi) / 2`? Keduanya sama dalam matematika, tetapi dalam program `lo + hi` dapat melampaui `int` terbesar ketika larik berisi lebih dari sekitar satu miliar elemen, dan jumlah itu lalu *meluap* menjadi bilangan negatif. Kesalahan persis inilah yang bertahun-tahun bersemayam dalam kode pustaka yang dipakai luas sebelum disadari [9]; bentuk yang aman tidak pernah menghitung bilangan yang lebih besar daripada `hi`.

Biayanya. Setiap pencarian melakukan paling banyak $\lfloor \log_2 n \rfloor + 1$ pemeriksaan, sehingga satu penghitungan memerlukan $O(\log n)$ langkah, entah kunci itu muncul tiga kali atau tiga juta kali. Menghitung dengan berjalan keluar dari satu elemen yang ditemukan akan berbiaya $O(\log n + k)$ untuk k salinan, yaitu $O(n)$ bila sebagian besar larik berisi kunci yang sama.

4.8.4 Penyelesaian C++

Kode 4.21 adalah program lengkapnya.

Kode 4.21 `count_range.cpp`: batas bawah, batas atas, dan selisih keduanya.

```cpp
// count_range.cpp - Soal 4.6: berapa kali sebuah kunci muncul dalam larik
// terurut? Dua pencarian biner menemukan awal dan akhir bloknya.
#include <iostream>
using namespace std;

// Indeks i pertama dengan a[i] >= key (n bila tidak ada).
int lowerBound(const int a[], int n, int key) {
    int lo = 0, hi = n;                  // jawabannya ada di [lo, hi]
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] < key) {
            lo = mid + 1;                // a[mid] terlalu kecil: ke kanan mid
        } else {
            hi = mid;                    // a[mid] >= key: mid bisa jadi jawabannya
        }
    }
    return lo;
}

// Indeks i pertama dengan a[i] > key (n bila tidak ada).
int upperBound(const int a[], int n, int key) {
    int lo = 0, hi = n;
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] <= key) {
            lo = mid + 1;
        } else {
            hi = mid;
        }
    }
    return lo;
}

int main() {
    const int N = 9;
    int years[N] = {1998, 2001, 2001, 2001, 2005, 2010, 2010, 2019, 2023};
    int keys[] = {2001, 2010, 2002, 2023, 1990, 2030};
    for (int key : keys) {
        int first = lowerBound(years, N, key);
        int last = upperBound(years, N, key);
        cout << key << ": lower " << first << ", upper " << last
             << ", count " << last - first << '\n';
    }
    return 0;
}
```

```
2001: lower 1, upper 4, count 3
2010: lower 5, upper 7, count 2
2002: lower 4, upper 4, count 0
2023: lower 8, upper 9, count 1
1990: lower 0, upper 0, count 0
2030: lower 9, upper 9, count 0
```

Setiap baris cocok dengan hitungan tangan kita, dan baris untuk 2002, 1990, dan 2030 menunjukkan batas-batas blok kosong: sebelum elemen pertama (0), di antara dua elemen (4), dan lewat ujung (9). Syarat gelung `lo < hi` dan pembaruan `hi = mid` (bukan `mid - 1`) saling memerlukan: rentang itu selalu memuat sedikitnya satu calon, dan menyusut pada setiap langkah, karena `mid` selalu benar-benar lebih kecil daripada `hi`. Pustaka standar C++ menyediakan tepat kedua fungsi ini, `std::lower_bound` dan `std::upper_bound`, dengan makna yang sama, serta `std::equal_range`, yang mengembalikan keduanya sekaligus [5].

4.8.5 Penyelesaian Java dan Python

Kode 4.22 dan **Kode 4.23** memberikan rancangan yang sama dalam Java dan dalam Python.

Kode 4.22 `CountRange.java`: kedua pencarian batas dalam Java.

```java
// CountRange.java - Soal 4.6: berapa kali sebuah kunci muncul dalam larik
// terurut? Dua pencarian biner menemukan awal dan akhir bloknya.
public class CountRange {
    // Indeks i pertama dengan a[i] >= key (a.length bila tidak ada).
    static int lowerBound(int[] a, int key) {
        int lo = 0, hi = a.length;           // jawabannya ada di [lo, hi]
        while (lo < hi) {
            int mid = lo + (hi - lo) / 2;
            if (a[mid] < key) {
                lo = mid + 1;            // a[mid] terlalu kecil: ke kanan mid
            } else {
                hi = mid;                // a[mid] >= key: mid bisa jadi jawabannya
            }
        }
        return lo;
    }

    // Indeks i pertama dengan a[i] > key (a.length bila tidak ada).
    static int upperBound(int[] a, int key) {
        int lo = 0, hi = a.length;
        while (lo < hi) {
            int mid = lo + (hi - lo) / 2;
            if (a[mid] <= key) {
                lo = mid + 1;
            } else {
                hi = mid;
            }
        }
        return lo;
    }

    public static void main(String[] args) {
        int[] years = {1998, 2001, 2001, 2001, 2005, 2010, 2010, 2019, 2023};
        int[] keys = {2001, 2010, 2002, 2023, 1990, 2030};
        for (int key : keys) {
            int first = lowerBound(years, key);
            int last = upperBound(years, key);
            System.out.println(key + ": lower " + first + ", upper " + last
                    + ", count " + (last - first));
        }
    }
}
```

Kode 4.23 `count_range.py`: kedua pencarian batas dalam Python.

```python
# count_range.py - Soal 4.6: berapa kali sebuah kunci muncul dalam larik
# terurut? Dua pencarian biner menemukan awal dan akhir bloknya.


def lower_bound(a, key):
    """Indeks i pertama dengan a[i] >= key (len(a) bila tidak ada)."""
    lo, hi = 0, len(a)                  # jawabannya ada di [lo, hi]
    while lo < hi:
        mid = (lo + hi) // 2
        if a[mid] < key:
            lo = mid + 1            # a[mid] terlalu kecil: ke kanan mid
        else:
            hi = mid                # a[mid] >= key: mid bisa jadi jawabannya
    return lo


def upper_bound(a, key):
    """Indeks i pertama dengan a[i] > key (len(a) bila tidak ada)."""
    lo, hi = 0, len(a)
    while lo < hi:
        mid = (lo + hi) // 2
        if a[mid] <= key:
            lo = mid + 1
        else:
            hi = mid
    return lo


years = [1998, 2001, 2001, 2001, 2005, 2010, 2010, 2019, 2023]
for key in (2001, 2010, 2002, 2023, 1990, 2030):
    first = lower_bound(years, key)
    last = upper_bound(years, key)
    print(f"{key}: lower {first}, upper {last}, count {last - first}")
```

Keduanya mencetak keenam baris yang sama dengan program C++. Di Python bilangan bulat tidak pernah meluap, sehingga `(lo + hi) // 2` aman di sana; Java, seperti C++, memerlukan bentuk yang cermat. Modul standar Python `bisect` menyediakan kedua fungsi itu sebagai `bisect_left` dan `bisect_right` [7]. `Arrays.binarySearch` pada Java mengembalikan indeks *salah satu* elemen yang cocok, belum tentu yang pertama, sehingga menghitung dengannya memerlukan kerja tambahan [6].

4.8.6 Penyelesaian keliru yang menggoda

"Indeks harus terletak antara 0 dan $n - 1$", sehingga menggoda untuk memulai rentang pencarian sebagai `[0, n-1]`. **Kode 4.24** melakukannya pada kedua fungsi.

Kode 4.24 `WrongCountRange.java`: rentang pencarian dimulai sebagai $[0, n - 1]$, sehingga jawaban n tidak pernah dapat dikembalikan.

```java
// WrongCountRange.java - penyelesaian keliru yang menggoda untuk Soal 4.6:
// rentang awalnya [0, n-1], jadi "lewat ujung" tak pernah bisa dijawab.
public class WrongCountRange {
    static int lowerBound(int[] a, int key) {
        int lo = 0, hi = a.length - 1;          // seharusnya a.length
        while (lo < hi) {
            int mid = lo + (hi - lo) / 2;
            if (a[mid] < key) {
                lo = mid + 1;
            } else {
                hi = mid;
            }
        }
        return lo;
    }

    static int upperBound(int[] a, int key) {
        int lo = 0, hi = a.length - 1;          // seharusnya a.length
        while (lo < hi) {
            int mid = lo + (hi - lo) / 2;
            if (a[mid] <= key) {
                lo = mid + 1;
            } else {
                hi = mid;
            }
        }
        return lo;
    }

    public static void main(String[] args) {
        int[] years = {1998, 2001, 2001, 2001, 2005, 2010, 2010, 2019, 2023};
        int[] keys = {2001, 2010, 2002, 2023, 1990, 2030};
        for (int key : keys) {
            int first = lowerBound(years, key);
            int last = upperBound(years, key);
            System.out.println(key + ": lower " + first + ", upper " + last
                    + ", count " + (last - first));
        }
    }
}
```

```
2001: lower 1, upper 4, count 3
2010: lower 5, upper 7, count 2
2002: lower 4, upper 4, count 0
2023: lower 8, upper 8, count 0
1990: lower 0, upper 0, count 0
2030: lower 8, upper 8, count 0
```

Lima dari enam baris benar. Baris untuk 2023 menyatakan bahwa rak itu tidak memuat buku terbitan 2023, padahal buku terakhir tepat terbitan tahun itu. Batas atas 2023 seharusnya 9, lewat ujung, tetapi rentang yang dimulai dari `[0, 8]` tidak pernah dapat menghasilkan 9, sehingga ia mengembalikan 8, dan $8 - 8 = 0$. Untuk 2030 kekeliruan yang sama terjadi dua kali (kedua batas menjadi 8, bukan 9) dan saling meniadakan, itulah sebabnya baris itu tampak benar. Inilah kesalahan meleset-satu yang klasik pada pencarian batas, dan ia bersembunyi tepat di satu tempat, yaitu elemen terakhir; uji dengan elemen *terakhir* sebagai kunci menemukannya, dan itulah sebabnya contoh-contoh tepi pada langkah 2 memuatnya.

> **UJI DIRI.** Dengan hanya memakai `LOWER-BOUND`, berapa buku yang terbit pada tahun 2000 sampai 2010 inklusif? Telusurilah kedua panggilan yang Anda perlukan.
>
> *Jawaban.* Buku-buku dengan $2000 \le year \le 2010$ menempati indeks mulai dari `LOWER-BOUND(2000)` sampai, tetapi tidak termasuk, `LOWER-BOUND(2011)`. `LOWER-BOUND(2000)` = 1 (tahun pertama yang ≥ 2000 adalah 2001 di indeks 1) dan `LOWER-BOUND(2011)` = 7 (tahun

pertama yang ≥ 2011 adalah 2019 di indeks 7), sehingga hitungannya $7 - 1 = 6$: tiga buku dari 2001, satu dari 2005, dan dua dari 2010.

4.9 Soal 4.7: Pencarian Interpolasi pada Kode Buku

4.9.1 Soalnya

SOAL 4.7. Rak komputer Pustaka Ceria dikatalogkan dengan kode angka, terurut: 025 *The C++ Programming*, 034 *Mastering Delphi 6*, 041 *Professional C#*, 056 *Pure JavaScript v2*, 063 *Advanced JSP & Servlet*, 072 *Calculus Make it Easy*, 088 *Visual Basic 2005 Express*, 096 *Artificial Life: Volume 1*. Pakailah pencarian interpolasi untuk mencari kode 88, 60, 41, 25, 96, dan 100, cetaklah setiap pemeriksaan, dan bandingkan banyaknya pemeriksaan dengan pencarian biner.

4.9.2 Memahaminya dengan tangan

Dengan kata-kata sederhana: kita mencari dalam larik terurut berisi delapan bilangan, tetapi alih-alih selalu memeriksa bagian tengah, seperti pencarian biner, kita *memperkirakan* di mana kunci itu seharusnya berada berdasarkan nilainya. Beginilah orang mencari di kamus: tidak ada orang yang mencari kata "zebra" dengan membuka kamus di tengah.

Perkiraan itu mengandaikan bahwa kode-kode tersebar cukup merata antara yang terkecil dan yang terbesar. Di antara `low` dan `high`, kodenya berkisar dari $a[low]$ sampai $a[high]$; kunci itu terletak sejauh pecahan $\frac{key - a[low]}{a[high] - a[low]}$ di sepanjang rentang *nilai* itu, sehingga kita memeriksa pecahan yang sama di sepanjang rentang *indeks*. Hasilnya adalah posisi pemeriksaan pada Persamaan (4.6), dibulatkan ke bawah menjadi indeks bulat.

$$pos = low + \left\lfloor \frac{(key - a[low]) \cdot (high - low)}{a[high] - a[low]} \right\rfloor \tag{4.6}$$

Dengan tangan, untuk kunci 88 dengan `low = 0` dan `high = 7`: pecahannya $\frac{88-25}{96-25} = \frac{63}{71} \approx 0.887$, dan $0.887 \cdot 7 \approx 6.21$, yang dibulatkan ke bawah menjadi 6. Dan $a[6] = 88$: ketemu dengan satu pemeriksaan. Untuk kunci 60: $\frac{(60-25)\cdot 7}{71} = \frac{245}{71} \approx 3.45$, sehingga pemeriksaannya di indeks 3, tempat $a[3] = 56 < 60$; kunci itu, bila ada, terletak di sebelah kanan, maka `low = 4`. Sekarang $a[4] = 63$ sudah lebih besar daripada 60, sehingga kunci itu tidak mungkin terletak antara $a[4]$ dan $a[7]$: tidak ditemukan, setelah satu pemeriksaan. Untuk kunci 41: $\frac{16\cdot 7}{71} \approx 1.58$, pemeriksaan di indeks 1, $a[1] = 34 < 41$, maka `low = 2`; pemeriksaan berikutnya $2 + \frac{(41-41)\cdot 5}{96-41} = 2$, dan $a[2] = 41$: ketemu setelah dua pemeriksaan.

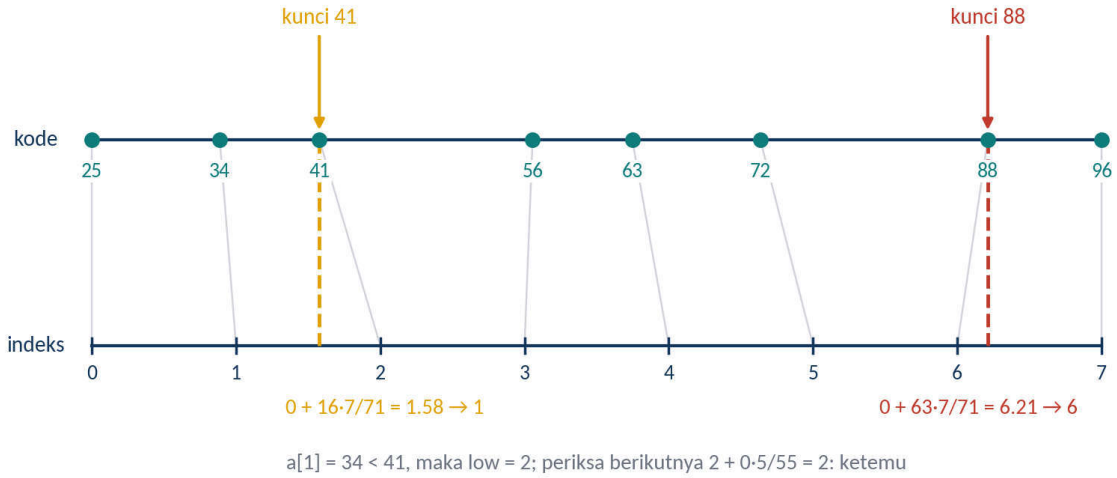
4.9.3 Merancangnya di atas kertas

Pseudokode 4.7 adalah pencarian interpolasi. Algoritma ini memiliki dua penjaga yang tidak diperlukan pencarian biner, dan keduanya mutlak perlu.

Pseudokode 4.7 Pencarian interpolasi; uji rentang pada baris 2 menjaga setiap pemeriksaan tetap di dalam larik, dan baris 3 mencegah pembagian dengan nol.

```
INTERPOLATION-SEARCH(a, n, key)
1  low = 0;  high = n - 1
2  while low <= high and a[low] <= key and key <= a[high]
3      if a[high] == a[low]
4          pos = low                        // hindari pembagian dengan nol
5      else pos = low + (key - a[low]) * (high - low) / (a[high] - a[low])
6      if a[pos] == key
7          return pos
8      elseif a[pos] < key
9          low = pos + 1
10     else high = pos - 1
11 return -1
```

Uji `a[low] <= key <= a[high]` pada baris 2 itulah yang menjaga pemeriksaan tetap di dalam rentang: ketika kunci terletak di antara kedua nilai ujung, pecahan pada Persamaan (4.6) bernilai antara 0 dan 1, sehingga `pos` terletak antara `low` dan `high`. Uji pada baris 3 menangani rentang yang semua nilainya sama, tempat penyebut Persamaan (4.6) akan bernilai nol. **Gambar 4.8** menggambarkan gagasannya: garis atas adalah sumbu *nilai*, garis bawah sumbu *indeks*, dan sebuah kunci diproyeksikan dari yang satu ke yang lain.



Gambar 4.8 Pencarian interpolasi memproyeksikan kunci dari sumbu nilai kode ke sumbu indeks; 88 mendarat di 6,21 dan ditemukan di indeks 6, sedangkan 41 mendarat di 1,58 dan memerlukan pemeriksaan kedua.

Biayanya. Ketika kunci-kunci tersebar merata, pencarian interpolasi rata-rata hanya memerlukan sekitar $\log_2 \log_2 n$ pemeriksaan [10], yaitu 5 untuk satu miliar butir, dibandingkan 30 untuk pencarian biner. Ketika tidak merata, ia bisa sangat lambat: pada kode 1, 2, 3, …, 99 dan 10^9, pencarian 99 hanya maju satu indeks per pemeriksaan, seluruhnya $O(n)$ pemeriksaan. Pencarian biner tidak pernah memerlukan lebih dari Persamaan (4.2), apa pun nilainya [2]. Itulah tawar-menawarnya: rata-rata yang lebih baik ditukar dengan kasus terburuk yang lebih buruk.

4.9.4 Penyelesaian C++

Kode 4.25 adalah program lengkapnya.

Kode 4.25 `interpolation_search.cpp`: pencarian interpolasi dengan jejak yang dicetak, berdampingan dengan pencarian biner yang menghitung pemeriksaannya.

```cpp
// interpolation_search.cpp - Soal 4.7: pencarian interpolasi pada
// kode buku Pustaka Ceria, mencetak setiap pemeriksaan dan membandingkannya
// dengan pencarian biner.
#include <iostream>
#include <string>
using namespace std;

const int N = 8;
int codes[N] = {25, 34, 41, 56, 63, 72, 88, 96};
string titles[N] = {"The C++ Programming", "Mastering Delphi 6",
                    "Professional C#", "Pure JavaScript v2",
                    "Advanced JSP & Servlet", "Calculus Make it Easy",
                    "Visual Basic 2005 Express", "Artificial Life: Volume 1"};

int interpolationSearch(int key, int& probes) {
    int low = 0, high = N - 1;
    probes = 0;
    // Kunci hanya mungkin ada selama codes[low] <= key <= codes[high].
    while (low <= high && key >= codes[low] && key <= codes[high]) {
        int pos = low;
        if (codes[high] != codes[low]) {    // hindari pembagian dengan nol
            pos = low + (key - codes[low]) * (high - low)
                      / (codes[high] - codes[low]);
        }
        probes++;
        cout << " probe " << pos << " (code " << codes[pos] << ")";
        if (codes[pos] == key) {
            return pos;
        } else if (codes[pos] < key) {
            low = pos + 1;
        } else {
            high = pos - 1;
        }
    }
    return -1;
}
```

Kode 4.25 (Lanjutan) *interpolation_search.cpp: pencarian interpolasi dengan jejak yang dicetak, berdampingan dengan pencarian biner yang menghitung pemeriksaannya.*

```
int binarySearchProbes(int key) {
    int low = 0, high = N - 1, probes = 0;
    while (low <= high) {
        int mid = (low + high) / 2;
        probes++;
        if (codes[mid] == key) {
            break;
        } else if (codes[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return probes;
}

int main() {
    int keys[] = {88, 60, 41, 25, 96, 100};
    for (int key : keys) {
        int probes;
        cout << "Search " << key << ":";
        int i = interpolationSearch(key, probes);
        if (i >= 0) {
            cout << " -> found at index " << i << ", \"" << titles[i] << "\"";
        } else {
            cout << " -> not found";
        }
        cout << "; probes: " << probes << " (binary search: "
             << binarySearchProbes(key) << ")\n";
    }
    return 0;
}
```

```
Search 88: probe 6 (code 88) -> found at index 6, "Visual Basic 2005 Express"; probes: 1
(binary search: 3)
Search 60: probe 3 (code 56) -> not found; probes: 1 (binary search: 3)
Search 41: probe 1 (code 34) probe 2 (code 41) -> found at index 2, "Professional C#";
probes: 2 (binary search: 3)
Search 25: probe 0 (code 25) -> found at index 0, "The C++ Programming"; probes: 1 (binary
search: 3)
Search 96: probe 7 (code 96) -> found at index 7, "Artificial Life: Volume 1"; probes: 1
(binary search: 4)
Search 100: -> not found; probes: 0 (binary search: 4)
```

Tiga baris pertama cocok dengan jejak tangan kita. Baris-baris lainnya menunjukkan kedua ujung rentang: 25 dan 96 langsung ditemukan, karena pecahan pada Persamaan (4.6) tepat bernilai 0 atau 1. Kunci 100 lebih besar daripada setiap kode, sehingga penjaga pada baris 2 menghentikan pencarian sebelum pemeriksaan pertama: nol pemeriksaan. Pada keenam kunci ini, pencarian interpolasi memerlukan 1 atau 2 pemeriksaan sementara pencarian biner memerlukan 3 atau 4, sesuai ramalan teori untuk kunci yang tersebar merata. Ungkapan `(key - codes[low]) * (high - low)` sengaja dihitung *sebelum* pembagian: membagi lebih dulu, `(key - codes[low]) / (codes[high] - codes[low])`, akan menjadi pembagian bilangan bulat antara bilangan yang lebih kecil dan yang lebih besar, dan hampir selalu menghasilkan 0.

4.9.5 Penyelesaian Java dan Python

Kode 4.26 dan **Kode 4.27** memberikan rancangan yang sama dalam Java dan dalam Python.

Kode 4.26 `InterpolationSearch.java`: pencarian yang sama dalam Java.

```java
// InterpolationSearch.java - Soal 4.7: pencarian interpolasi pada
// kode buku Pustaka Ceria, mencetak setiap pemeriksaan dan membandingkannya
// dengan pencarian biner.
public class InterpolationSearch {
    static int[] codes = {25, 34, 41, 56, 63, 72, 88, 96};
    static String[] titles = {"The C++ Programming", "Mastering Delphi 6",
        "Professional C#", "Pure JavaScript v2",
        "Advanced JSP & Servlet", "Calculus Make it Easy",
        "Visual Basic 2005 Express", "Artificial Life: Volume 1"};
    static int probes;

    static int interpolationSearch(int key) {
        int low = 0, high = codes.length - 1;
        probes = 0;
        // Kunci hanya mungkin ada selama codes[low] <= key <= codes[high].
        while (low <= high && key >= codes[low] && key <= codes[high]) {
            int pos = low;
            if (codes[high] != codes[low]) {     // hindari pembagian dengan nol
                pos = low + (key - codes[low]) * (high - low)
                        / (codes[high] - codes[low]);
            }
            probes++;
            System.out.print(" probe " + pos + " (code " + codes[pos] + ")");
            if (codes[pos] == key) {
                return pos;
            } else if (codes[pos] < key) {
                low = pos + 1;
            } else {
                high = pos - 1;
            }
        }
        return -1;
    }

    static int binarySearchProbes(int key) {
        int low = 0, high = codes.length - 1, count = 0;
        while (low <= high) {
            int mid = (low + high) / 2;
            count++;
            if (codes[mid] == key) {
                break;
            } else if (codes[mid] < key) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
```

Kode 4.26 (Lanjutan) *InterpolationSearch.java: pencarian yang sama dalam Java.*

```
        }
        return count;
    }

    public static void main(String[] args) {
        int[] keys = {88, 60, 41, 25, 96, 100};
        for (int key : keys) {
            System.out.print("Search " + key + ":");
            int i = interpolationSearch(key);
            if (i >= 0) {
                System.out.print(" -> found at index " + i
                        + ", \"" + titles[i] + "\"");
            } else {
                System.out.print(" -> not found");
            }
            System.out.println("; probes: " + probes + " (binary search: "
                    + binarySearchProbes(key) + ")");
        }
    }
}
```

Kode 4.27 `interpolation_search.py`: pencarian yang sama dalam Python, dengan jejak yang dikumpulkan dalam sebuah string.

```
# interpolation_search.py - Soal 4.7: pencarian interpolasi pada
# kode buku Pustaka Ceria, mencetak setiap pemeriksaan dan membandingkannya
# dengan pencarian biner.
CODES = [25, 34, 41, 56, 63, 72, 88, 96]
TITLES = ["The C++ Programming", "Mastering Delphi 6",
          "Professional C#", "Pure JavaScript v2",
          "Advanced JSP & Servlet", "Calculus Make it Easy",
          "Visual Basic 2005 Express", "Artificial Life: Volume 1"]


def interpolation_search(key):
    """Mengembalikan (indeks atau -1, banyak pemeriksaan, teks jejaknya)."""
    low, high = 0, len(CODES) - 1
    probes, trace = 0, ""
    # Kunci hanya mungkin ada selama CODES[low] <= key <= CODES[high].
    while low <= high and CODES[low] <= key <= CODES[high]:
        pos = low
        if CODES[high] != CODES[low]:   # hindari pembagian dengan nol
            pos = low + ((key - CODES[low]) * (high - low)
                         // (CODES[high] - CODES[low]))
        probes += 1
        trace += f" probe {pos} (code {CODES[pos]})"
        if CODES[pos] == key:
            return pos, probes, trace
        elif CODES[pos] < key:
            low = pos + 1
        else:
            high = pos - 1
    return -1, probes, trace


def binary_search_probes(key):
    low, high, probes = 0, len(CODES) - 1, 0
    while low <= high:
        mid = (low + high) // 2
        probes += 1
        if CODES[mid] == key:
            break
        elif CODES[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return probes
```

Kode 4.27 (Lanjutan) *interpolation_search.py: pencarian yang sama dalam Python, dengan jejak yang dikumpulkan dalam sebuah string.*

```
for key in (88, 60, 41, 25, 96, 100):
    i, probes, trace = interpolation_search(key)
    if i >= 0:
        result = f' -> found at index {i}, "{TITLES[i]}"'
    else:
        result = " -> not found"
    print(f"Search {key}:{trace}{result}; probes: {probes} "
          f"(binary search: {binary_search_probes(key)})")
```

Keduanya mencetak keenam baris yang sama dengan program C++. Versi Python mengembalikan tiga nilai sekaligus, yaitu indeks, banyaknya pemeriksaan, dan jejaknya, sebagai sebuah *tuple*, sedangkan versi C++ mengirim `probes` melalui referensi dan mencetak sambil berjalan. Python menulis penjaganya sebagai *perbandingan berantai* `CODES[low] <= key <= CODES[high]`, yang berarti persis sama dengan dua perbandingan yang digabung dengan `and`, dan memakai `//` untuk

pembagian bilangan bulat pada Persamaan (4.6). `/` pada Python akan menghasilkan bilangan titik mengambang, yang tidak dapat dipakai sebagai indeks.

4.9.6 Penyelesaian keliru yang menggoda

Penjaga pada baris 2 **Pseudokode 4.7** tampak seperti rincian kecil, sehingga menggoda untuk hanya menulis `while low <= high`, persis seperti pada pencarian biner. **Kode 4.28** melakukannya.

Kode 4.28 `wrong_interpolation.py`: pencarian interpolasi tanpa penjaga rentang.

```
# wrong_interpolation.py - penyelesaian keliru yang menggoda untuk Soal 4.7:
# gelung lupa memeriksa bahwa CODES[low] <= key <= CODES[high].
CODES = [25, 34, 41, 56, 63, 72, 88, 96]


def interpolation_search(key):
    low, high = 0, len(CODES) - 1
    while low <= high:                  # tanpa pemeriksaan rentang!
        pos = low + ((key - CODES[low]) * (high - low)
                     // (CODES[high] - CODES[low]))
        print(f"  key {key}: probe {pos}")
        if CODES[pos] == key:
            return pos
        elif CODES[pos] < key:
            low = pos + 1
        else:
            high = pos - 1
    return -1


print("88 ->", interpolation_search(88))
print("200 ->", interpolation_search(200))
```

```
  key 88: probe 6
88 -> 6
  key 200: probe 17
Traceback (most recent call last):
  File "/tmp/Chapter04/python/tempting/wrong_interpolation.py", line 22, in <module>
    print("200 ->", interpolation_search(200))
                    ^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/tmp/Chapter04/python/tempting/wrong_interpolation.py", line 12, in
interpolation_search
    if CODES[pos] == key:
       ~~~~~^^^^^
IndexError: list index out of range
```

Untuk kunci 88 tidak ada yang salah. Untuk 200, pecahan $\frac{200-25}{71}$ jauh lebih besar daripada 1, dan Persamaan (4.6) memberi $0 + \lfloor 175 \cdot 7/71 \rfloor = 17$, indeks yang jauh di luar larik berisi delapan elemen. Python berhenti dengan `IndexError`; Java akan berhenti dengan `ArrayIndexOutOfBoundsException`; C++, yang tidak memeriksa indeks, akan membaca apa pun yang terletak 17 `int` setelah awal larik dan melanjutkan dengan sampah, kecuali bila dijalankan di bawah *sanitizer*. Kunci yang *lebih kecil* daripada kode pertama malah lebih buruk di Python: ia menghasilkan indeks negatif, dan indeks negatif Python menghitung dari ujung list, sehingga `CODES[-2]` diam-diam membaca 88 alih-alih gagal. Penjaga itu bukan rincian kecil; dialah yang membuat rumus itu aman.

> **UJI DIRI.** Kode 063 diganti 090, sehingga kode terurutnya menjadi 25, 34, 41, 56, 72, 88, 90, 96. Telusurilah pencarian interpolasi untuk kunci 72.

Jawaban. Pemeriksaan 1: $0 + \lfloor 47 \cdot 7/71 \rfloor = \lfloor 4.63 \rfloor = 4$, dan $a[4] = 72$: ketemu dengan satu pemeriksaan. (Pencarian biner: `mid` = 3, $56 < 72$, `low` = 4; `mid` = 5, $88 > 72$, `high` = 4; `mid` = 4, ketemu: tiga pemeriksaan.)

4.10 Soal 4.8: Menelusuri Pengurutan Dasar

4.10.1 Soalnya

SOAL 4.8. Urutkan larik 5, 6, 8, 1, 3, 25, 10 secara naik dengan bubble sort, selection sort, dan insertion sort. Untuk setiap algoritma, tuliskan larik itu setelah setiap lintasan, dan hitunglah perbandingan antarelemen serta pertukarannya (atau, untuk insertion sort, pergeserannya).

4.10.2 Memahaminya dengan tangan

Dengan kata-kata sederhana: ini bukan soal dengan satu jawaban, melainkan sebuah *penelusuran*, latihan yang mengubah tiga algoritma dari Bab 3 dari sekadar resep menjadi sesuatu yang dapat Anda lihat. Larik itu memiliki $n = 7$ elemen, dan ketiga algoritma akan berakhir dengan 1, 3, 5, 6, 8, 10, 25. Yang berbeda adalah jalannya.

Sebelum menelusuri, ada baiknya menghitung satu hal: *inversi* larik itu, yaitu pasangan elemen yang berdiri dalam urutan yang salah satu terhadap yang lain. Telusurilah larik itu dan, untuk setiap elemen, hitunglah elemen-elemen lebih kecil di sebelah kanannya: 5 memiliki dua (1 dan 3), 6 memiliki dua (1 dan 3), 8 memiliki dua (1 dan 3), 1 tidak memiliki, 3 tidak memiliki, 25 memiliki satu (10), dan 10 tidak memiliki. Jumlahnya $2 + 2 + 2 + 1 = 7$ inversi. Ingatlah bilangan itu; ia akan muncul lagi dalam hitungan-hitungan nanti.

4.10.3 Merancangnya di atas kertas

Pseudokode 4.8 menghimpun ketiga algoritma di satu tempat, persis seperti yang didefinisikan Bab 3.

Pseudokode 4.8 Bubble sort, selection sort, dan insertion sort, sebagaimana ditelusuri pada Soal 4.8.

```
BUBBLE-SORT(a, n)
1  for i = 0 to n - 2
2      for j = 0 to n - 2 - i            // i tempat terakhir sudah tetap
3          if a[j] > a[j + 1]
4              swap a[j] and a[j + 1]
SELECTION-SORT(a, n)
5  for i = 0 to n - 2
6      m = index of the smallest of a[i..n-1]
7      if m != i
8          swap a[i] and a[m]
INSERTION-SORT(a, n)
9  for i = 1 to n - 1
10     key = a[i];  j = i - 1
11     while j >= 0 and a[j] > key
12         a[j + 1] = a[j];  j = j - 1    // geser
13     a[j + 1] = key
```

Gambar 4.9 menelusuri lintasan pertama bubble sort perbandingan demi perbandingan. Lintasan itu membandingkan tetangga pada $j = 0,1,\ldots,5$ dan menukar tiga kali (8 dengan 1, 8 dengan 3, 25 dengan 10). Nilai terbesar, 25, terbawa seperti gelembung ke ujung kanan, tempat ia menetap; itulah sebabnya gelung dalam pada lintasan berikutnya boleh berhenti satu tempat lebih awal.



Gambar 4.9 Enam perbandingan pada lintasan pertama bubble sort; pasangan kuning ditukar, pasangan hijau toska sudah berurutan.

Sekarang hitungannya, diturunkan dan bukan ditebak. *Bubble sort* membandingkan $6 + 5 + 4 + 3 + 2 + 1 = 21$ kali, yaitu Persamaan (4.1) untuk $n = 7$, apa pun datanya. Setiap pertukaran menukar dua tetangga yang tidak berurutan, sehingga menghapus tepat satu inversi tanpa menciptakan inversi baru; larik terurut ketika tidak ada lagi inversi yang tersisa; jadi bubble sort melakukan pertukaran tepat sebanyak inversinya: 7. *Selection sort* juga membandingkan 21 kali, karena mencari yang terkecil di antara $n - i$ elemen memerlukan $n - i - 1$ perbandingan, tetapi ia menukar paling banyak sekali per lintasan: paling banyak 6 pertukaran. *Insertion sort* menggeser setiap elemen ke kiri melewati tepat elemen-elemen yang lebih besar di depannya, sehingga ia juga melakukan tepat 7 pergeseran, satu per inversi; tetapi ia hanya membandingkan sampai kuncinya menemukan tempat, yang biasanya jauh kurang dari 21 kali.

Biayanya. Ketiganya $O(n^2)$ pada kasus terburuk, yaitu larik terbalik, yang memiliki inversi maksimum sebanyak $\frac{n(n-1)}{2}$. Pada larik yang sudah terurut, insertion sort hanya memerlukan $n - 1$ perbandingan, sedangkan bubble sort (dalam bentuk ini) dan selection sort tetap melakukan $\frac{n(n-1)}{2}$. Itulah sebabnya insertion sort yang dipakai dalam praktik untuk larik kecil atau hampir terurut, termasuk di dalam pengurutan pustaka C++, Java, dan Python [11].

4.10.4 Penyelesaian C++

Kode 4.29 adalah program lengkapnya.

Kode 4.29 `trace_sorts.cpp`: ketiga pengurutan dasar, masing-masing mencetak larik setelah setiap lintasan beserta hitungannya.

```cpp
// trace_sorts.cpp - Soal 4.8: bubble, selection, dan insertion sort ditelusuri
// lintasan demi lintasan pada tujuh bilangan yang sama, sambil menghitung.
#include <iostream>
using namespace std;

const int N = 7;

void show(const int a[]) {
    for (int i = 0; i < N; i++) {
        cout << ' ' << a[i];
    }
    cout << '\n';
}

void copyData(int a[]) {
    int data[N] = {5, 6, 8, 1, 3, 25, 10};
    for (int i = 0; i < N; i++) {
        a[i] = data[i];
    }
}

void bubbleSort(int a[]) {
    int comparisons = 0, swaps = 0;
    for (int i = 0; i < N - 1; i++) {
        for (int j = 0; j < N - 1 - i; j++) {   // i tempat akhir sudah tetap
            comparisons++;
            if (a[j] > a[j + 1]) {
                int t = a[j]; a[j] = a[j + 1]; a[j + 1] = t;
                swaps++;
            }
        }
        cout << "  pass " << i + 1 << ":";
        show(a);
    }
    cout << "  comparisons " << comparisons << ", swaps " << swaps << '\n';
}
```

Kode 4.29 (Lanjutan) *trace_sorts.cpp: ketiga pengurutan dasar, masing-masing mencetak larik setelah setiap lintasan beserta hitungannya.*

```
void selectionSort(int a[]) {
    int comparisons = 0, swaps = 0;
    for (int i = 0; i < N - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < N; j++) {       // terkecil di a[i..N-1]
            comparisons++;
            if (a[j] < a[minIndex]) {
                minIndex = j;
            }
        }
        if (minIndex != i) {
            int t = a[i]; a[i] = a[minIndex]; a[minIndex] = t;
            swaps++;
        }
        cout << "  pass " << i + 1 << ":";
        show(a);
    }
    cout << "  comparisons " << comparisons << ", swaps " << swaps << '\n';
}

void insertionSort(int a[]) {
    int comparisons = 0, shifts = 0;
    for (int i = 1; i < N; i++) {
        int key = a[i];
        int j = i - 1;
        while (j >= 0) {
            comparisons++;
            if (a[j] <= key) {
                break;                          // key sudah di tempatnya
            }
            a[j + 1] = a[j];                    // geser yang lebih besar ke kanan
            shifts++;
            j--;
        }
        a[j + 1] = key;
        cout << "  pass " << i << ":";
        show(a);
    }
    cout << "  comparisons " << comparisons << ", shifts " << shifts << '\n';
}
```

Kode 4.29 (Lanjutan) *trace_sorts.cpp: ketiga pengurutan dasar, masing-masing mencetak larik setelah setiap lintasan beserta hitungannya.*

```
int main() {
    int a[N];
    copyData(a);
    cout << "Start:";
    show(a);
    cout << "Bubble sort\n";
    bubbleSort(a);
    copyData(a);
    cout << "Selection sort\n";
    selectionSort(a);
    copyData(a);
    cout << "Insertion sort\n";
    insertionSort(a);
    return 0;
}
```

```
Start: 5 6 8 1 3 25 10
Bubble sort
  pass 1: 5 6 1 3 8 10 25
  pass 2: 5 1 3 6 8 10 25
  pass 3: 1 3 5 6 8 10 25
  pass 4: 1 3 5 6 8 10 25
  pass 5: 1 3 5 6 8 10 25
  pass 6: 1 3 5 6 8 10 25
  comparisons 21, swaps 7
Selection sort
  pass 1: 1 6 8 5 3 25 10
  pass 2: 1 3 8 5 6 25 10
  pass 3: 1 3 5 8 6 25 10
  pass 4: 1 3 5 6 8 25 10
  pass 5: 1 3 5 6 8 25 10
  pass 6: 1 3 5 6 8 10 25
  comparisons 21, swaps 5
Insertion sort
  pass 1: 5 6 8 1 3 25 10
  pass 2: 5 6 8 1 3 25 10
  pass 3: 1 5 6 8 3 25 10
  pass 4: 1 3 5 6 8 25 10
  pass 5: 1 3 5 6 8 25 10
  pass 6: 1 3 5 6 8 10 25
  comparisons 12, shifts 7
```

Hitungan yang tercetak menegaskan setiap ramalan Subbab 4.10.3: 21 perbandingan dan 7 pertukaran untuk bubble sort, 21 perbandingan dan 5 pertukaran untuk selection sort (pada lintasan 5 nilai terkecil yang tersisa sudah berada di tempatnya), 12 perbandingan dan 7 pergeseran untuk insertion sort. **Tabel 4.4** menghimpunnya.

Tabel 4.4 Hitungan Soal 4.8 untuk larik 5, 6, 8, 1, 3, 25, 10, yang memiliki 7 inversi.

Algoritma	Perbandingan	Pertukaran atau pergeseran	Lintasan yang mengubah larik
Bubble sort	21	7 pertukaran	3 dari 6
Selection sort	21	5 pertukaran	5 dari 6
Insertion sort	12	7 pergeseran	3 dari 6

Perhatikan dua hal dalam jejak itu. Bubble sort sebenarnya sudah menyelesaikan pekerjaannya setelah lintasan 3, tetapi masih melakukan tiga lintasan lagi yang tidak mengubah apa pun; penanda Boolean `swapped` yang menghentikan pengurutan setelah lintasan tanpa pertukaran akan berhenti setelah lintasan 4, lintasan pertama tanpa pertukaran, sehingga menghemat $2 + 1 = 3$ perbandingan dari lintasan 5 dan 6. Dan ketiga variabel pencacah (`comparisons`, `swaps`, `shifts`) adalah jenis instrumentasi yang mengubah perasaan samar ("insertion sort tampaknya lebih cepat") menjadi sebuah pengukuran.

4.10.5 Penyelesaian Java dan Python

Kode 4.30 dan **Kode 4.31** memberikan rancangan yang sama dalam Java dan dalam Python.

Kode 4.30 `TraceSorts.java`: ketiga pengurutan yang ditelusuri, dalam Java.

```java
// TraceSorts.java - Soal 4.8: bubble, selection, dan insertion sort ditelusuri
// lintasan demi lintasan pada tujuh bilangan yang sama, sambil menghitung.
public class TraceSorts {
    static final int[] DATA = {5, 6, 8, 1, 3, 25, 10};

    static void show(int[] a) {
        StringBuilder sb = new StringBuilder();
        for (int x : a) {
            sb.append(' ').append(x);
        }
        System.out.println(sb);
    }

    static void bubbleSort(int[] a) {
        int n = a.length, comparisons = 0, swaps = 0;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - 1 - i; j++) {    // i tempat akhir sudah tetap
                comparisons++;
                if (a[j] > a[j + 1]) {
                    int t = a[j]; a[j] = a[j + 1]; a[j + 1] = t;
                    swaps++;
                }
            }
            System.out.print("  pass " + (i + 1) + ":");
            show(a);
        }
        System.out.println("  comparisons " + comparisons + ", swaps " + swaps);
    }

    static void selectionSort(int[] a) {
        int n = a.length, comparisons = 0, swaps = 0;
        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {        // terkecil di a[i..n-1]
                comparisons++;
                if (a[j] < a[minIndex]) {
                    minIndex = j;
                }
            }
            if (minIndex != i) {
                int t = a[i]; a[i] = a[minIndex]; a[minIndex] = t;
                swaps++;
            }
            System.out.print("  pass " + (i + 1) + ":");
            show(a);
        }
```

Kode 4.30 (Lanjutan) *TraceSorts.java: ketiga pengurutan yang ditelusuri, dalam Java.*

```
        System.out.println("  comparisons " + comparisons + ", swaps " + swaps);
    }

    static void insertionSort(int[] a) {
        int n = a.length, comparisons = 0, shifts = 0;
        for (int i = 1; i < n; i++) {
            int key = a[i];
            int j = i - 1;
            while (j >= 0) {
                comparisons++;
                if (a[j] <= key) {
                    break;                          // key sudah di tempatnya
                }
                a[j + 1] = a[j];                    // geser yang lebih besar
                shifts++;
                j--;
            }
            a[j + 1] = key;
            System.out.print("  pass " + i + ":");
            show(a);
        }
        System.out.println("  comparisons " + comparisons
                + ", shifts " + shifts);
    }

    public static void main(String[] args) {
        System.out.print("Start:");
        show(DATA);
        System.out.println("Bubble sort");
        bubbleSort(DATA.clone());
        System.out.println("Selection sort");
        selectionSort(DATA.clone());
        System.out.println("Insertion sort");
        insertionSort(DATA.clone());
    }
}
```

Kode 4.31 `trace_sorts.py`: ketiga pengurutan yang ditelusuri, dalam Python.

```
# trace_sorts.py - Soal 4.8: bubble, selection, dan insertion sort ditelusuri
# lintasan demi lintasan pada tujuh bilangan yang sama, sambil menghitung.
DATA = [5, 6, 8, 1, 3, 25, 10]


def row(a):
    return "".join(f" {x}" for x in a)


def bubble_sort(a):
    n, comparisons, swaps = len(a), 0, 0
    for i in range(n - 1):
        for j in range(n - 1 - i):          # i tempat akhir sudah tetap
            comparisons += 1
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
                swaps += 1
        print(f"  pass {i + 1}:{row(a)}")
    print(f"  comparisons {comparisons}, swaps {swaps}")


def selection_sort(a):
    n, comparisons, swaps = len(a), 0, 0
    for i in range(n - 1):
        min_index = i
        for j in range(i + 1, n):           # cari yang terkecil di a[i..n-1]
            comparisons += 1
            if a[j] < a[min_index]:
                min_index = j
        if min_index != i:
            a[i], a[min_index] = a[min_index], a[i]
            swaps += 1
        print(f"  pass {i + 1}:{row(a)}")
    print(f"  comparisons {comparisons}, swaps {swaps}")
```

Kode 4.31 (Lanjutan) *trace_sorts.py: ketiga pengurutan yang ditelusuri, dalam Python.*

```
def insertion_sort(a):
    n, comparisons, shifts = len(a), 0, 0
    for i in range(1, n):
        key = a[i]
        j = i - 1
        while j >= 0:
            comparisons += 1
            if a[j] <= key:
                break                       # key sudah di tempatnya
            a[j + 1] = a[j]                 # geser yang lebih besar ke kanan
            shifts += 1
            j -= 1
        a[j + 1] = key
        print(f"  pass {i}:{row(a)}")
    print(f"  comparisons {comparisons}, shifts {shifts}")


print(f"Start:{row(DATA)}")
print("Bubble sort")
bubble_sort(DATA[:])
print("Selection sort")
selection_sort(DATA[:])
print("Insertion sort")
insertion_sort(DATA[:])
```

Keduanya mencetak persis keluaran program C++. Setiap pengurutan menerima *salinan* baru dari data, `DATA.clone()` di Java dan `DATA[:]` di Python; tanpa salinan, pengurutan kedua akan

menerima larik yang sudah diurutkan oleh yang pertama, dan hitungannya menjadi tidak bermakna. Python menukar dua elemen dalam satu pernyataan, `a[j], a[j + 1] = a[j + 1], a[j]`, yang mengevaluasi sisi kanan sepenuhnya sebelum memberi nilai; C++ dan Java memerlukan pertukaran tiga langkah melalui variabel sementara `t`.

4.10.6 Penyelesaian keliru yang menggoda

Kesalahan paling lazim dalam bubble sort buatan sendiri adalah batas gelung dalamnya. **Kode 4.32** membiarkan `j` berjalan sampai `n - i - 1`, bukan `n - i - 2`, satu langkah terlalu jauh.

Kode 4.32 `WrongBubble.java`: gelung dalam bubble sort berjalan satu langkah terlalu jauh.

```java
// WrongBubble.java - penyelesaian keliru yang menggoda untuk Soal 4.8:
// gelung dalam berjalan satu langkah terlalu jauh dan membaca lewat ujung larik.
public class WrongBubble {
    public static void main(String[] args) {
        int[] a = {5, 6, 8, 1, 3, 25, 10};
        int n = a.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i; j++) {        // seharusnya j < n - 1 - i
                if (a[j] > a[j + 1]) {
                    int t = a[j]; a[j] = a[j + 1]; a[j + 1] = t;
                }
            }
        }
        System.out.println(java.util.Arrays.toString(a));
    }
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 7 out of bounds
for length 7
    at WrongBubble.main(WrongBubble.java:9)
```

Pada lintasan pertama, $i = 0$, nilai terakhir `j` adalah 6, dan perbandingan `a[j] > a[j + 1]` membaca `a[7]`, yang tidak ada dalam larik sepanjang 7. Java memeriksa setiap indeks dan menghentikan program dengan sebuah *exception* yang menyebutkan indeks yang salah, panjang larik, dan barisnya. Itu sebuah hadiah. Kesalahan yang sama dalam C++ tidak menghasilkan pesan sama sekali: `a[7]` membaca `int` apa pun yang kebetulan terletak sesudah larik, dan bila nilai itu lebih kecil daripada elemen terakhir, pertukarannya *menulis* ke memori yang bukan milik program. Program C++ itu mungkin mencetak larik yang terurut, larik yang salah, atau berhenti mendadak nanti di tempat yang sama sekali lain. Itulah sebabnya setiap program C++ dalam bab ini juga dijalankan di bawah `-fsanitize=address`, yang mengubah pembacaan semacam itu menjadi laporan galat lengkap dengan barisnya.

> **UJI DIRI.** Tanpa menjalankan apa pun: berapa perbandingan dan pergeseran yang dilakukan insertion sort pada larik terbalik 25, 10, 8, 6, 5, 3, 1, dan berapa pada larik terurut 1, 3, 5, 6, 8, 10, 25?
>
> *Jawaban.* Larik terbalik memiliki inversi maksimum, $\frac{7 \cdot 6}{2} = 21$, sehingga 21 pergeseran; setiap kunci berjalan sampai ke depan, membandingkan dengan setiap elemen di depannya tanpa pernah bertemu yang lebih kecil, sehingga 21 perbandingan pula. Larik terurut tidak memiliki inversi, jadi 0 pergeseran, dan setiap dari 6 kunci cukup memerlukan satu perbandingan untuk mengetahui bahwa ia sudah di tempatnya: 6 perbandingan, yaitu $n - 1$.

4.11 Soal 4.9: Menggabungkan Dua Daftar Terurut

4.11.1 Soalnya

> **SOAL 4.9.** Dua cabang Pustaka Ceria menyimpan kode buku baru mereka dalam larik terurut: cabang A memiliki 3, 9, 12, 15, 31 dan cabang B memiliki 5, 9, 17, 23, 35, 40, 41. Hasilkan katalog gabungan sebagai satu larik terurut, dalam satu lintasan, tanpa mengurutkan dari awal, dan hitunglah perbandingannya.

4.11.2 Memahaminya dengan tangan

Dengan kata-kata sederhana: kita memiliki dua tumpukan terurut dan menginginkan satu tumpukan terurut. Cara tercepat dengan tangan adalah cara Anda menggabungkan dua tumpuk kartu terurut yang terbuka menghadap ke atas: lihat kartu teratas setiap tumpukan, ambil yang lebih kecil, dan ulangi. Kartu teratasnya 3 (A) dan 5 (B): ambil 3. Sekarang 9 (A) dan 5 (B): ambil 5. Sekarang 9 dan 9: seri; kita mengambil yang dari A lebih dulu, agar kode yang sama mempertahankan urutan "A sebelum B" (ini menjaga penggabungan tetap stabil). Sekarang 12 dan 9: ambil 9 dari B. Lalu 12 (A), 15 (A), 17 (B), 23 (B), 31 (A). Cabang A kini kosong, dan kita telah melakukan sembilan perbandingan, satu untuk setiap kartu yang diambil selama kedua tumpukan masih berisi. Sisa B, yaitu 35, 40, 41, dapat diambil tanpa perbandingan, karena tidak ada lagi yang dapat dibandingkan dengannya. Hasilnya 3, 5, 9, 9, 12, 15, 17, 23, 31, 35, 40, 41: dua belas kode.

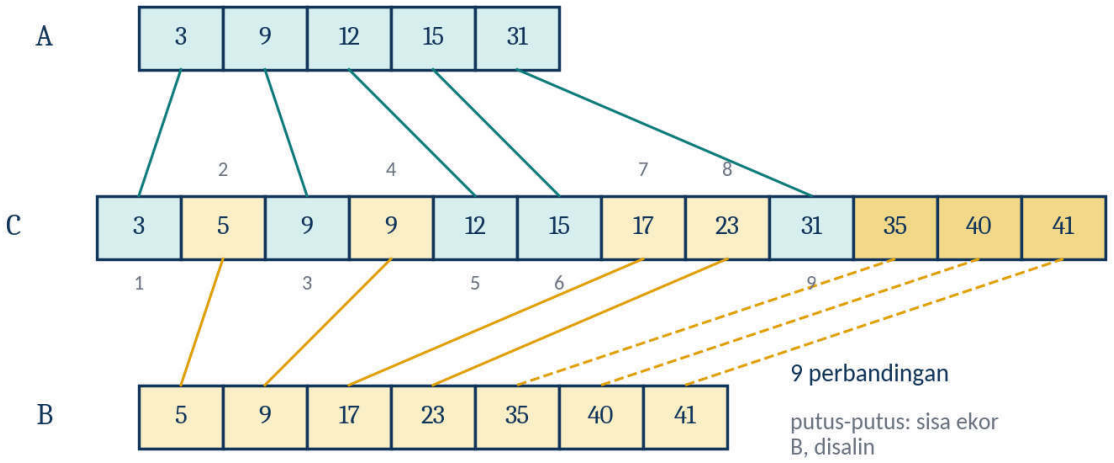
4.11.3 Merancangnya di atas kertas

Kita memelihara tiga indeks: `i` ke dalam A, `j` ke dalam B, dan `k` ke dalam hasil C. **Pseudokode 4.9** adalah penggabungan (*merge*), langkah yang memberi nama kepada merge sort (Bab 3).

Pseudokode 4.9 Menggabungkan dua larik terurut ke larik ketiga; paling banyak satu dari kedua gelung penyalin yang bekerja.

```
MERGE(a, n, b, m, c)
1  i = 0;  j = 0;  k = 0
2  while i < n and j < m            // kedua larik masih berisi
3      if a[i] <= b[j]              // <= mengambil yang seri dari a: stabil
4          c[k] = a[i];  i = i + 1
5      else c[k] = b[j];  j = j + 1
6      k = k + 1
7  while i < n                      // salin sisa a, bila ada
8      c[k] = a[i];  i = i + 1;  k = k + 1
9  while j < m                      // salin sisa b, bila ada
10     c[k] = b[j];  j = j + 1;  k = k + 1
```

Gambar 4.10 menggambar asal setiap elemen C: garis hijau toska dari A, garis kuning dari B, dan garis putus-putus untuk ekor B yang disalin tanpa perbandingan. Bilangan kecil di dekat setiap elemen C adalah nomor perbandingan yang menempatkannya.



Gambar 4.10 Menggabungkan A dan B ke dalam C: setiap elemen C dihubungkan ke asalnya, bilangan kecil menghitung perbandingan, dan ekor B yang putus-putus disalin tanpa perbandingan.

Mengapa hasilnya terurut? Pada setiap langkah, elemen yang diambil adalah yang terkecil di antara semua elemen yang belum diambil: A dan B terurut, sehingga elemen tersisa terkecil di A adalah `a[i]` dan elemen tersisa terkecil di B adalah `b[j]`, dan kita mengambil yang lebih kecil dari keduanya. Karena setiap elemen yang diambil sedikitnya sama besar dengan semua yang diambil sebelumnya, C tumbuh dalam urutan naik.

Biayanya. Setiap perbandingan menempatkan satu elemen ke dalam C, dan elemen terakhir tidak memerlukan perbandingan, sehingga penggabungan n dan m elemen melakukan paling banyak $n + m - 1$ perbandingan, dan $n + m$ penyalinan: $O(n + m)$ langkah. Di sini $n + m - 1 = 11$, tetapi hanya 9 yang diperlukan, karena A habis ketika tiga elemen B masih tersisa. Mengurutkan kedua belas kode dari awal dengan insertion sort dapat berbiaya sampai $\frac{12 \cdot 11}{2} = 66$ perbandingan; keterurutan kedua masukan memang berharga.

4.11.4 Penyelesaian C++

Kode 4.33 adalah program lengkapnya.

Kode 4.33 `merge_sorted.cpp`: penggabungan dua larik terurut, sambil menghitung perbandingan.

```cpp
// merge_sorted.cpp - Soal 4.9: gabungkan dua larik terurut menjadi satu
// larik terurut dalam satu jalan kiri-ke-kanan (inti merge sort, Bab 3).
#include <iostream>
using namespace std;

void show(const char* label, const int a[], int n) {
    cout << label;
    for (int i = 0; i < n; i++) {
        cout << ' ' << a[i];
    }
    cout << '\n';
}

// Menggabung a[0..n-1] dan b[0..m-1] ke c; mengembalikan banyak perbandingan.
int mergeArrays(const int a[], int n, const int b[], int m, int c[]) {
    int i = 0, j = 0, k = 0, comparisons = 0;
    while (i < n && j < m) {                // kedua larik masih berisi
        comparisons++;
        if (a[i] <= b[j]) {                 // <= yang seri diambil dari a dulu
            c[k++] = a[i++];
        } else {
            c[k++] = b[j++];
        }
    }
    while (i < n) {                         // salin sisa a ...
        c[k++] = a[i++];
    }
    while (j < m) {                         // ... atau sisa b
        c[k++] = b[j++];
    }
    return comparisons;
}

int main() {
    int a[] = {3, 9, 12, 15, 31};
    int b[] = {5, 9, 17, 23, 35, 40, 41};
    int n = 5, m = 7;
    int c[12];
    int comparisons = mergeArrays(a, n, b, m, c);
    show("Branch A:", a, n);
    show("Branch B:", b, m);
    show("Merged:  ", c, n + m);
    cout << "Comparisons: " << comparisons << '\n';
    return 0;
}
```

```
Branch A: 3 9 12 15 31
Branch B: 5 9 17 23 35 40 41
Merged:   3 5 9 9 12 15 17 23 31 35 40 41
Comparisons: 9
```

Keluarannya adalah daftar yang kita susun dengan tangan, dengan sembilan perbandingan yang kita hitung. `c[k++] = a[i++];` adalah idiom C++ yang ringkas: ia menyalin `a[i]` ke `c[k]` lalu *setelah itu* menaikkan `i` dan `k` masing-masing satu, karena `++` yang ditulis sesudah variabel (bentuk *postfix*) memakai nilai lamanya lebih dulu. Larik hasil `c` memiliki tepat $5 + 7 = 12$ kotak; penggabungan tidak pernah menulis lebih dari $n + m$ elemen, sehingga tidak mungkin meluap. Versi pustakanya adalah `std::merge(a, a + n, b, b + m, c)` [5].

4.11.5 Penyelesaian Java dan Python

Kode 4.34 dan **Kode 4.35** memberikan rancangan yang sama dalam Java dan dalam Python.

Kode 4.34 `MergeSorted.java`: penggabungan yang sama dalam Java, mengembalikan larik baru.

```java
// MergeSorted.java - Soal 4.9: gabungkan dua larik terurut menjadi satu
// larik terurut dalam satu jalan kiri-ke-kanan (inti merge sort, Bab 3).
public class MergeSorted {
    static int comparisons;

    static void show(String label, int[] a) {
        StringBuilder sb = new StringBuilder(label);
        for (int x : a) {
            sb.append(' ').append(x);
        }
        System.out.println(sb);
    }

    static int[] mergeArrays(int[] a, int[] b) {
        int[] c = new int[a.length + b.length];
        int i = 0, j = 0, k = 0;
        comparisons = 0;
        while (i < a.length && j < b.length) {   // kedua larik masih berisi
            comparisons++;
            if (a[i] <= b[j]) {                  // <= yang seri diambil dari a dulu
                c[k++] = a[i++];
            } else {
                c[k++] = b[j++];
            }
        }
        while (i < a.length) {                   // salin sisa a ...
            c[k++] = a[i++];
        }
        while (j < b.length) {                   // ... atau sisa b
            c[k++] = b[j++];
        }
        return c;
    }

    public static void main(String[] args) {
        int[] a = {3, 9, 12, 15, 31};
        int[] b = {5, 9, 17, 23, 35, 40, 41};
        int[] c = mergeArrays(a, b);
        show("Branch A:", a);
        show("Branch B:", b);
        show("Merged:  ", c);
        System.out.println("Comparisons: " + comparisons);
    }
}
```

Kode 4.35 `merge_sorted.py`: penggabungan yang sama dalam Python, dengan ekornya ditambahkan melalui `extend`.

```
# merge_sorted.py - Soal 4.9: gabungkan dua larik terurut menjadi satu
# larik terurut dalam satu jalan kiri-ke-kanan (inti merge sort, Bab 3).


def merge_arrays(a, b):
    """Mengembalikan (list hasil gabungan, banyak perbandingan)."""
    c = []
    i = j = comparisons = 0
    while i < len(a) and j < len(b):    # kedua list masih berisi
        comparisons += 1
        if a[i] <= b[j]:                # <= yang seri diambil dari a dulu
            c.append(a[i])
            i += 1
        else:
            c.append(b[j])
            j += 1
    c.extend(a[i:])                     # salin sisa a ...
    c.extend(b[j:])                     # ... atau sisa b
    return c, comparisons


a = [3, 9, 12, 15, 31]
b = [5, 9, 17, 23, 35, 40, 41]
c, comparisons = merge_arrays(a, b)
print("Branch A:", *a)
print("Branch B:", *b)
print("Merged:  ", *c)
print("Comparisons:", comparisons)
```

Keduanya mencetak keempat baris yang persis sama. Metode Java membuat sendiri larik hasilnya, dengan ukuran yang tepat, lalu mengembalikannya, dan cara ini lebih aman daripada meminta pemanggil menyediakan larik berukuran tepat. Versi Python membangun list dengan `append` dan menambahkan kedua ekor sisa dengan `extend`; salah satu dari kedua irisan `a[i:]` dan `b[j:]` selalu kosong, sehingga menambahkan keduanya benar dan ringkas. Pustaka standar Python menyediakan `heapq.merge(a, b)`, yang menggabungkan urutan terurut sebanyak apa pun [7].

4.11.6 Penyelesaian keliru yang menggoda

Gelung utama terasa seperti seluruh algoritmanya, dan dua gelung penyalin kecil sesudahnya terasa seperti tambahan belaka. **Kode 4.36** menghilangkan keduanya.

Kode 4.36 `wrong_merge.cpp`: penggabungan yang melupakan elemen-elemen sisa.

```cpp
// wrong_merge.cpp - penyelesaian keliru yang menggoda untuk Soal 4.9:
// gelung berhenti saat SATU larik habis, dan sisanya terlupakan.
#include <iostream>
using namespace std;

int main() {
    int a[] = {3, 9, 12, 15, 31};
    int b[] = {5, 9, 17, 23, 35, 40, 41};
    int n = 5, m = 7;
    int c[12];
    int i = 0, j = 0, k = 0;
    while (i < n && j < m) {
        if (a[i] <= b[j]) {
            c[k++] = a[i++];
        } else {
            c[k++] = b[j++];
        }
    }
    cout << "Merged:";
    for (int t = 0; t < k; t++) {
        cout << ' ' << c[t];
    }
    cout << "\n(" << k << " of " << n + m << " items)\n";
    return 0;
}
```

```
Merged: 3 5 9 9 12 15 17 23 31
(9 of 12 items)
```

Sembilan dari dua belas kode selamat; 35, 40, dan 41 hilang. Gelung utama berhenti segera setelah *salah satu* larik habis, dan apa pun yang tersisa di larik lainnya tidak pernah disalin. Program ini tampak benar pada masukan yang elemen-elemen terbesarnya kebetulan bergantian di antara kedua larik sampai paling akhir, itulah sebabnya ia lolos dari pengujian sambil lalu. Hitungan yang tercetak, `(9 of 12 items)`, adalah pemeriksaan murah yang sebaiknya dilakukan setiap penggabungan: keluarannya harus memuat tepat $n + m$ elemen.

> **UJI DIRI.** Katalog gabungan harus mencantumkan setiap kode sekali saja, sehingga 9 yang ganda harus hilang. Baris mana dari **Pseudokode 4.9** yang berubah, dan berapa hasil serta banyaknya perbandingan?
>
> *Jawaban.* Tambahkan kasus ketiga pada gelung utama: bila `a[i] == b[j]`, salin elemen itu sekali dan majukan `i` *dan* `j`. Hasilnya 3, 5, 9, 12, 15, 17, 23, 31, 35, 40, 41 (sebelas kode). Perbandingannya: 3 terhadap 5, 9 terhadap 5, 9 terhadap 9 (sama, keduanya maju), 12 terhadap 17, 15 terhadap 17, 31 terhadap 17, 31 terhadap 23, 31 terhadap 35: delapan kali lewat gelung utama. (Bila setiap kali lewat menguji `<`, `==`, dan `>` secara terpisah, hitunglah *kali lewat*-nya, bukan uji satu per satu.)

4.12 Soal 4.10: Menghapus Nilai di Tempat

4.12.1 Soalnya

> **SOAL 4.10.** Sebuah larik berisi bilangan 7, 3, 3, 5, 3, 9. Hapuslah setiap kemunculan nilai x tertentu dari larik itu *di tempat* (tanpa larik kedua), dengan mempertahankan urutan bilangan-bilangan lainnya, lalu laporkan ukuran barunya. Ujilah dengan x = 3, 7, 9, dan 4, serta pada larik 3, 3, 3 dengan $x = 3$.

4.12.2 Memahaminya dengan tangan

Dengan kata-kata sederhana: larik dalam C++ memiliki banyak kotak yang tetap, sehingga "menghapus" tidak dapat membuat larik itu secara fisik lebih pendek. Artinya adalah: pindahkan bilangan-bilangan yang kita simpan ke depan, dengan urutannya, dan ingatlah berapa kotak terdepan yang kini dihitung. Bilangan itulah *ukuran* (*size*) yang baru; apa pun yang terletak di kotak-kotak sesudahnya tidak lagi menjadi bagian larik, meskipun masih ada di memori. Pemisahan antara *kapasitas* (kotak yang ada) dan *ukuran* (kotak yang terpakai) ini mengalir di sepanjang buku ini: *stack* dan *queue* di Bab 5 dibangun di atasnya.

Dengan tangan, menghapus 3 dari 7, 3, 3, 5, 3, 9 menyisakan 7, 5, 9, ukuran baru 3, dan 3 elemen terhapus. Menghapus 7 (elemen pertama) menyisakan 3, 3, 5, 3, 9; menghapus 9 (yang terakhir) menyisakan 7, 3, 3, 5, 3; menghapus 4 (tidak ada) tidak mengubah apa pun; dan menghapus 3 dari 3, 3, 3 menyisakan larik kosong berukuran 0. Semua tepinya ada di sini: pertama, terakhir, tidak ada, semuanya, dan dua tetangga yang sama berturut-turut.

4.12.3 Merancangnya di atas kertas

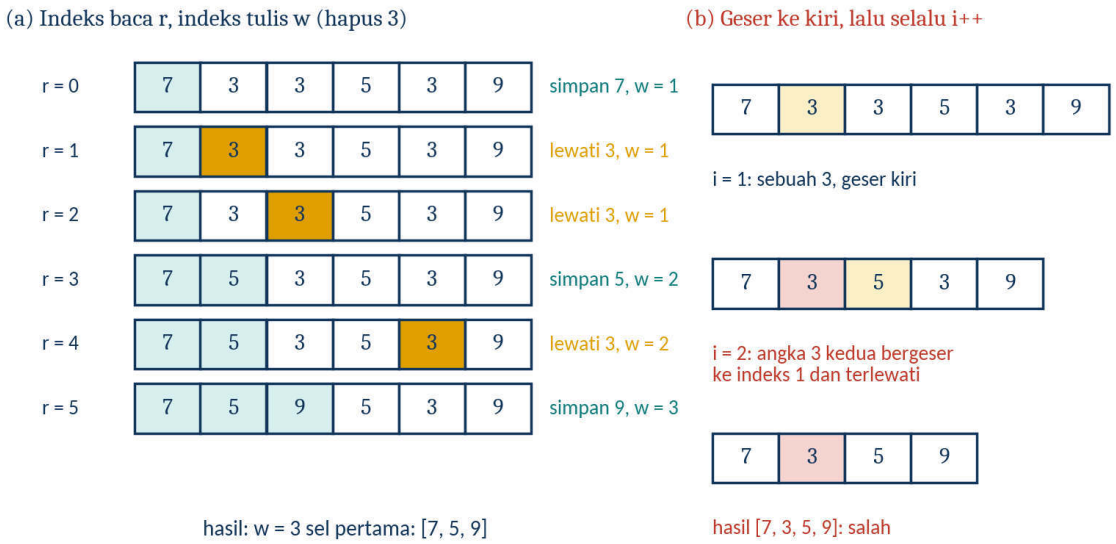
Rancangan yang paling jelas mengikuti resep "cari, dan bila ketemu, hapus" dari Bab 2: temukan sebuah 3, geser semua yang sesudahnya satu tempat ke kiri, kurangi ukurannya, lalu lanjutkan. Cara itu berhasil bila Anda cermat (lihat Subbab 4.12.6), tetapi setiap penghapusan menggeser sampai n elemen, sehingga menghapus banyak nilai berbiaya $O(n^2)$.

Rancangan satu lintasan memakai dua indeks yang bergerak dalam larik yang sama: indeks *baca* `r` yang mengunjungi setiap elemen sekali, dan indeks *tulis* `w` yang menandai kotak kosong berikutnya dari bagian yang disimpan. Setiap kali `a[r]` adalah bilangan yang kita simpan, kita menyalinnya ke `a[w]` dan memajukan `w`; bila ia x, kita hanya memajukan `r`. **Pseudokode 4.10** adalah seluruh algoritmanya.

Pseudokode 4.10 Menghapus setiap kemunculan x dalam satu lintasan dengan indeks baca r dan indeks tulis w.

```
DELETE-ALL(a, n, x)
1  w = 0                                  // tempat kosong berikutnya
2  for r = 0 to n - 1                     // baca setiap butir tepat sekali
3      if a[r] != x
4          a[w] = a[r]
5          w = w + 1
6  return w                                // ukuran baru
```

Gambar 4.11(a) menelusurinya untuk $x = 3$. Invariannya, yaitu sifat yang berlaku sebelum setiap putaran gelung, adalah: *`a[0..w-1]` memuat, dengan urutan aslinya, tepat elemen-elemen yang disimpan di antara `a[0..r-1]`.* Sifat itu benar di awal, ketika kedua rentang kosong. Setiap putaran mempertahankannya: elemen yang disimpan ditambahkan di `w`, elemen yang dihapus dilewati. Ketika gelung berakhir, `r = n`, sehingga `a[0..w-1]` memuat tepat elemen-elemen yang disimpan dari seluruh larik. Mengapa menulis ke `a[w]` tidak pernah merusak elemen yang masih kita perlukan? Karena `w <= r` selalu berlaku (`w` tumbuh paling cepat secepat `r`), sehingga kita hanya pernah menimpa kotak yang sudah dilewati indeks baca.



Gambar 4.11 (a) Indeks baca r mengunjungi setiap kotak sekali dan indeks tulis w merapatkan bilangan yang disimpan ke depan; (b) menggeser ke kiri lalu selalu memajukan i melewatkan 3 kedua dari dua 3 yang bersebelahan.

Biayanya. Satu lintasan, satu perbandingan dan paling banyak satu penyalinan per elemen: waktu $O(n)$ dan ruang tambahan $O(1)$, dibandingkan $O(n^2)$ untuk penggeseran berulang.

4.12.4 Penyelesaian C++

Kode 4.37 adalah program lengkapnya.

Kode 4.37 `delete_value.cpp`: menghapus setiap kemunculan sebuah nilai dengan indeks baca dan indeks tulis.

```cpp
// delete_value.cpp - Soal 4.10: hapus setiap kemunculan sebuah nilai dari
// larik di tempat, dalam satu lintasan, urutan butir lain tetap.
#include <iostream>
using namespace std;

void show(const int a[], int n) {
    cout << '[';
    for (int i = 0; i < n; i++) {
        cout << (i > 0 ? ", " : "") << a[i];
    }
    cout << ']';
}

// Simpan setiap butir yang bukan x, dirapatkan ke depan; kembalikan ukuran baru.
int deleteAll(int a[], int n, int x) {
    int w = 0;                          // tempat kosong berikutnya
    for (int r = 0; r < n; r++) {       // r membaca setiap butir tepat sekali
        if (a[r] != x) {
            a[w] = a[r];
            w++;
        }
    }
    return w;
}

void test(const int start[], int n, int x) {
    int a[10] = {0};
    for (int i = 0; i < n; i++) {
        a[i] = start[i];
    }
    cout << "delete " << x << " from ";
    show(a, n);
    int newSize = deleteAll(a, n, x);
    cout << " -> ";
    show(a, newSize);
    cout << " (" << n - newSize << " removed)\n";
}
```

Kode 4.37 (Lanjutan) *delete_value.cpp: menghapus setiap kemunculan sebuah nilai dengan indeks baca dan indeks tulis.*

```cpp
int main() {
    int data[] = {7, 3, 3, 5, 3, 9};
    int threes[] = {3, 3, 3};
    test(data, 6, 3);
    test(data, 6, 7);
    test(data, 6, 9);
    test(data, 6, 4);
    test(threes, 3, 3);
    return 0;
}
```

```
delete 3 from [7, 3, 3, 5, 3, 9] -> [7, 5, 9] (3 removed)
delete 7 from [7, 3, 3, 5, 3, 9] -> [3, 3, 5, 3, 9] (1 removed)
delete 9 from [7, 3, 3, 5, 3, 9] -> [7, 3, 3, 5, 3] (1 removed)
delete 4 from [7, 3, 3, 5, 3, 9] -> [7, 3, 3, 5, 3, 9] (0 removed)
delete 3 from [3, 3, 3] -> [] (3 removed)
```

Kelima baris cocok dengan hasil tangan. Fungsi `test` lebih dulu menyalin data asli ke larik lokal, agar setiap uji berangkat dari bilangan yang sama; `int a[10] = {0};` memberi larik lokal itu isi yang terdefinisi di setiap kotaknya, termasuk kotak yang tidak dipakai uji. Ungkapan `(i > 0 ? ", " : "")` pada `show` adalah *operator kondisional*: ia mencetak pemisah sebelum setiap elemen kecuali yang pertama, trik baku untuk mencetak daftar tanpa koma di ujung. Pustaka C++

mengerjakan algoritma persis ini dalam `std::remove(a, a + n, x)`, yang mengembalikan pointer ke ujung yang baru; namanya menyesatkan banyak pemrogram, karena fungsi itu tidak mengubah ukuran apa pun, ia hanya merapatkan elemen yang disimpan ke depan [5].

4.12.5 Penyelesaian Java dan Python

Kode 4.38 dan **Kode 4.39** memberikan rancangan yang sama dalam Java dan dalam Python.

Kode 4.38 `DeleteValue.java`: penghapusan dua indeks yang sama dalam Java.

```java
// DeleteValue.java - Soal 4.10: hapus setiap kemunculan sebuah nilai dari
// larik di tempat, dalam satu lintasan, urutan butir lain tetap.
public class DeleteValue {
    static String show(int[] a, int n) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < n; i++) {
            sb.append(i > 0 ? ", " : "").append(a[i]);
        }
        return sb.append(']').toString();
    }

    // Simpan setiap butir yang bukan x, rapat ke depan; kembalikan ukuran baru.
    static int deleteAll(int[] a, int n, int x) {
        int w = 0;                      // tempat kosong berikutnya
        for (int r = 0; r < n; r++) {      // r membaca setiap butir tepat sekali
            if (a[r] != x) {
                a[w] = a[r];
                w++;
            }
        }
        return w;
    }

    static void test(int[] start, int x) {
        int[] a = start.clone();
        int n = a.length;
        String before = show(a, n);
        int newSize = deleteAll(a, n, x);
        System.out.println("delete " + x + " from " + before + " -> "
                + show(a, newSize) + " (" + (n - newSize) + " removed)");
    }

    public static void main(String[] args) {
        int[] data = {7, 3, 3, 5, 3, 9};
        test(data, 3);
        test(data, 7);
        test(data, 9);
        test(data, 4);
        test(new int[] {3, 3, 3}, 3);
    }
}
```

Kode 4.39 `delete_value.py`: penghapusan dua indeks yang sama dalam Python.

```
# delete_value.py - Soal 4.10: hapus setiap kemunculan sebuah nilai dari
# larik di tempat, dalam satu lintasan, urutan butir lain tetap.


def show(a, n):
    return "[" + ", ".join(str(a[i]) for i in range(n)) + "]"


def delete_all(a, n, x):
    """Simpan setiap butir yang bukan x, rapat ke depan; kembalikan ukuran baru."""
    w = 0                                  # tempat kosong berikutnya
    for r in range(n):                     # r membaca setiap butir tepat sekali
        if a[r] != x:
            a[w] = a[r]
            w += 1
    return w


def test(start, x):
    a = start[:]
    n = len(a)
    before = show(a, n)
    new_size = delete_all(a, n, x)
    print(f"delete {x} from {before} -> {show(a, new_size)} "
          f"({n - new_size} removed)")


data = [7, 3, 3, 5, 3, 9]
test(data, 3)
test(data, 7)
test(data, 9)
test(data, 4)
test([3, 3, 3], 3)
```

Keduanya mencetak kelima baris yang sama. Keduanya juga menyimpan ukuran dalam variabel tersendiri, untuk menunjukkan algoritma persis seperti dalam C++. Dalam Python sehari-hari, idiomnya adalah *list comprehension*, `a[:] = [v for v in a if v != x]`, yang membangun list berisi butir yang disimpan lalu mengganti isi `a` dengannya; pemberian nilai kepada irisan `a[:]` mempertahankan objek list yang sama, sehingga variabel lain yang merujuk kepadanya ikut melihat perubahan itu. Di Java orang akan memakai `ArrayList<Integer>` dan `list.removeIf(v -> v == x)`.

4.12.6 Penyelesaian keliru yang menggoda

Rancangan yang paling jelas dari Subbab 4.12.3, bila ditulis dengan cara yang paling jelas, adalah **Kode 4.40**.

Kode 4.40 `wrong_delete_value.py`: penghapusan dengan geser-kiri yang selalu memajukan indeks.

```
# wrong_delete_value.py - penyelesaian keliru yang menggoda untuk Soal 4.10:
# setelah sisanya digeser ke kiri, i maju dan melewatkan butir yang bergeser masuk.
a = [7, 3, 3, 5, 3, 9]
n = len(a)
x = 3
i = 0
while i < n:
    if a[i] == x:
        for k in range(i, n - 1):       # geser a[i+1..n-1] satu tempat ke kiri
            a[k] = a[k + 1]
        n -= 1
    i += 1                              # selalu maju: inilah kesalahannya
print(a[:n])
```

```
[7, 3, 5, 9]
```

Satu angka 3 selamat. **Gambar 4.11**(b) menunjukkan sebabnya: pada `i = 1` program menemukan 3 dan menggeser semua yang sesudahnya ke kiri, sehingga memindahkan angka 3 yang *kedua* ke kotak 1. Lalu `i += 1` berpindah ke kotak 2, dan angka 3 yang kini duduk di kotak 1 tidak pernah dilihat. Kesalahan ini hanya muncul ketika dua salinan x bersebelahan; pada 7, 3, 5, 3, 9 program itu berhasil, dan itulah sebabnya uji dengan duplikat bersebelahan penting. Perbaikannya adalah memajukan `i` hanya bila tidak ada yang dihapus (dalam cabang `else`), yang membuat rancangan itu benar tetapi tetap $O(n^2)$. Rancangan dua indeks menghindari persoalan itu sepenuhnya, karena ia tidak pernah memindahkan elemen yang belum dibaca.

> **UJI DIRI.** Sesuaikan **Pseudokode 4.10** untuk menghapus *duplikat* dari larik terurut, dengan menyimpan satu salinan setiap nilai: 1, 1, 2, 3, 3, 3, 7 harus menjadi 1, 2, 3, 7. Apa yang berubah, dan berapa ukuran barunya?
>
> *Jawaban.* Dalam larik terurut, duplikat sebuah nilai berdiri bersama, sehingga sebuah elemen disimpan tepat ketika ia berbeda dari elemen terakhir yang disimpan. Mulailah dengan `w = 1` (elemen pertama selalu disimpan, bila larik tidak kosong) dan biarkan `r` berjalan mulai dari 1; simpan `a[r]` bila `a[r] != a[w - 1]`. Elemen yang disimpan adalah 1, 2, 3, 7, sehingga ukuran barunya 4, tetap dalam satu lintasan. (Inilah `std::unique` dalam C++.)

4.13 Soal 4.11: Dua Harga yang Cocok dengan Voucher

4.13.1 Soalnya

> **SOAL 4.11.** Profitina Laundry menawarkan delapan layanan dengan harga (dalam rupiah) 18000, 7000, 25000, 12000, 30000, 5000, 20000, dan 15000. Seorang pelanggan memegang voucher yang membayar *tepat dua layanan yang berbeda* yang harganya berjumlah sama dengan nilai voucher itu. Untuk voucher senilai 37000, 12000, dan 11000, daftarkan setiap pasangan harga yang cocok, dengan langkah yang lebih sedikit daripada mencoba semua pasangan.

4.13.2 Memahaminya dengan tangan

Dengan kata-kata sederhana: diberikan delapan bilangan dan sebuah target, temukan pasangan-pasangan bilangan yang jumlahnya sama dengan target. Cara pertama yang jujur adalah mencoba setiap pasangan: harga pertama dengan masing-masing dari tujuh lainnya, harga kedua dengan enam sesudahnya, dan seterusnya, yaitu $\frac{8 \cdot 7}{2} = 28$ pasangan menurut Persamaan (4.1). Melakukannya dengan tangan untuk 37000 menemukan dua pasangan: $7000 + 30000$ dan $12000 + 25000$. Untuk 12000 ditemukan satu, $5000 + 7000$. Untuk 11000 tidak ada: dua layanan termurah saja sudah berharga 12000, lebih dari 11000.

Dua puluh delapan pasangan itu sedikit. Tetapi pertanyaan yang sama untuk katalog berisi sejuta produk berarti sekitar $5 \cdot 10^{11}$ pasangan, dan **Tabel 4.2** menjawab tidak. Kita menginginkan cara yang tumbuh seperti $n \log n$.

4.13.3 Merancangnya di atas kertas

Langkah 1: urutkan. Setelah diurutkan, harganya 5000, 7000, 12000, 15000, 18000, 20000, 25000, 30000.

Langkah 2: berjalan ke dalam dari kedua ujung. Taruh indeks `i` pada harga termurah dan `j` pada harga termahal, lalu lihat jumlah keduanya:

- Jika jumlahnya *sama dengan* target, kita memperoleh sebuah pasangan. Catat, lalu gerakkan kedua indeks ke dalam.
- Jika jumlahnya *terlalu kecil*, harga di `i` tidak mungkin menjadi bagian pasangan mana pun: pasangan termahalnya yang mungkin, `a[j]`, pun belum cukup, dan setiap pasangan lain yang masih tersedia lebih murah. Maka kita membuang `a[i]` dengan menggeser `i` satu langkah ke kanan.
- Jika jumlahnya *terlalu besar*, harga di `j` tidak mungkin menjadi bagian pasangan mana pun: pasangan termurahnya yang mungkin, `a[i]`, pun sudah terlalu banyak. Maka kita membuang `a[j]` dengan menggeser `j` satu langkah ke kiri.

Perjalanan berakhir ketika kedua indeks bertemu. **Pseudokode 4.11** adalah caranya, dan **Gambar 4.12** menelusuri voucher 37000: lima langkah, bukan 28 pasangan.

Pseudokode 4.11 Jalan dua indeks yang menemukan setiap pasangan berjumlah tertentu dalam larik terurut.

```
PAIRS-WITH-SUM(a, n, target)            // a harus terurut naik
1  i = 0;  j = n - 1
2  while i < j
3      s = a[i] + a[j]
4      if s == target
5          report (a[i], a[j]);  i = i + 1;  j = j - 1
6      elseif s < target
7          i = i + 1                    // a[i] terlalu murah bagi pasangan mana pun
8      else j = j - 1                   // a[j] terlalu mahal bagi pasangan mana pun
```



Gambar 4.12 Jalan dua indeks untuk voucher 37000 pada harga yang terurut, dalam ribuan rupiah: setiap langkah menemukan sebuah pasangan atau membuang harga termurah atau termahal yang tersisa.

Mengapa perjalanan itu tidak pernah melewatkan sebuah pasangan? Setiap langkah membuang satu atau dua harga, dan kita telah menunjukkan bahwa harga yang dibuang tidak mungkin menjadi bagian pasangan mana pun *dengan harga-harga yang masih berada di antara kedua indeks*; ia juga tidak mungkin berpasangan dengan harga yang sudah dibuang, karena harga itu dibuang dengan alasan yang sama. Jadi setiap pasangan yang ada ditemukan sebelum salah satu anggotanya dibuang. (Dengan harga-harga yang berbeda, seperti di sini, menggerakkan kedua indeks setelah cocok itu aman; uji diri subbab ini menanyakan apa yang berubah ketika ada harga yang sama.)

Biayanya. Pengurutan berbiaya $O(n\log n)$ dengan pengurutan yang baik (program kita memakai insertion sort, $O(n^2)$, karena $n = 8$). Perjalanan itu menggerakkan sedikitnya satu indeks per langkah dan berhenti ketika keduanya bertemu, sehingga melakukan paling banyak $n - 1$ langkah: $O(n)$. Totalnya didominasi oleh pengurutan: $O(n\log n)$, dibandingkan $O(n^2)$ untuk semua pasangan. Alternatifnya, juga $O(n\log n)$, adalah mengurutkan lalu mencari `target - a[i]` dengan pencarian biner untuk setiap `i`: n pencarian masing-masing $O(\log n)$.

4.13.4 Penyelesaian C++

Kode 4.41 adalah program lengkapnya.

Kode 4.41 `pair_sum.cpp`: urutkan harga, lalu temukan setiap pasangan yang cocok dengan jalan dua indeks.

```
// pair_sum.cpp - Soal 4.11: dua layanan Profitina Laundry mana yang harganya
// tepat senilai voucher? Urutkan harga, lalu berjalan ke dalam dengan
// dua indeks.
#include <iostream>
using namespace std;

const int N = 8;

void show(const char* label, const int a[]) {
    cout << label;
    for (int i = 0; i < N; i++) {
        cout << ' ' << a[i];
    }
    cout << '\n';
}

void insertionSort(int a[]) {
    for (int i = 1; i < N; i++) {
        int key = a[i], j = i - 1;
        while (j >= 0 && a[j] > key) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = key;
    }
}

// Cetak tiap pasangan a[i] + a[j] == target (i < j) larik a yang TERURUT.
void pairsWithSum(const int a[], int target) {
    int i = 0, j = N - 1, steps = 0, found = 0;
    cout << "Voucher " << target << ":";
    while (i < j) {
        steps++;
        int sum = a[i] + a[j];
        if (sum == target) {
            cout << ' ' << a[i] << " + " << a[j] << ';';
            found++;
            i++;
            j--;
        } else if (sum < target) {
            i++;                          // kurang: butir kecil lebih mahal
        } else {
            j--;                          // terlalu banyak: butir besar lebih murah
        }
    }
    if (found == 0) {
```

Kode 4.41 (Lanjutan) *pair_sum.cpp: urutkan harga, lalu temukan setiap pasangan yang cocok dengan jalan dua indeks.*

```
        cout << " no pair;";
    }
    cout << " steps " << steps << " (every pair: " << N * (N - 1) / 2 << ")\n";
}

int main() {
    int prices[N] = {18000, 7000, 25000, 12000, 30000, 5000, 20000, 15000};
    show("Prices:", prices);
    insertionSort(prices);
    show("Sorted:", prices);
    pairsWithSum(prices, 37000);
    pairsWithSum(prices, 12000);
    pairsWithSum(prices, 11000);
    return 0;
}
```

```
Prices: 18000 7000 25000 12000 30000 5000 20000 15000
Sorted: 5000 7000 12000 15000 18000 20000 25000 30000
Voucher 37000: 7000 + 30000; 12000 + 25000; steps 5 (every pair: 28)
Voucher 12000: 5000 + 7000; steps 7 (every pair: 28)
Voucher 11000: no pair; steps 7 (every pair: 28)
```

Pasangan-pasangannya cocok dengan pencarian tangan, dan hitungan langkahnya menegaskan analisisnya: 5 langkah untuk 37000 dan 7 (yaitu $n - 1$) untuk dua lainnya, dibandingkan 28 pasangan pada cara *brute force*. Untuk 12000 perjalanan itu harus menggerakkan `j` jauh ke bawah dari 30000 sampai 7000, satu langkah demi satu langkah, sebelum jumlahnya cukup kecil; itulah kasus terburuk perjalanan ini, dan tetap linear.

4.13.5 Penyelesaian Java dan Python

Kode 4.42 dan **Kode 4.43** memberikan rancangan yang sama dalam Java dan dalam Python.

Kode 4.42 `PairSum.java`: pengurutan dan jalan dua indeks yang sama dalam Java.

```java
// PairSum.java - Soal 4.11: dua layanan Profitina Laundry mana yang harganya
// tepat senilai voucher? Urutkan harga, lalu berjalan ke dalam dengan
// dua indeks.
public class PairSum {
    static void show(String label, int[] a) {
        StringBuilder sb = new StringBuilder(label);
        for (int x : a) {
            sb.append(' ').append(x);
        }
        System.out.println(sb);
    }

    static void insertionSort(int[] a) {
        for (int i = 1; i < a.length; i++) {
            int key = a[i], j = i - 1;
            while (j >= 0 && a[j] > key) {
                a[j + 1] = a[j];
                j--;
            }
            a[j + 1] = key;
        }
    }

    // Cetak tiap pasangan a[i] + a[j] == target (i < j) larik a yang TERURUT.
    static void pairsWithSum(int[] a, int target) {
        int n = a.length, i = 0, j = n - 1, steps = 0, found = 0;
        StringBuilder sb = new StringBuilder("Voucher " + target + ":");
        while (i < j) {
            steps++;
            int sum = a[i] + a[j];
            if (sum == target) {
                sb.append(' ').append(a[i]).append(" + ").append(a[j])
                  .append(';');
                found++;
                i++;
                j--;
            } else if (sum < target) {
                i++;                    // kurang: butir kecil lebih mahal
            } else {
                j--;                    // terlalu banyak: butir besar lebih murah
            }
        }
        if (found == 0) {
            sb.append(" no pair;");
        }
        sb.append(" steps ").append(steps).append(" (every pair: ")
```

Kode 4.42 (Lanjutan) *PairSum.java: pengurutan dan jalan dua indeks yang sama dalam Java.*

```
            .append(n * (n - 1) / 2).append(')');
        System.out.println(sb);
    }

    public static void main(String[] args) {
        int[] prices = {18000, 7000, 25000, 12000, 30000, 5000, 20000, 15000};
        show("Prices:", prices);
        insertionSort(prices);
        show("Sorted:", prices);
        pairsWithSum(prices, 37000);
        pairsWithSum(prices, 12000);
        pairsWithSum(prices, 11000);
    }
}
```

Kode 4.43 `pair_sum.py`: pengurutan dan jalan dua indeks yang sama dalam Python.

```
# pair_sum.py - Soal 4.11: dua layanan Profitina Laundry mana yang harganya tepat
# senilai voucher? Urutkan harga, lalu berjalan ke dalam dengan dua indeks.


def insertion_sort(a):
    for i in range(1, len(a)):
        key, j = a[i], i - 1
        while j >= 0 and a[j] > key:
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key


def pairs_with_sum(a, target):
    """Cetak tiap pasangan a[i] + a[j] == target (i < j) list a TERURUT."""
    n = len(a)
    i, j, steps, text = 0, n - 1, 0, f"Voucher {target}:"
    found = 0
    while i < j:
        steps += 1
        total = a[i] + a[j]
        if total == target:
            text += f" {a[i]} + {a[j]};"
            found += 1
            i += 1
            j -= 1
        elif total < target:
            i += 1                      # kurang: butir kecil lebih mahal
        else:
            j -= 1                      # terlalu banyak: butir besar lebih murah
    if found == 0:
        text += " no pair;"
    print(f"{text} steps {steps} (every pair: {n * (n - 1) // 2})")


prices = [18000, 7000, 25000, 12000, 30000, 5000, 20000, 15000]
print("Prices:", *prices)
insertion_sort(prices)
print("Sorted:", *prices)
for voucher in (37000, 12000, 11000):
    pairs_with_sum(prices, voucher)
```

Keduanya mencetak persis keluaran program C++. Versi Java menyusun setiap baris keluaran dalam sebuah `StringBuilder`, cara yang efisien untuk menyambung banyak potongan teks kecil di Java; versi Python mengumpulkan baris itu dalam string `text` lalu mencetaknya sekali. Python menawarkan rancangan ketiga yang layak diketahui: telusuri harga-harga sekali, simpan sebuah

set berisi harga yang sudah terlihat, dan untuk setiap harga p tanyakan apakah `target - p` ada di dalam set itu. Keanggotaan set rata-rata memerlukan $O(1)$ langkah (Bab 15), sehingga seluruh pencarian menjadi $O(n)$ tanpa pengurutan sama sekali, dengan harga memori tambahan untuk set itu.

4.13.6 Penyelesaian keliru yang menggoda

Jalan dua indeks begitu pendek sehingga menggoda untuk melupakan langkah yang membuatnya berhasil. **Kode 4.44** berjalan ke dalam pada harga-harga sebagaimana adanya.

Kode 4.44 `wrong_pair_sum.cpp`: jalan dua indeks pada harga yang tidak pernah diurutkan.

```
// wrong_pair_sum.cpp - penyelesaian keliru yang menggoda untuk Soal 4.11:
// jalan dua indeks pada harga yang tidak pernah diurutkan.
#include <iostream>
using namespace std;

int main() {
    const int N = 8;
    int prices[N] = {18000, 7000, 25000, 12000, 30000, 5000, 20000, 15000};
    int targets[] = {37000, 12000, 11000};
    for (int target : targets) {
        int i = 0, j = N - 1, found = 0;
        cout << "Voucher " << target << ":";
        while (i < j) {
            int sum = prices[i] + prices[j];
            if (sum == target) {
                cout << ' ' << prices[i] << " + " << prices[j] << ';';
                found++;
                i++;
                j--;
            } else if (sum < target) {
                i++;
            } else {
                j--;
            }
        }
        if (found == 0) {
            cout << " no pair;";
        }
        cout << '\n';
    }
    return 0;
}
```

```
Voucher 37000: no pair;
Voucher 12000: no pair;
Voucher 11000: no pair;
```

Tiga voucher, tiga kali `no pair`, padahal dua di antaranya memiliki pasangan. Pada data yang tidak terurut, "jumlahnya terlalu kecil" tidak lagi berarti `a[i]` terlalu murah bagi *setiap* pasangan, karena pasangan yang lebih mahal bisa bersembunyi di mana saja dalam larik. Argumen Subbab 4.13.3 sepenuhnya bertumpu pada keterurutan, dan tanpanya perjalanan itu membuang harga-harga yang menjadi anggota pasangan. Seperti pencarian biner pada Soal 4.3, jalan dua indeks memiliki prasyarat yang tidak memberi pesan galat apa pun ketika dilanggar.

> **UJI DIRI.** Layanan kesembilan seharga 18500 ditambahkan, dan seorang pelanggan membawa voucher 36000. Telusurilah perjalanan itu pada kesembilan harga yang terurut, dan sebutkan pasangan mana saja yang dilaporkannya.
>
> *Jawaban.* Terurut: 5000, 7000, 12000, 15000, 18000, 18500, 20000, 25000, 30000. Dalam ribuan: $5 + 30 = 35 < 36$, `i++`; $7 + 30 = 37 > 36$, `j--`; $7 + 25 = 32 < 36$, `i++`; $12 + 25 =$

> $37 > 36$, `j--`; $12 + 20 = 32 < 36$, `i++`; $15 + 20 = 35 < 36$, `i++`; $18 + 20 = 38 > 36$, `j--`; $18 + 18.5 = 36.5 > 36$, `j--`; sekarang `i = j`, berhenti. Tidak ada pasangan, setelah 8 langkah. (Memang tidak ada dua harga berbeda yang berjumlah 36000; $18000 + 18000$ akan cocok, tetapi hanya ada satu layanan seharga 18000.)

4.14 Soal-Soal Berdampingan

Sebelas soal dapat mengabur menjadi satu. **Tabel 4.5** menjajarkannya dengan gagasan yang paling baik diajarkan masing-masing, biayanya, dan jebakan yang menjerat penyelesaian keliru yang menggoda. Jika Anda hanya mengingat kolom paling kanan, Anda akan terhindar dari sebagian besar kesalahan yang ditulis pemula ketika pertama kali mempelajari struktur data.

Tabel 4.5 Kesebelas soal dalam bab ini, gagasan yang diajarkan masing-masing, biayanya, dan jebakan yang diperagakan penyelesaian keliru yang menggodanya.

Soal	Gagasan kunci	Biaya	Jebakan penyelesaian keliru yang menggoda
4.1 Kata	pengurutannya tetap, perbandingannya berganti	$O(n^2 L)$	`<` biasa membandingkan kode karakter
4.2 Karakter berulang	nilainya menjadi indeks: pencacah	$O(n)$	berhenti pada karakter pertama yang pernah terlihat
4.3 Uang menurut nama	rekaman; pencarian sekuensial	$O(n)$ per kueri	pencarian biner pada data tak terurut
4.4 Kepentingan	dua kunci dalam satu aturan; kestabilan	$O(n^2 L)$	dua pengurutan stabil dengan urutan keliru
4.5 Alamat	$b + (rC + c)s$	$O(1)$	melupakan s; baris alih-alih kolom
4.6 Menghitung kunci	pencarian biner untuk batas	$O(\log n)$	rentang $[0, n-1]$ tak dapat menjawab n
4.7 Interpolasi	perkirakan posisi dari nilai	$O(\log\log n)$ rata-rata	tanpa penjaga rentang: indeks di luar batas
4.8 Menelusuri pengurutan	pertukaran dan pergeseran sama dengan inversi	$O(n^2)$	gelung dalam satu langkah terlalu jauh
4.9 Penggabungan	tiga indeks, lalu salin ekornya	$O(n + m)$	melupakan ekor sisa
4.10 Penghapusan	indeks baca dan indeks tulis	$O(n)$	menggeser, lalu melewatkan butir berikutnya
4.11 Pasangan voucher	urutkan, lalu berjalan ke dalam	$O(n \log n)$	berjalan pada data tak terurut

Tiga pola menonjol. Pertama, *prasyarat*: pencarian biner (Soal 4.3 dan 4.6), pencarian interpolasi (4.7), penggabungan (4.9), dan jalan dua indeks (4.11) semuanya hanya benar pada data terurut, dan tidak satu pun melaporkan galat ketika datanya tidak terurut; semuanya sekadar menjawab dengan salah. Kedua, *batas*: indeks yang berjalan satu langkah terlalu jauh (4.2 dan 4.8), rentang yang tidak dapat menjangkau lewat ujung (4.6), dan ekor yang tidak pernah disalin (4.9) semuanya kesalahan sejenis di tepi sebuah larik. Ketiga, *contoh-contoh tepi menemukan setiap kesalahan*: kata berhuruf campuran, string `xyzzyx`, buku terakhir di rak, kunci di luar rentang, dua nilai sama yang bersebelahan. Menguji tepi bukanlah tambahan; *itulah* pengujian.

4.15 Profitina dalam Praktik

Profitina (app.profitina.com) adalah platform visibilitas AI: ia mengukur seberapa siap situs web sebuah merek untuk dikutip oleh mesin penjawab AI, menjelaskan setiap sinyal yang diukurnya,

dan membantu pemiliknya memperbaiki apa yang ditemukannya. Soal-soal dalam bab ini kecil, tetapi bentuk-bentuknya muncul dalam produk semacam itu dalam ukuran penuh.

Pengurutan dengan pemutus seri. Komunitas Profitina memeringkat anggotanya pada papan peringkat PROFIT, dengan PROFIT sebagai koin utilitas platform itu, yang diperoleh dengan memperbaiki respons AI. Sebuah peringkat adalah Soal 4.4 dalam kostum lain: kunci utama (jumlah yang diperoleh) dan kunci kedua yang memutus seri, diterapkan dengan pengurutan stabil atau digabungkan dalam satu aturan perbandingan, agar dua anggota dengan jumlah yang sama selalu tampil dalam urutan yang sama dan daftarnya tidak berkedip-kedip di antara dua kali pemuatan halaman.

Mencari di dalam waktu. Dasbor analitik historis menampilkan tren skor mingguan untuk setiap merek dan mengekspornya sebagai CSV. Riwayat pemindaian adalah larik yang terurut menurut waktu *dengan sendirinya*, karena pemindaian ditambahkan menurut urutan terjadinya. "Semua pemindaian antara 1 Juli dan 30 September" dengan demikian persis uji diri Soal 4.6: dua pencarian batas dan satu pengurangan, $O(\log n)$ sepanjang apa pun riwayatnya tumbuh, alih-alih satu lintasan atas setiap pemindaian yang pernah dilakukan.

Menggabungkan aliran terurut. Dasbor itu dirancang sebagai kokpit: satu permintaan mengumpulkan pemberitahuan pengguna dari banyak jenis (balasan, suka, *repost*, pesan langsung, sematan, status umpan balik, pembelian PROFIT) ke dalam satu daftar. Ketika setiap jenis sudah tiba terurut menurut waktu, cara yang wajar untuk menggabungkannya menjadi satu daftar berurutan waktu adalah penggabungan pada Soal 4.9, yang diperumum dari dua daftar menjadi beberapa, alih-alih mengurutkan ulang segalanya.

Mencacah alih-alih membandingkan. Kanal-kanal komunitas menampilkan pencacah "N baru", dan pencacah itu hanya menghitung unggahan tingkat atas. Mencacah per kategori, gagasan Soal 4.2, adalah cara agar bilangan semacam itu tetap murah: satu lintasan atas butir-butir baru, satu pencacah per kanal, tanpa membandingkan setiap butir dengan setiap butir lainnya.

Latihan yang dapat dijalankan dalam buku ini tetap memakai Profitina Laundry, studi kasus pengajaran. Soal 4.11 adalah kemunculan pertamanya di sini: delapan layanan, harganya dalam rupiah, dan sebuah voucher, diselesaikan dengan pengurutan dan jalan dua indeks. Bab-bab berikutnya akan memberi binatu itu antrean pesanan (Bab 5), daftar pelanggan (Bab 7), dan tabel pencarian harga (Bab 15).

4.16 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Memakai pencarian biner pada data yang tidak terurut. Tidak ada galat, hanya jawaban yang salah (Soal 4.3). Hal yang sama berlaku bagi pencarian interpolasi, penggabungan, dan jalan dua indeks. Sebelum memakai salah satunya, tunjuklah baris program Anda yang menjamin keterurutan itu.

Membandingkan string menurut kode karakter padahal yang dimaksud urutan kamus. Dalam ASCII setiap huruf kapital mendahului setiap huruf kecil, sehingga `<` biasa menaruh `Cherry` sebelum `apple` (Soal 4.1). Putuskan apa arti "menurut abjad" dan tulislah aturan perbandingannya sendiri.

Salah membaca pernyataan yang pendek. "Karakter duplikat pertama" memiliki dua tafsiran, dan hanya contoh-contohnya yang memutuskan di antara keduanya (Soal 4.2). Periksalah tafsiran Anda terhadap *setiap* contoh, bukan hanya yang pertama.

Membiarkan gelung berjalan satu langkah terlalu jauh. `i <= n` padahal yang dimaksud `i < n`, atau gelung dalam bubble sort yang membaca `a[j + 1]` pada `j = n - 1` (Soal 4.2 dan 4.8). Telusurilah lintasan *terakhir* setiap gelung di atas kertas. Dalam C++, jalankan dengan `-fsanitize=address,undefined` selama pengembangan.

Memulai pencarian batas pada rentang $[0, n-1]$**.** Batas bawah atau batas atas bisa bernilai n, satu langkah lewat ujung (Soal 4.6). Mulailah dengan `hi = n`.

Menghitung titik tengah sebagai `(lo + hi) / 2` pada larik yang sangat besar. Dalam C++ dan Java jumlah itu dapat meluap; tulislah `lo + (hi - lo) / 2` [9].

Menerapkan kunci-kunci pengurutan berlapis dengan urutan keliru. Dengan pengurutan stabil, urutkan menurut kunci yang paling tidak penting lebih dulu dan yang paling penting paling akhir; atau masukkan semua kunci ke dalam satu aturan perbandingan (Soal 4.4).

Melupakan ukuran elemen dalam alamat. Indeks menghitung elemen, alamat menghitung bita; kalikan dengan ukurannya, dan dalam dua dimensi dengan banyaknya *kolom* (Soal 4.5).

Membagi sebelum mengalikan dalam aritmetika bilangan bulat. Pada Persamaan (4.6), `(key - lo_value) / (hi_value - lo_value)` hampir selalu bernilai 0 dalam pembagian bilangan bulat; kalikan lebih dulu (Soal 4.7).

Melupakan ekor penggabungan. Ketika satu larik habis, sisa larik lainnya tetap harus disalin; periksalah bahwa keluarannya memuat $n + m$ elemen (Soal 4.9).

Memajukan indeks setelah menghapus dengan penggeseran. Elemen yang berpindah ke kotak saat ini terlewati (Soal 4.10). Pakailah indeks baca dan indeks tulis.

Membandingkan string Java dengan `==`. Operator itu membandingkan referensi, bukan huruf; pakailah `equals` (Soal 4.3).

Hanya menguji contoh-contoh dalam soal. Setiap penyelesaian keliru yang menggoda dalam bab ini lolos pada sedikitnya satu contoh yang diberikan. Tambahkan masukan kosong, satu elemen, posisi pertama dan terakhir, kunci yang tidak ada, keadaan seri, dan duplikat yang bersebelahan [12].

4.17 Ringkasan Bab

Bab ini memantapkan Bab 1 sampai Bab 3 dengan menyelesaikan soal alih-alih memperkenalkan struktur baru.

Sebuah metode. Subbab 4.2 menguraikan delapan langkah: nyatakan ulang soalnya; kerjakan contoh dengan tangan; pilih struktur dan algoritma; telusuri rancangan di atas kertas; perkirakan biayanya; tulis kodenya, C++ lebih dulu; jalankan dan bandingkan dengan jejak tangan; dan tinjau ulang untuk mencari tepi, jebakan, dan varian. **Tabel 4.2**, Persamaan (4.1), dan Persamaan (4.2) mengubah langkah 5 menjadi aritmetika: $\frac{n(n-1)}{2}$ perbandingan untuk semua pasangan, dan paling banyak $\lfloor \log_2 n \rfloor + 1$ pemeriksaan untuk pencarian biner.

Empat soal klasik. Kata-kata diurutkan menurut urutan kamus oleh insertion sort yang satu-satunya bagian barunya adalah aturan perbandingan (Soal 4.1). Karakter pertama yang berulang ditemukan dalam $O(n)$ dengan satu pencacah untuk setiap karakter yang mungkin, setelah contoh-contohnya memutuskan tafsiran mana dari dua tafsiran yang dimaksud soal (Soal 4.2). Uang dicari menurut nama dengan pencarian sekuensial pada larik rekaman, dan pencarian biner pada data tak terurut terbukti gagal tanpa suara (Soal 4.3). String diurutkan menurut dua kunci

dalam satu pengurutan stabil, dan dua pengurutan stabil dengan urutan keliru terbukti menyerahkan kendali kepada kunci yang kurang penting (Soal 4.4).

Tujuh soal baru. Alamat larik dengan Persamaan (4.3) dan Persamaan (4.4), ditegaskan terhadap memori nyata (Soal 4.5); menghitung kunci dengan dua pencarian batas dan Persamaan (4.5) (Soal 4.6); pencarian interpolasi dengan Persamaan (4.6), beserta penjaga yang menjaga pemeriksaannya tetap di dalam larik (Soal 4.7); ketiga pengurutan dasar ditelusuri lintasan demi lintasan, dengan pertukaran dan pergeseran yang sama banyaknya dengan inversi (Soal 4.8); penggabungan dua larik terurut dalam $O(n + m)$ (Soal 4.9); penghapusan di tempat dengan indeks baca dan indeks tulis (Soal 4.10); dan pasangan berjumlah tertentu dengan mengurutkan lalu berjalan ke dalam (Soal 4.11).

Menatap ke depan. Bab 5 membangun *stack* dan *queue* tepat di atas gagasan larik-dengan-ukuran pada Soal 4.10, dan Bab 6 membuka pointer dan referensi yang telah dipakai Soal 4.3 dan 4.5 tanpa komentar.

4.18 Soal Latihan

4.1 Telusurilah insertion sort dari Soal 4.1 pada lima kata `delta Alpha charlie alpha Bravo`, dengan menuliskan larik itu setelah setiap langkah. Apa hasil terurutnya?

4.2 Untuk string `abcba` dan `programming`, berikan jawaban program yang benar dari Soal 4.2 dan jawaban penyelesaian keliru yang menggoda. Pada string yang mana keduanya berbeda, dan mengapa?

4.3 Berapa perbandingan nama yang dilakukan pencarian sekuensial Soal 4.3 untuk setiap dari kelima kueri dalam `names.txt`, dan berapa totalnya?

4.4 Urutkan string `wow!!`, `ok`, `yes!`, `no!`, `hm` (dengan urutan masukan ini) menurut aturan kepentingan Soal 4.4. Dua string mana yang seri pada kedua kunci, dan dengan urutan apa keduanya muncul?

4.5 Larik `int C[4][6]` berisi bilangan bulat 4 bita dimulai di `0x3000`. (a) Berapakah alamat `C[3][2]`? (b) Elemen mana yang dimulai di alamat `0x3034`?

4.6 Dengan memakai batas bawah dan batas atas Soal 4.6 pada rak tahun terbit, berapa buku yang terbit dari 2005 sampai 2019 inklusif? Sebutkan kedua panggilan yang Anda lakukan dan hasilnya.

4.7 Telusurilah pencarian interpolasi dan pencarian biner untuk kunci 63 pada delapan kode buku Soal 4.7. Berapa pemeriksaan yang dilakukan masing-masing?

4.8 Telusurilah selection sort pada 4, 2, 5, 1, 3: tuliskan larik itu setelah setiap lintasan, lalu hitunglah perbandingan dan pertukarannya. Berapa pertukaran yang akan dilakukan bubble sort pada larik yang sama, dan mengapa Anda dapat menjawabnya tanpa menelusurinya?

4.9 Gabungkan 1, 4, 7 dengan 2, 3, 8, 9 menurut **Pseudokode 4.9**. Berapa perbandingan yang dilakukan, dan elemen mana yang disalin oleh gelung ekor?

4.10 Pakailah **Pseudokode 4.10**, dengan uji “simpan `a[r]` bila ganjil”, untuk menghapus semua bilangan genap dari 2, 7, 4, 4, 9, 6, 1. Berikan nilai `w` setelah setiap putaran dan hasilnya.

4.11 Pada harga binatu terurut dari Soal 4.11, telusurilah jalan dua indeks untuk voucher 45000. Pasangan mana yang dilaporkan, dan setelah berapa langkah?

4.12 Terapkan delapan langkah Subbab 4.2 pada soal berikut, dengan menuliskan setiap langkahnya: "Diberikan sebuah larik bilangan bulat, temukan nilai *berbeda* terbesar kedua, atau laporkan bahwa nilai itu tidak ada."

4.19 Penyelesaian Soal Latihan

4.1 Dalam huruf kecil kata-katanya adalah delta, alpha, charlie, alpha, bravo. Langkah $i = 1$: `Alpha` didahulukan dari `delta`, menghasilkan `Alpha delta charlie alpha Bravo`. Langkah 2: `charlie` masuk di antara keduanya: `Alpha charlie delta alpha Bravo`. Langkah 3: `alpha` melewati `delta` dan `charlie`; terhadap `Alpha` kedua salinan huruf kecilnya seri, dan `alpha` < `Alpha` bernilai salah menurut urutan kode (97 > 65), sehingga ia berhenti: `Alpha alpha charlie delta Bravo`. Langkah 4: `Bravo` melewati `delta` dan `charlie` lalu berhenti setelah `alpha`: `Alpha alpha Bravo charlie delta`. Itulah hasilnya.

4.2 `abcba`: hitungannya a 2, b 2, c 1; program yang benar menjawab `a` (karakter paling kiri dengan hitungan sedikitnya 2), sedangkan program yang keliru melihat a, b, c lalu b lagi dan menjawab `b`. `programming`: hitungannya p 1, r 2, o 1, g 2, a 1, m 2, i 1, n 1; jawaban yang benar adalah `r` (indeks 1); program yang keliru melihat p, r, o, g lalu r lagi, dan juga menjawab `r`. Keduanya berbeda pada `abcba` karena di sana karakter yang kemunculan pertamanya paling kiri (`a`) berulang *lebih lambat* daripada karakter lain (`b`).

4.3 `Pujo` ditemukan di indeks 3 setelah 4 perbandingan; `Komeng` tidak ada, 5 perbandingan; `Lestari` di indeks 4, 5 perbandingan; `pujo` tidak ada, 5; `Puspita` di indeks 0, 1. Totalnya $4 + 5 + 5 + 5 + 1 = 20$ perbandingan.

4.4 Kuncinya (tanda seru, panjang): `wow!!` (2, 5), `ok` (0, 2), `yes!` (1, 4), `no!` (1, 3), `hm` (0, 2). Urutannya `wow!!`, lalu string bertanda satu yang lebih pendek dulu, `no!`, `yes!`, lalu string bertanda nol. `ok` dan `hm` seri pada kedua kunci, dan pengurutan stabil mempertahankan urutan masukannya: `ok`, `hm`.

4.5 (a) Menurut Persamaan (4.4), $k = 3 \cdot 6 + 2 = 20$ dan selisihnya $20 \cdot 4 = 80$ bita, `0x50`, sehingga alamatnya `0x3050`. (b) Selisihnya 0x34 = 52 bita, yaitu $52/4 = 13$ elemen; $13 = 2 \cdot 6 + 1$, sehingga elemennya `C[2][1]`.

4.6 Buku-buku dengan $2005 \le year \le 2019$ terletak mulai dari `lower(2005)` sampai, tetapi tidak termasuk, `upper(2019)`. `lower(2005)` = 4 (tahun 2005 di indeks 4) dan `upper(2019)` = 8 (tahun pertama yang lebih besar dari 2019 adalah 2023 di indeks 8), sehingga hitungannya $8 - 4 = 4$: 2005, 2010, 2010, dan 2019.

4.7 Interpolasi: $0 + \lfloor 38 \cdot 7/71 \rfloor = \lfloor 3.75 \rfloor = 3$, dan $a[3] = 56 < 63$, sehingga `low` = 4; lalu $4 + \lfloor 0 \cdot 3/33 \rfloor = 4$, dan $a[4] = 63$: ketemu setelah 2 pemeriksaan. Pencarian biner: `mid` = 3 (56 < 63, `low` = 4), `mid` = 5 (72 > 63, `high` = 4), `mid` = 4 (ketemu): 3 pemeriksaan.

4.8 Lintasan 1: yang terkecil dari kelimanya adalah 1 di indeks 3; tukar dengan indeks 0: 1, 2, 5, 4, 3. Lintasan 2: yang terkecil dari 2, 5, 4, 3 adalah 2, sudah di tempatnya, tanpa pertukaran. Lintasan 3: yang terkecil dari 5, 4, 3 adalah 3 di indeks 4; tukar: 1, 2, 3, 4, 5. Lintasan 4: yang terkecil dari 4, 5 adalah 4, sudah di tempatnya. Perbandingan $4 + 3 + 2 + 1 = 10$, pertukaran 2. Bubble sort melakukan satu pertukaran per inversi; inversinya (4,2), (4,1), (4,3), (2,1), (5,1), (5,3), sehingga ia melakukan 6 pertukaran.

4.9 1 lawan 2, ambil 1; 4 lawan 2, ambil 2; 4 lawan 3, ambil 3; 4 lawan 8, ambil 4; 7 lawan 8, ambil 7. Larik pertama habis setelah 5 perbandingan, dan gelung ekor menyalin 8 dan 9. Hasilnya 1, 2, 3, 4, 7, 8, 9.

4.10 `r` = 0: 2 genap, lewati, `w` = 0. `r` = 1: 7 disimpan, `w` = 1. `r` = 2 dan 3: 4 dan 4 dilewati, `w` = 1. `r` = 4: 9 disimpan, `w` = 2. `r` = 5: 6 dilewati. `r` = 6: 1 disimpan, `w` = 3. Hasilnya 7, 9, 1, ukuran baru 3; kedua angka 4 yang bersebelahan tidak menimbulkan masalah, karena tidak ada yang digeser.

4.11 Dalam ribuan: $5 + 30 = 35 < 45$, `i++`; $7 + 30 = 37$, `i++`; $12 + 30 = 42$, `i++`; $15 + 30 = 45$, pasangan, keduanya bergerak; $18 + 25 = 43$, `i++`; $20 + 25 = 45$, pasangan, keduanya bergerak; sekarang `i` > `j`, berhenti. Dua pasangan, 15000 + 30000 dan 20000 + 25000, setelah 6 langkah.

4.12 Langkah 1: masukannya larik berisi n bilangan bulat; keluarannya nilai terbesar yang benar-benar lebih kecil daripada maksimumnya, atau "tidak ada". Langkah 2: 4, 9, 2, 9, 7 memberi 7 (9 yang kedua tidak berbeda); 5, 5, 5 memberi tidak ada; satu elemen memberi tidak ada; $-3, -8$ memberi -8. Langkah 3: sebuah larik dan satu lintasan, dengan dua variabel, `first` (terbesar sejauh ini) dan `second` (terbesar sejauh ini yang lebih kecil daripada `first`), beserta penanda "belum terlihat". Langkah 4: telusuri 4, 9, 2, 9, 7: first 4; first 9, second 4; 2 tidak mengubah apa pun; 9 sama dengan first, tidak ada perubahan; 7 terletak di antara keduanya, second 7. Langkah 5: satu lintasan, waktu $O(n)$, ruang $O(1)$; mengurutkan lebih dulu akan berbiaya $O(n \log n)$. Langkah 6: tulislah dalam C++, lalu Java dan Python. Langkah 7: bandingkan dengan jejak-jejak langkah 2. Langkah 8: penyelesaian keliru yang menggoda mengurutkan lalu mengembalikan `a[n-2]`, yang menjawab 9 untuk 4, 9, 2, 9, 7, karena melupakan kata *berbeda*; sebuah variannya menanyakan nilai berbeda terbesar ke-k.

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

[3] G. Pólya, *How to Solve It: A New Aspect of Mathematical Method*, 2nd ed. Princeton, NJ, USA: Princeton University Press, 1957.

[4] American National Standards Institute, *Coded Character Set — 7-Bit American National Standard Code for Information Interchange*, ANSI X3.4-1986, 1986.

[5] cppreference.com, "Algorithms library" (`std::sort`, `std::stable_sort`, `std::lower_bound`, `std::upper_bound`, `std::equal_range`, `std::merge`, `std::remove`, `std::unique`). Diakses: 24 September 2026. [Daring]. Tersedia: https://en.cppreference.com/w/cpp/algorithm

[6] Oracle, *Java Platform, Standard Edition 25 API Specification*, class `java.util.Arrays`, 2025. Diakses: 24 September 2026. [Daring]. Tersedia: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Arrays.html

[7] Python Software Foundation, "Sorting Techniques," "bisect — Array bisection algorithm," "heapq — Heap queue algorithm," and "collections — Container datatypes," Python 3.14 documentation. Diakses: 24 September 2026. [Daring]. Tersedia: https://docs.python.org/3/

[8] B. Stroustrup, *A Tour of C++*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2022.

[9] J. Bloch, "Extra, Extra — Read All About It: Nearly All Binary Searches and Mergesorts are Broken," Google Research Blog, Jun. 2006. [Daring]. Tersedia: https://research.google/blog/extra-extra-read-all-about-it-nearly-all-binary-searches-and-mergesorts-are-broken/

[10] Y. Perl, A. Itai, and H. Avni, "Interpolation search—a log log N search," *Communications of the ACM*, vol. 21, no. 7, pp. 550–553, 1978.

[11] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011.

[12] J. Bentley, *Programming Pearls*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2000.

Bab 5
Stack dan Queue

Setelah menyelesaikan bab ini, Anda akan mampu menjelaskan — dengan gambar untuk setiap langkah — apa itu *stack* (tumpukan) dan *queue* (antrean), dan mengapa keduanya sengaja melarang Anda menyentuh sebagian besar datanya sendiri; mengimplementasikan keduanya di atas larik biasa dalam C++, Java, dan Python, dengan setiap operasi berjalan dalam waktu konstan; mengenali empat pekerjaan klasik yang dilakukan stack dalam perangkat lunak nyata — mencocokkan tanda kurung, membatalkan suntingan, menerjemahkan aritmetika dari cara manusia menulisnya ke cara mesin mengevaluasinya, dan mencatat fungsi mana yang sedang berjalan — lalu menulis masing-masing sendiri; mendiagnosis *bug* "pergeseran" (*drifting*) yang membuat queue larik yang naif menolak data baru padahal separuh slotnya kosong, lalu menyembuhkannya dengan aritmetika modular; memilih di antara dua cara klasik untuk membedakan queue sirkular yang penuh dari yang kosong; memperluas queue menjadi *double-ended queue* dan melihat sekilas *priority queue*; serta memakai versi siap pakai dalam `std::stack`, `std::queue`, `std::deque`, `ArrayDeque` di Java, dan `list`, `collections.deque`, serta `queue` di Python, lengkap dengan alasan memilih yang mana. Bab ini dibuka dengan teka-teki yang sangat tua tentang para biarawan dan cakram emas, karena teka-teki itu sesungguhnya adalah stack yang menyamar.

5.1 Posisi Kita Sejauh Ini

Bab 1 sampai Bab 3 memperlakukan larik sebagai ruangan dengan pintu terbuka. Bab 1 menyimpan katalog perpustakaan Pustaka Ceria dalam memori yang bersebelahan dan memperlihatkan bahwa elemen mana pun, $A[i]$, dapat dicapai dalam satu langkah, karena alamatnya adalah alamat dasar ditambah i kali ukuran elemen. Bab 2 menelusuri larik itu — secara berurutan ketika belum terurut, dan dengan membagi dua ketika sudah terurut — dan Bab 3 mengurutkannya, memindahkan elemen ke posisi mana pun yang dikehendaki algoritmanya. Bab 4 kemudian membahas soal-soal utuh dari jenis yang persis sama. Dalam keempat bab itu algoritma memiliki kebebasan penuh: membaca elemen mana saja, menulis elemen mana saja, dalam urutan apa saja.

Bab ini sengaja mencabut kebebasan itu.

5.1.1 Dari akses bebas ke akses berdisiplin

Bayangkan troli pengembalian buku di meja depan Pustaka Ceria. Buku yang dikembalikan diletakkan di puncak tumpukan; pustakawan yang mengembalikannya ke rak selalu mengangkat buku teratas lebih dulu. Tidak ada yang menarik buku dari tengah — tumpukannya bisa roboh. Sekarang bayangkan meja buku baru di ruang belakang, tempat buku yang baru dibongkar menunggu dalam barisan untuk dikatalogkan. Buku yang tiba pertama dikatalogkan pertama; kardus baru bergabung di ujung barisan. Tidak ada yang menyerobot — itu tidak adil bagi buku yang sudah menunggu paling lama.

Keduanya adalah kumpulan buku, dan keduanya dapat disimpan dalam larik. Namun pada keduanya hanya **satu** posisi yang pernah disentuh: puncak tumpukan pada kasus pertama, dan

kedua ujung barisan pada kasus kedua. Selebihnya terlarang. Pembatasan ini terdengar seperti kerugian. Kenyataannya, ia sumber tiga keuntungan:

- **Kesederhanaan.** Operasi yang hanya boleh menyentuh satu posisi yang sudah diketahui tidak memerlukan pencarian ataupun pergeseran. Setiap operasi pada struktur dasar bab ini berjalan dalam waktu $O(1)$ — waktu konstan, tidak bergantung pada banyaknya elemen yang tersimpan.
- **Kebenaran yang dijamin oleh konstruksi.** Jika satu-satunya cara mengambil buku adalah dari puncak, tidak ada potongan kode yang dapat secara tidak sengaja mengembalikan buku yang salah ke rak. Aturannya ditegakkan oleh struktur, bukan oleh ingatan pemrogram.
- **Kecocokan dengan masalah.** Banyak masalah nyata pada dasarnya bersifat *terakhir masuk, pertama keluar* atau *pertama masuk, pertama keluar*: membatalkan suntingan, mencocokkan tanda kurung, melayani pelanggan, mencetak dokumen. Bila bentuk struktur data sama dengan bentuk masalahnya, program seolah-olah menulis dirinya sendiri.

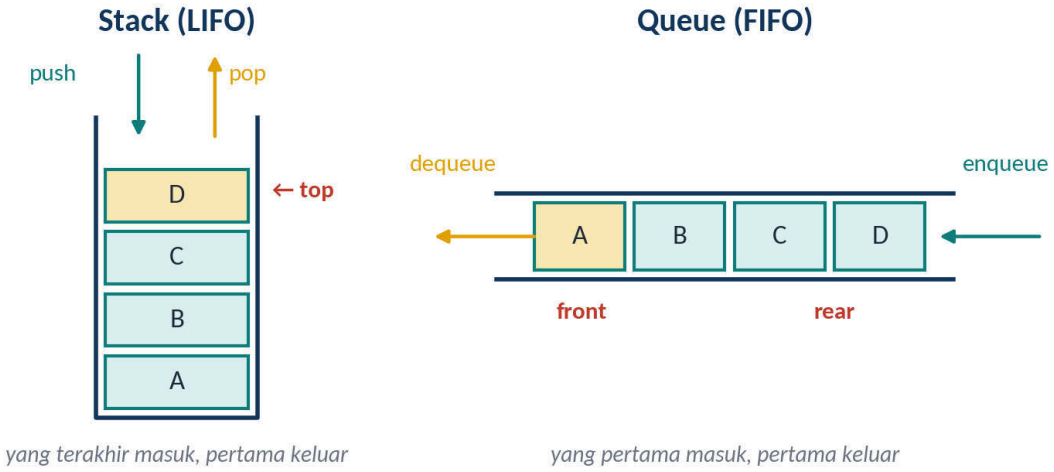
5.1.2 Tipe data abstrak

Sebelum membangun apa pun, kita memerlukan satu istilah.

> **DEFINISI.** *Tipe data abstrak* (*abstract data type*, ADT) adalah deskripsi sekumpulan nilai **beserta operasi yang diizinkan terhadapnya dan apa yang dilakukan operasi-operasi itu**, tanpa komitmen tentang cara nilai-nilai itu disimpan. *Struktur data* adalah satu cara konkret menyimpan nilai-nilai tersebut sehingga operasinya dapat dilaksanakan.

Perbedaan ini penting. "Stack" adalah ADT: ia menjanjikan `push`, `pop`, `peek`, dan `isEmpty`, dan menjanjikan bahwa `pop` mengembalikan nilai yang paling akhir di-*push* dan belum diambil. Ia tidak berkata apa-apa tentang larik atau pointer. Larik dengan indeks `top` adalah satu struktur data yang mengimplementasikan ADT stack; rangkaian simpul bertaut adalah struktur lain; `list` Python adalah yang ketiga. Kode yang memakai stack hanya melalui operasinya akan bekerja tanpa perubahan dengan yang mana pun. Cormen dkk. menyajikan stack dan queue persis seperti ini, sebagai *himpunan dinamis* yang operasi penghapusannya sudah ditentukan sebelumnya [1], dan pembahasan klasik Knuth tentang "senarai linear" menyebut tiga bentuk terbatas yang sama dengan yang kita pelajari di sini: stack, queue, dan deque [2].

Gambar 5.1 memperlihatkan kedua disiplin itu berdampingan. Pada stack di kiri, butir masuk dan keluar melalui bukaan yang sama, yaitu puncak (*top*), sehingga yang terakhir masuk adalah yang pertama keluar: **LIFO** (*last in, first out*). Pada queue di kanan, butir masuk di belakang (*rear*) dan keluar di depan (*front*), sehingga yang pertama masuk adalah yang pertama keluar: **FIFO** (*first in, first out*).



Gambar 5.1 Dua disiplin dalam bab ini. Stack menambah dan mengambil di ujung yang sama (LIFO); queue menambah di belakang dan mengambil di depan (FIFO).

5.2 Kisah Pembuka: Menara Hanoi

Pada tahun 1883 matematikawan Prancis Édouard Lucas menjual sebuah teka-teki di Paris dengan nama samaran "Professeur N. Claus de Siam" — anagram dari "Lucas d'Amiens". Kotaknya disertai sebuah legenda [3]. Konon di sebuah kuil di kota Benares terdapat tiga jarum berlian, dan pada salah satunya Tuhan menempatkan enam puluh empat cakram emas murni, yang terbesar di dasar dan setiap cakram lebih kecil daripada cakram di bawahnya. Siang dan malam para pendeta kuil memindahkan cakram-cakram itu dari jarum pertama ke jarum ketiga dengan mematuhi dua aturan yang tidak boleh dilanggar. Ketika mereka selesai, kuil itu akan runtuh menjadi debu dan dunia akan berakhir.

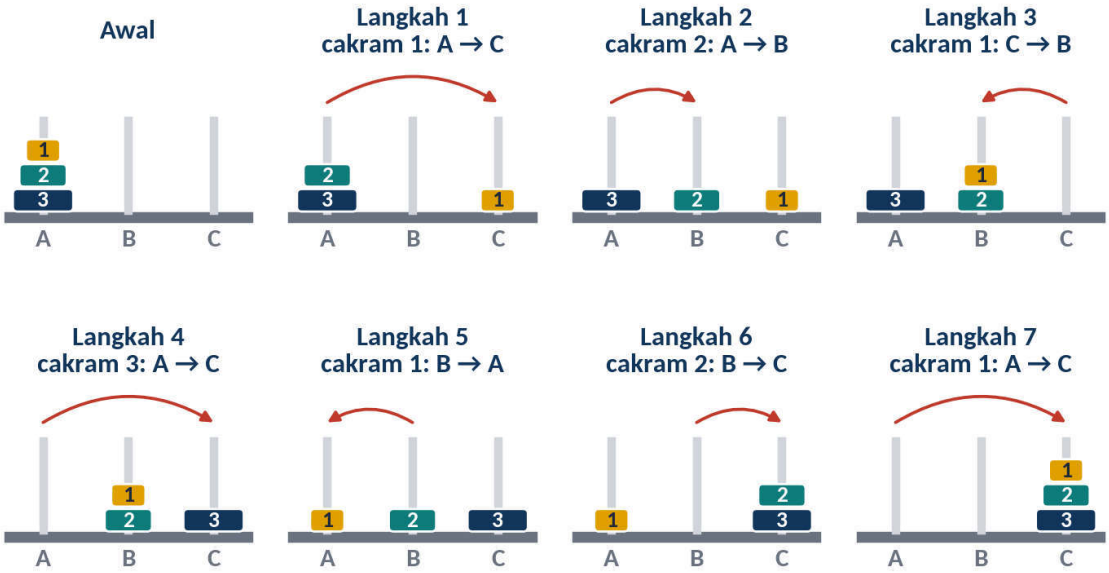
Aturannya adalah seluruh teka-teki itu:

1. Pindahkan **satu cakram setiap kali**, dan hanya cakram yang berada **di puncak** sebuah jarum.
2. Jangan pernah meletakkan sebuah cakram di atas cakram yang **lebih kecil**.

Perhatikan aturan 1 sekali lagi. Pada setiap jarum — kita akan menyebutnya *tiang* — satu-satunya cakram yang boleh diambil adalah yang di puncak, dan satu-satunya tempat meletakkan cakram adalah di puncak. Cakram yang terakhir diletakkan pada sebuah tiang adalah yang pertama dapat diangkat darinya. Setiap tiang adalah stack. Menara Hanoi adalah teka-teki tentang tiga stack ditambah satu aturan tentang ukuran.

5.2.1 Tiga cakram, dengan tangan

Mari kita namai tiangnya A (tempat cakram bermula), B (penolong), dan C (tujuan), dan nomori cakramnya 1 (terkecil) sampai 3 (terbesar). **Gambar 5.2** memperlihatkan penyelesaian terpendek, satu panel untuk setiap keadaan.



Gambar 5.2 Tujuh langkah yang memindahkan tiga cakram dari tiang A ke tiang C. Panah merah menandai langkah yang menghasilkan setiap keadaan; pada setiap langkah hanya cakram teratas yang berpindah, dan cakram itu tidak pernah mendarat di atas cakram yang lebih kecil.

Ikuti perlahan, karena polanya adalah intinya.

- **Langkah 1–3** menyingkirkan dua cakram kecil. Cakram 1 ke C, cakram 2 ke B, lalu cakram 1 naik ke atas cakram 2. Setelah langkah 3, tiang B memuat menara kecil yang benar berisi dua cakram, dan cakram 3 di A akhirnya terbuka.
- **Langkah 4** adalah satu-satunya langkah dalam seluruh penyelesaian yang menyentuh cakram 3: dari A ke C. Langkah itu hanya mungkin karena C kosong dan cakram 3 berada di puncak A.
- **Langkah 5–7** mengulangi tiga langkah pertama dalam bingkai yang berbeda: menara dua cakram di B dibawa ke atas cakram 3 di C, dengan A sebagai penolong.

Jadi, menyelesaikan tiga cakram berarti menyelesaikan dua cakram sebanyak dua kali, dengan satu pemindahan cakram besar di antaranya. Argumen yang sama berlaku untuk berapa pun banyaknya cakram, dan argumen itu memberi kita sebuah hitungan.

5.2.2 Berapa banyak langkah?

Tulis $M(n)$ untuk banyaknya langkah terkecil yang memindahkan n cakram dari satu tiang ke tiang lain. Cakram terbesar harus berpindah sedikitnya sekali. Sebelum ia dapat berpindah, semua cakram lainnya — seluruh $n-1$ cakram — harus sudah keluar dari tiangnya *dan* tidak berada di tiang tujuan, artinya semuanya bertumpuk di tiang ketiga: itu memerlukan sedikitnya $M(n-1)$ langkah. Setelah cakram terbesar mendarat, $n-1$ cakram itu harus dibawa ke atasnya, yang lagi-lagi memerlukan sedikitnya $M(n-1)$ langkah. Strategi pada **Gambar 5.2** mencapai persis jumlah itu. Jadi

$$M(n) = 2M(n-1) + 1, \qquad M(1) = 1. \tag{5.1}$$

Persamaan (5.1) adalah sebuah *relasi rekurensi*: ia mendefinisikan setiap nilai berdasarkan nilai sebelumnya. Uraikan beberapa kali untuk melihat polanya:

$$M(1) = 1, \quad M(2) = 2 \cdot 1 + 1 = 3, \quad M(3) = 2 \cdot 3 + 1 = 7, \quad M(4) = 2 \cdot 7 + 1 = 15, \quad M(5) = 31.$$

Setiap nilai kurang satu dari sebuah perpangkatan dua, dan memang

$$M(n) = 2^n - 1. \tag{5.2}$$

Untuk memeriksa Persamaan (5.2), substitusikan ke ruas kanan Persamaan (5.1): $2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1$, yang sama dengan ruas kiri, dan $2^1 - 1 = 1$ cocok dengan nilai awalnya. (Bab 10 membuktikan hasil semacam ini secara formal dengan induksi dan menyelesaikan teka-teki ini dengan fungsi rekursif; di sini kita hanya memerlukan angkanya.)

Kembali ke legenda. Dengan $n = 64$ cakram, para pendeta memerlukan $2^{64} - 1 =$ 18,446,744,073,709,551,615 langkah. Dengan satu langkah per detik, siang dan malam, itu sama dengan

$$\frac{2^{64} - 1}{60 \cdot 60 \cdot 24 \cdot 365.25} \approx 5.85 \times 10^{11} \text{ tahun}, \tag{5.3}$$

sekitar 585 miliar tahun menurut Persamaan (5.3), kira-kira empat puluh kali umur alam semesta saat ini. Dunia masih aman untuk sementara. Pelajaran bagi pemrogram lebih tajam: algoritma yang biayanya berlipat dua untuk setiap tambahan satu unit masukan menjadi tidak berdaya jauh sebelum masukannya tampak besar.

5.2.3 Menyelesaikan Hanoi dengan tiga stack tanpa rekursi

Karena setiap tiang adalah stack, kita dapat menulis program yang menyimpan ketiga tiang sebagai tiga stack dan memindahkan cakram hanya dengan `push` dan `pop`. Bab 10 akan menyelesaikan teka-teki ini dengan rekursi, cara alami untuk mengungkapkan "selesaikan $n - 1$, pindahkan satu, selesaikan $n - 1$". Namun ada pula metode nonrekursif yang indah, dan metode itu tidak memakai apa pun selain operasi stack:

- Pada setiap langkah **bernomor ganjil**, pindahkan cakram 1 satu tiang mengikuti siklus tetap: A → C → B → A → … bila n ganjil, dan A → B → C → A → … bila n genap.
- Pada setiap langkah **bernomor genap**, lakukan satu-satunya langkah sah yang tidak melibatkan cakram 1. Di antara dua tiang yang tidak memuat cakram 1 selalu ada tepat satu langkah sah: cakram puncak yang lebih kecil dipindahkan ke tiang yang lain (atau ke tiang yang kosong, bila ada yang kosong).

Mengapa pada langkah genap hanya ada satu pilihan? Cakram 1 adalah yang terkecil, sehingga ia berada di puncak tiangnya; dua tiang lainnya memiliki cakram puncak yang sama-sama lebih besar dari 1, dan di antara dua cakram puncak yang berbeda hanya yang lebih kecil yang boleh berpindah ke atas yang lebih besar. **Kode 5.1** mengimplementasikan metode ini. Setiap tiang adalah `vector<int>` yang dipakai secara ketat sebagai stack — `push_back` untuk *push*, `back` untuk *peek*, `pop_back` untuk *pop* — dan `moveDisk` menolak setiap langkah yang melanggar aturan 2.

Kode 5.1 `01_hanoi_stacks.cpp`: Menara Hanoi diselesaikan dengan tiga stack dan sebuah perulangan, disusul banyaknya langkah untuk 1 sampai 10 cakram.

```
// Towers of Hanoi solved WITHOUT recursion: three pegs, each one a stack.
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct Peg {
    char name;
    vector<int> disks;                          // back() is the top of the peg
    bool isEmpty() const { return disks.empty(); }
    int  peek()    const { return disks.back(); }
};

string show(const Peg& p) {                     // bottom -> top, e.g. "[3 2 1]"
    string s = "[";
    for (size_t i = 0; i < p.disks.size(); ++i) {
        if (i > 0) s += " ";
        s += to_string(p.disks[i]);
    }
    return s + "]";
}

// Move the top disk of `from` onto `to`; refuse a larger disk on a smaller one.
bool moveDisk(Peg& from, Peg& to) {
    if (from.isEmpty()) return false;
    if (!to.isEmpty() && to.peek() < from.peek()) return false;
    to.disks.push_back(from.peek());            // push onto the target ...
    from.disks.pop_back();                      // ... pop from the source
    return true;
}

// The only legal move between two pegs (the smaller top disk goes on the other).
void legalMoveBetween(Peg& x, Peg& y) {
    if (x.isEmpty())                moveDisk(y, x);
    else if (y.isEmpty())           moveDisk(x, y);
    else if (x.peek() < y.peek())   moveDisk(x, y);
    else                            moveDisk(y, x);
}
```

Kode 5.1 (Lanjutan) *01_hanoi_stacks.cpp: Menara Hanoi diselesaikan dengan tiga stack dan sebuah perulangan, disusul banyaknya langkah untuk 1 sampai 10 cakram.*

```
long long solve(int n, bool verbose) {
    Peg peg[3] = {{'A', {}}, {'B', {}}, {'C', {}}};
    for (int d = n; d >= 1; --d) peg[0].disks.push_back(d);
    // Disk 1 cycles A->C->B->A when n is odd, A->B->C->A when n is even.
    int step = (n % 2 == 1) ? 2 : 1;             // index distance of disk 1's moves
    int small = 0;                               // the peg that holds disk 1
    long long moves = 0;
    auto report = [&](const string& what) {
        if (!verbose) return;
        cout << left << setw(23) << what << "A: " << setw(9) << show(peg[0])
             << "B: " << setw(9) << show(peg[1]) << "C: " << show(peg[2]) << "\n";
    };
    report("start");
    while (peg[2].disks.size() != static_cast<size_t>(n)) {
        int next = (small + step) % 3;           // move disk 1 one peg along its cycle
        moveDisk(peg[small], peg[next]);
        ++moves;
        report("move " + to_string(moves) + ": disk 1 " + peg[small].name + " -> " +
peg[next].name);
        small = next;
        if (peg[2].disks.size() == static_cast<size_t>(n)) break;
        int o1 = (small + 1) % 3, o2 = (small + 2) % 3;   // the two pegs without disk 1
        Peg& x = peg[o1];
        Peg& y = peg[o2];
        char fromName = (x.isEmpty() || (!y.isEmpty() && y.peek() < x.peek())) ? y.name :
x.name;
        char toName   = (fromName == x.name) ? y.name : x.name;
        int  disk     = (fromName == x.name) ? x.peek() : y.peek();
        legalMoveBetween(x, y);
        ++moves;
        report("move " + to_string(moves) + ": disk " + to_string(disk) + " " + fromName +
" -> " + toName);
    }
    return moves;
}

int main() {
    cout << "Towers of Hanoi, 3 disks; each peg printed bottom -> top\n";
    long long m = solve(3, true);
    cout << "moves made: " << m << "\n\n";
    cout << "disks   moves   2^n - 1\n" << right;
    for (int n = 1; n <= 10; ++n)
        cout << setw(5) << n << setw(8) << solve(n, false) << setw(10) << (1LL << n) - 1 <<
"\n";
    return 0;
}
```

Setelah dikompilasi dengan `g++ -std=c++17 -O2 -Wall -Wextra` dan dijalankan, program ini mencetak:

```
Towers of Hanoi, 3 disks; each peg printed bottom -> top
start                 A: [3 2 1]  B: []       C: []
move 1: disk 1 A -> C  A: [3 2]    B: []       C: [1]
move 2: disk 2 A -> B  A: [3]      B: [2]      C: [1]
move 3: disk 1 C -> B  A: [3]      B: [2 1]    C: []
move 4: disk 3 A -> C  A: []       B: [2 1]    C: [3]
move 5: disk 1 B -> A  A: [1]      B: [2]      C: [3]
move 6: disk 2 B -> C  A: [1]      B: []       C: [3 2]
move 7: disk 1 A -> C  A: []       B: []       C: [3 2 1]
moves made: 7

disks   moves   2^n - 1
    1       1         1
    2       3         3
    3       7         7
    4      15        15
    5      31        31
    6      63        63
    7     127       127
    8     255       255
    9     511       511
   10    1023      1023
```

Blok pertama mereproduksi **Gambar 5.2** langkah demi langkah, mencetak setiap tiang dari dasar ke puncak. Tabelnya menegaskan Persamaan (5.2) untuk setiap n dari 1 sampai 10: perulangan yang tidak pernah melihat lebih jauh daripada puncak tiang mana pun ini melakukan tepat $2^n - 1$ langkah. Versi Java dan Python di folder Code mencetak baris-baris yang sama.

Dua hal patut diperhatikan dalam kodenya. Pertama, `moveDisk` memeriksa `to.peek() < from.peek()` sebelum melakukan *push*: antarmuka stack memberi kita persis informasi yang diperlukan aturan 2 — cakram puncak — dan tidak lebih. Kedua, tidak ada yang pernah mengakses bagian tengah sebuah tiang dengan indeks. Itulah yang membuat ketiga vektor itu stack, bukan larik.

5.3 Stack sebagai Tipe Data Abstrak

Sekarang kita buat gagasan itu tepat.

> **DEFINISI.** *Stack* adalah kumpulan elemen yang penyisipan dan pengambilannya sama-sama terjadi di satu ujung, yang disebut *top* (puncak). Elemen yang diambil selalu yang paling akhir disisipkan di antara elemen-elemen yang masih ada. Disiplin ini disebut *last in, first out* (LIFO): terakhir masuk, pertama keluar.

Contohnya dalam keseharian ada di mana-mana: tumpukan piring di kantin, tumpukan buku di meja, tabung kok bulu tangkis, tumpukan nampan di pujasera. Pada semuanya Anda menambah di atas dan mengambil dari atas.

5.3.1 Operasi-operasinya

Tabel 5.1 mendaftar operasi ADT stack. Nama `push`, `pop`, dan `peek` bersifat universal; sebagian pustaka memakai `top` sebagai ganti `peek`.

Tabel 5.1 Operasi ADT stack, prasyaratnya, dan biayanya pada implementasi larik di Subbab 5.4.

Operasi	Yang dilakukan	Prasyarat	Biaya
`push(x)`	Meletakkan x di puncak stack	Stack tidak penuh (khusus stack berkapasitas tetap)	$O(1)$
`pop()`	Mengambil elemen puncak dan mengembalikannya	Stack tidak kosong	$O(1)$
`peek()` / `top()`	Mengembalikan elemen puncak tanpa mengambilnya	Stack tidak kosong	$O(1)$
`isEmpty()`	Bernilai benar bila stack tidak memuat elemen	—	$O(1)$
`isFull()`	Bernilai benar bila stack berkapasitas tetap tidak punya slot kosong	—	$O(1)$
`size()`	Banyaknya elemen	—	$O(1)$
`clear()`	Mengosongkan seluruh elemen	—	$O(1)$ untuk larik bernilai sederhana

Dua prasyarat itu memiliki nama, karena pelanggarannya adalah *bug* stack yang paling sering terjadi:

- **Stack overflow** (luapan): memanggil `push` pada stack berkapasitas tetap yang sudah penuh.
- **Stack underflow** (kekurangan): memanggil `pop` atau `peek` pada stack kosong.

Sebuah implementasi harus memutuskan apa yang terjadi dalam setiap kasus. Pilihannya adalah menolak dengan sopan (mengembalikan `false`), memberi sinyal galat (melempar eksepsi), atau tumbuh (mengalokasikan ruang tambahan, Subbab 5.5). Membiarkan perilakunya tidak terdefinisi — membaca `data[-1]` — bukanlah pilihan; itu *bug*.

5.3.2 Operasi dalam pseudokode

Pseudokode 5.1 menuliskan operasi untuk stack yang disimpan dalam larik $S[0..N-1]$ dengan bilangan bulat top yang memuat indeks elemen puncak. Konvensinya, $top = -1$ berarti kosong, sehingga banyaknya elemen adalah

$$\text{size} = top + 1. \tag{5.4}$$

Pseudokode 5.1 Operasi stack pada larik $S[0..N-1]$ dengan indeks top; $top = -1$ berarti kosong.

```
STACK-EMPTY(S)
 1.  return S.top = -1

STACK-FULL(S)
 2.  return S.top = N - 1

PUSH(S, x)
 3.  if STACK-FULL(S)
 4.      error "overflow"
 5.  S.top ← S.top + 1              // naikkan top dulu ...
 6.  S[S.top] ← x                   // ... baru tulis

POP(S)
 7.  if STACK-EMPTY(S)
 8.      error "underflow"
 9.  x ← S[S.top]                   // baca dulu ...
10.  S.top ← S.top - 1              // ... baru turunkan top
11.  return x

PEEK(S)
12.  if STACK-EMPTY(S)
13.      error "underflow"
14.  return S[S.top]
```

Dengan kata sederhana: `PUSH` menaikkan penanda puncak satu slot lalu menulis di sana; `POP` membaca slot di bawah penanda lalu menurunkan penandanya. *Urutan* di dalam setiap prosedur itu penting. Jika `PUSH` menulis dahulu lalu menaikkan, *push* pertama akan menulis ke $S[-1]$. Jika `POP` menurunkan dahulu lalu membaca, ia akan mengembalikan elemen *di bawah* puncak. Dalam C++ dan Java kedua langkah itu muat dalam satu ekspresi — `data[++top] = x` dan `return data[top--]` — dan letak `++` serta `--` menyandikan urutannya dengan tepat: *pre-increment* terjadi sebelum operasi tulis, *post-decrement* terjadi setelah pembacaan.

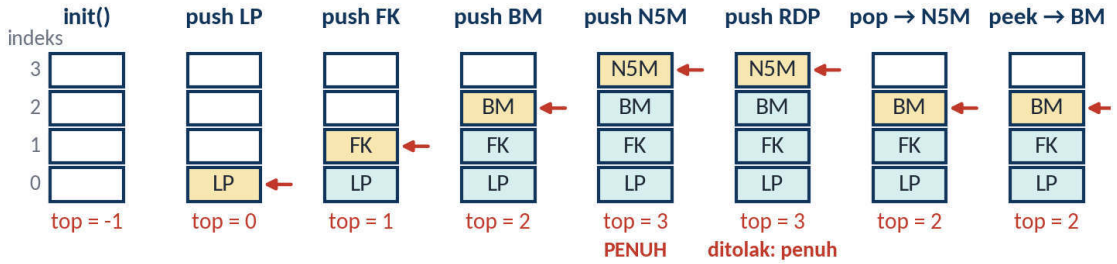
Sebagian buku memakai konvensi lain: top adalah banyaknya elemen, sehingga $top = 0$ berarti kosong dan elemen puncak adalah $S[top - 1]$. Keduanya benar. Yang salah adalah mencampurnya — menginisialisasi top dengan 0 lalu menguji kekosongan dengan $top = -1$. Persamaan (5.4) adalah pengingat satu baris tentang konvensi yang dipakai buku ini.

5.4 Stack Berbasis Larik

Mari kita bangun stack seperti yang dijabarkan ADT-nya: sebuah larik berkapasitas tetap ditambah satu bilangan bulat. Troli pengembalian Pustaka Ceria memuat empat buku, jadi stack kita berkapasitas 4.

5.4.1 Mengamati larik

Sebelum membaca kode apa pun, amatilah lariknya. **Gambar 5.3** mengikuti delapan operasi pada stack berkapasitas 4, dengan judul buku disingkat: LP untuk *Laskar Pelangi*, FK untuk *Filosofi Kopi*, BM untuk *Bumi Manusia*, N5M untuk *Negeri 5 Menara*, dan RDP untuk *Ronggeng Dukuh Paruk*. Indeks 0 berada di dasar setiap kolom.



Gambar 5.3 Stack larik berkapasitas 4 melalui delapan operasi. Sel kuning adalah elemen puncak dan panah merah adalah indeks `top`; setelah *push* keempat stack penuh dan *push* kelima ditolak.

Bacalah kolom-kolomnya dari kiri ke kanan:

1. `init()` menetapkan $top = -1$. Larik punya empat slot, tetapi tidak ada yang terpakai. (Slot-slot itu mungkin berisi sisa data lama; tidak masalah, karena tidak ada data sampah pada atau di bawah *top*, dan tidak ada yang pernah dibaca di atasnya.)
2. Masing-masing dari empat *push* berikutnya menaikkan *top* satu slot lalu menulis di sana: *top* menjadi 0,1,2,3. Setelah *push* keempat, $top = 3 = N - 1$, sehingga `isFull()` bernilai benar.
3. *Push* kelima, untuk RDP, ditolak. Tidak ada yang berubah — baik larik maupun *top*.
4. `pop` mengembalikan elemen di bawah penanda, N5M, dan memindahkan penanda ke 2. Perhatikan apa yang terjadi pada larik: *tidak ada*. N5M secara fisik masih berada di slot 3 sampai ada yang menimpanya. Ia keluar dari stack semata-mata karena *top* tidak lagi mencakupnya.
5. `peek` mengembalikan BM dan tidak mengubah apa pun.

Butir 4 layak dilihat dua kali, karena inilah hal yang paling mengejutkan tentang stack larik bagi pemula: **mengambil elemen tidak menghapusnya**. Isi stack didefinisikan sebagai "slot 0 sampai *top*", sehingga memindahkan *top* sudah cukup. Itulah pula sebabnya `clear()` hanya berupa satu penugasan, $top \leftarrow -1$, pada versi C++ di bawah.

5.4.2 Stack dalam C++

Kode 5.2 mengimplementasikan stack sebagai *class template*, sehingga kode yang sama hari ini menyimpan judul dan esok dapat menyimpan bilangan bulat, karakter, atau rekaman `Book`. Anggota privatnya persis seperti dalam **Pseudokode 5.1**: larik `data_`, kapasitasnya, dan `top_`.

Kode 5.2 `02_array_stack.cpp`: *template* stack berbasis larik dalam C++, dipakai sebagai troli pengembalian buku Pustaka Ceria.

```cpp
// An array-based stack: Pustaka Ceria's book return cart.
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;

template <typename T>
class ArrayStack {
public:
    explicit ArrayStack(int capacity) : data_(new T[capacity]), capacity_(capacity), top_(-
1) {}
    ~ArrayStack() { delete[] data_; }
    ArrayStack(const ArrayStack&) = delete;             // no accidental copies of the
array
    ArrayStack& operator=(const ArrayStack&) = delete;

    bool isEmpty() const { return top_ == -1; }
    bool isFull()  const { return top_ == capacity_ - 1; }
    int  size()    const { return top_ + 1; }
    int  topIndex() const { return top_; }

    bool push(const T& value) {                         // false = overflow, nothing
changed
        if (isFull()) return false;
        data_[++top_] = value;                          // first move top up, then write
        return true;
    }
    T pop() {
        if (isEmpty()) throw underflow_error("pop from an empty stack");
        return data_[top_--];                           // first read, then move top down
    }
    const T& peek() const {
        if (isEmpty()) throw underflow_error("peek at an empty stack");
        return data_[top_];
    }
    void clear() { top_ = -1; }                         // the old values are simply
forgotten
    const T& at(int i) const { return data_[i]; }       // for printing only

private:
    T*  data_;
    int capacity_;
    int top_;                                           // index of the top element, -1 if
empty
};
```

Kode 5.2 (Lanjutan) *02_array_stack.cpp: template stack berbasis larik dalam C++, dipakai sebagai troli pengembalian buku Pustaka Ceria.*

```
void printCart(const ArrayStack<string>& s) {
    cout << "cart, top to bottom:";
    for (int i = s.topIndex(); i >= 0; --i) cout << (i == s.topIndex() ? " " : " | ") <<
s.at(i);
    cout << "\n";
}

int main() {
    cout << "=== Pustaka Ceria: book return cart (array stack, capacity 4) ===\n";
    ArrayStack<string> cart(4);
    const string returned[] = {"Laskar Pelangi", "Filosofi Kopi", "Bumi Manusia",
                               "Negeri 5 Menara", "Ronggeng Dukuh Paruk"};
    for (const string& title : returned) {
        if (cart.push(title))
            cout << "push " << title << " -> ok, size " << cart.size()
                 << ", top index " << cart.topIndex() << "\n";
        else
            cout << "push " << title << " -> REJECTED, the cart is full\n";
    }
    printCart(cart);
    cout << "peek -> " << cart.peek() << " (size is still " << cart.size() << ")\n";
    while (!cart.isEmpty()) {
        string t = cart.pop();
        cout << "pop  -> " << t << ", size " << cart.size()
             << ", top index " << cart.topIndex() << "\n";
    }
    try {
        cart.pop();
    } catch (const underflow_error& e) {
        cout << "pop on the empty cart -> error: " << e.what() << "\n";
    }
    cart.push("Cantik Itu Luka");
    cart.push("Gadis Kretek");
    cout << "refilled, size " << cart.size() << "; clear() -> ";
    cart.clear();
    cout << "size " << cart.size() << ", isEmpty " << (cart.isEmpty() ? "true" : "false")
<< "\n";
    return 0;
}
```

```
=== Pustaka Ceria: book return cart (array stack, capacity 4) ===
push Laskar Pelangi -> ok, size 1, top index 0
push Filosofi Kopi -> ok, size 2, top index 1
push Bumi Manusia -> ok, size 3, top index 2
push Negeri 5 Menara -> ok, size 4, top index 3
push Ronggeng Dukuh Paruk -> REJECTED, the cart is full
cart, top to bottom: Negeri 5 Menara | Bumi Manusia | Filosofi Kopi | Laskar Pelangi
peek -> Negeri 5 Menara (size is still 4)
pop  -> Negeri 5 Menara, size 3, top index 2
pop  -> Bumi Manusia, size 2, top index 1
pop  -> Filosofi Kopi, size 1, top index 0
pop  -> Laskar Pelangi, size 0, top index -1
pop on the empty cart -> error: pop from an empty stack
refilled, size 2; clear() -> size 0, isEmpty true
```

Cocokkan keluaran ini dengan **Gambar 5.3**. Empat *push* yang diterima memindahkan indeks puncak ke 0, 1, 2, 3; *push* kelima ditolak dengan pesan “the cart is full”; *pop* mengembalikan buku dalam urutan yang persis terbalik dari urutan *push*; dan *pop* pada troli kosong dilaporkan sebagai galat, bukan diam-diam membaca data sampah.

Beberapa keputusan rancangan dalam kelas ini patut dijelaskan, karena masing-masing mencegah *bug* yang nyata.

- `push` **mengembalikan** `bool`**;** `pop` **dan** `peek` **melempar eksepsi.** Troli penuh adalah kejadian biasa di meja yang sibuk — pemanggil semestinya memeriksa dan bereaksi — sehingga `push` melaporkannya dengan nilai kembalian. Sebaliknya, *pop* pada stack kosong berarti kode pemanggil mengandung kesalahan logika, sehingga ia melempar `std::underflow_error`. Ini meniru pilihan pustaka standar sendiri (Subbab 5.16).
- **Konstruktor salinan dan penugasan salinan dihapus.** Kelas ini memiliki larik mentah yang dialokasikan dengan `new[]`. Jika C++ dibiarkan menyalin objek anggota demi anggota, dua objek akan memegang pointer yang sama, dan kedua destruktornya akan melakukan `delete[]` terhadapnya — *double free*. Menghapus operasi salinan mengubah bencana saat eksekusi itu menjadi galat saat kompilasi. Bab 6 menjelaskan kepemilikan dan pointer dengan semestinya.
- `peek` **mengembalikan** `const T&`**.** Pemanggil dapat melihat elemen puncak tanpa menyalinnya dan tanpa dapat mengubahnya melalui referensi itu.
- `at(i)` **ada hanya untuk pencetakan.** Stack sejati tidak akan menyediakannya. Kita mempertahankannya agar `printCart` dapat menampilkan seluruh troli, dan kita menamainya sedemikian rupa sehingga tidak ada yang mengiranya operasi stack.

5.4.3 Stack yang sama dalam Java dan Python

Buku ini ditulis untuk C++, tetapi setiap program di folder Code juga tersedia dalam Java dan Python dengan keluaran yang identik. Untuk stack itu sendiri, ketiganya layak dilihat berdampingan, karena perbedaannya mengajarkan sesuatu. **Kode 5.3** adalah versi Java.

Kode 5.3 `ArrayStackDemo.java`: stack larik yang sama sebagai kelas generik Java.

```
// An array-based stack: Pustaka Ceria's book return cart.
public class ArrayStackDemo {
    static class ArrayStack<T> {
        private final T[] data;
        private int top = -1;                       // index of the top element, -1 if
empty

        @SuppressWarnings("unchecked")
        ArrayStack(int capacity) { data = (T[]) new Object[capacity]; }

        boolean isEmpty() { return top == -1; }
        boolean isFull()  { return top == data.length - 1; }
        int size()        { return top + 1; }
        int topIndex()    { return top; }

        boolean push(T value) {                     // false = overflow, nothing changed
            if (isFull()) return false;
            data[++top] = value;                    // first move top up, then write
            return true;
        }
        T pop() {
            if (isEmpty()) throw new IllegalStateException("pop from an empty stack");
            T value = data[top];
            data[top--] = null;                     // let the garbage collector reclaim it
            return value;
        }
        T peek() {
            if (isEmpty()) throw new IllegalStateException("peek at an empty stack");
            return data[top];
        }
        void clear() { while (top >= 0) data[top--] = null; }
        T at(int i)  { return data[i]; }            // for printing only
    }

    static void printCart(ArrayStack<String> s) {
        StringBuilder line = new StringBuilder("cart, top to bottom:");
        for (int i = s.topIndex(); i >= 0; --i) line.append(i == s.topIndex() ? " " : " |
").append(s.at(i));
        System.out.println(line);
    }
```

Kode 5.3 (Lanjutan) *ArrayStackDemo.java: stack larik yang sama sebagai kelas generik Java.*

```
    public static void main(String[] args) {
        System.out.println("=== Pustaka Ceria: book return cart (array stack, capacity 4)
===");
        ArrayStack<String> cart = new ArrayStack<>(4);
        String[] returned = {"Laskar Pelangi", "Filosofi Kopi", "Bumi Manusia",
                             "Negeri 5 Menara", "Ronggeng Dukuh Paruk"};
        for (String title : returned) {
            if (cart.push(title))
                System.out.println("push " + title + " -> ok, size " + cart.size()
                                   + ", top index " + cart.topIndex());
            else
                System.out.println("push " + title + " -> REJECTED, the cart is full");
        }
        printCart(cart);
        System.out.println("peek -> " + cart.peek() + " (size is still " + cart.size() +
")");
        while (!cart.isEmpty()) {
            String t = cart.pop();
            System.out.println("pop  -> " + t + ", size " + cart.size() + ", top index " +
cart.topIndex());
        }
        try {
            cart.pop();
        } catch (IllegalStateException e) {
            System.out.println("pop on the empty cart -> error: " + e.getMessage());
        }
        cart.push("Cantik Itu Luka");
        cart.push("Gadis Kretek");
        System.out.print("refilled, size " + cart.size() + "; clear() -> ");
        cart.clear();
        System.out.println("size " + cart.size() + ", isEmpty " + cart.isEmpty());
    }
}
```

Java tidak dapat membuat larik bertipe generik secara langsung — `new T[capacity]` tidak sah, karena tipe generik dihapus (*erased*) saat eksekusi — sehingga konstruktornya membuat `Object[]` lalu melakukan *cast*, dan anotasi `@SuppressWarnings` mengakui bahwa *cast* itu aman karena larik tersebut tidak pernah keluar dari kelas. `pop` pada Java juga menulis `null` ke slot yang dikosongkan. Dalam C++ nilai lama di slot itu tidak berbahaya; dalam Java nilai itu adalah *referensi* yang membuat objeknya tetap hidup, sehingga *garbage collector* tidak akan pernah dapat mengambil kembali buku itu walaupun sudah keluar dari stack. Mengisi slot dengan `null` adalah cara Java mengatakan "slot ini tidak lagi berisi apa-apa".

Kode 5.4 adalah versi Python. `list` Python dapat berperan sebagai stack tanpa kelas sama sekali (Subbab 5.16 melakukan persis itu), tetapi di sini kita sengaja mengalokasikan blok tetap berisi `capacity` slot dan memegang `_top` sendiri, agar bentuk programnya sama dengan versi C++.

Kode 5.4 `02_array_stack.py`: stack larik yang sama dalam Python, dengan *list* berukuran tetap dan indeks puncak eksplisit.

```
# An array-based stack: Pustaka Ceria's book return cart.


class ArrayStack:
    def __init__(self, capacity):
        self._data = [None] * capacity   # a fixed block of slots, like a C++ array
        self._top = -1                   # index of the top element, -1 if empty

    def is_empty(self):
        return self._top == -1

    def is_full(self):
        return self._top == len(self._data) - 1

    def size(self):
        return self._top + 1

    def top_index(self):
        return self._top

    def push(self, value):               # False = overflow, nothing changed
        if self.is_full():
            return False
        self._top += 1                   # first move top up, then write
        self._data[self._top] = value
        return True

    def pop(self):
        if self.is_empty():
            raise IndexError("pop from an empty stack")
        value = self._data[self._top]
        self._data[self._top] = None     # drop the reference
        self._top -= 1
        return value

    def peek(self):
        if self.is_empty():
            raise IndexError("peek at an empty stack")
        return self._data[self._top]

    def clear(self):
        while self._top >= 0:
            self._data[self._top] = None
            self._top -= 1
```

Kode 5.4 (Lanjutan) *02_array_stack.py: stack larik yang sama dalam Python, dengan list berukuran tetap dan indeks puncak eksplisit.*

```
    def at(self, i):                    # for printing only
        return self._data[i]


def print_cart(s):
    parts = [s.at(i) for i in range(s.top_index(), -1, -1)]
    print("cart, top to bottom: " + " | ".join(parts))


def main():
    print("=== Pustaka Ceria: book return cart (array stack, capacity 4) ===")
    cart = ArrayStack(4)
    returned = ["Laskar Pelangi", "Filosofi Kopi", "Bumi Manusia",
                "Negeri 5 Menara", "Ronggeng Dukuh Paruk"]
    for title in returned:
        if cart.push(title):
            print(f"push {title} -> ok, size {cart.size()}, top index {cart.top_index()}")
        else:
            print(f"push {title} -> REJECTED, the cart is full")
    print_cart(cart)
    print(f"peek -> {cart.peek()} (size is still {cart.size()})")
    while not cart.is_empty():
        t = cart.pop()
        print(f"pop  -> {t}, size {cart.size()}, top index {cart.top_index()}")
    try:
        cart.pop()
    except IndexError as e:
        print(f"pop on the empty cart -> error: {e}")
    cart.push("Cantik Itu Luka")
    cart.push("Gadis Kretek")
    print(f"refilled, size {cart.size()}; clear() -> ", end="")
    cart.clear()
    print(f"size {cart.size()}, isEmpty {'true' if cart.is_empty() else 'false'}")


if __name__ == "__main__":
    main()
```

Kedua program mencetak keluaran yang persis sama dengan yang ditampilkan setelah **Kode 5.2**. Python melempar `IndexError` untuk *underflow*, yaitu eksepsi yang juga dilempar `list.pop()` bawaan Python pada *list* kosong, sehingga pemrogram Python bertemu eksepsi yang sudah dikenalnya.

5.4.4 Setiap operasi bernilai $O(1)$

Lihat kembali setiap metode dan hitung pekerjaannya. `push` melakukan satu perbandingan, satu penambahan, dan satu operasi tulis. `pop` melakukan satu perbandingan, satu pembacaan, dan satu pengurangan. `peek`, `isEmpty`, `isFull`, dan `size` melakukan satu atau dua langkah primitif. Tidak satu pun mengandung perulangan, dan tidak satu pun bergantung pada banyaknya elemen yang tersimpan. Karena itu masing-masing $O(1)$ — waktu konstan — baik stack itu memuat empat buku maupun empat juta. `clear` bernilai $O(1)$ dalam C++ (satu penugasan) dan $O(n)$ dalam versi Java dan Python, yang menelusuri larik sambil menulis `null`/`None` agar *garbage collector* dapat mengambil kembali objek-objeknya; biaya tambahan itu membeli perilaku memori yang benar.

Harga kecepatan ini adalah batas kapasitas. Stack berkapasitas tetap dapat meluap. Subbab berikutnya menghapus batas itu dengan dua cara yang berbeda.

5.5 Stack yang Tumbuh dan Stack Bertaut

Kapasitas tetap kadang memang tepat — troli pengembalian memang hanya memuat empat buku, dan sebuah pengendali tertanam (*embedded controller*) mungkin memang hanya punya ruang untuk 256 entri. Lebih sering kita tidak tahu sebelumnya berapa banyak elemen yang akan datang. Ada dua obat baku. Yang pertama mempertahankan larik tetapi menggantinya dengan larik yang lebih besar ketika penuh. Yang kedua meninggalkan larik dan menautkan elemen satu sama lain dengan pointer.

5.5.1 Menggandakan larik

Ketika `push` mendapati larik penuh, alokasikan larik baru yang **dua kali** lebih besar, salin setiap elemen ke sana, bebaskan larik lama, lalu lanjutkan. **Kode 5.5** melakukan persis hal itu, dimulai dari kapasitas 1 agar kita dapat menyaksikan beberapa kali penggandaan.

Kode 5.5 `03_growing_stack.cpp`: stack yang menggandakan lariknya ketika penuh, sambil menghitung setiap penyalinan elemen.

```
// A stack that never becomes full: when the array is full, allocate one twice as big.
#include <iostream>
using namespace std;

class GrowingStack {
public:
    GrowingStack() : data_(new int[1]), capacity_(1), top_(-1), copies_(0) {}
    ~GrowingStack() { delete[] data_; }
    GrowingStack(const GrowingStack&) = delete;
    GrowingStack& operator=(const GrowingStack&) = delete;

    void push(int value) {
        if (top_ == capacity_ - 1) grow();              // full: double first
        data_[++top_] = value;
    }
    int  pop()      { return data_[top_--]; }            // caller checks isEmpty() first
    bool isEmpty()  const { return top_ == -1; }
    int  size()     const { return top_ + 1; }
    int  capacity() const { return capacity_; }
    long copies()   const { return copies_; }

private:
    void grow() {
        int* bigger = new int[2 * capacity_];
        for (int i = 0; i <= top_; ++i) bigger[i] = data_[i];   // copy every element
        copies_ += top_ + 1;
        cout << "  grow: capacity " << capacity_ << " -> " << 2 * capacity_
             << ", copied " << top_ + 1 << " element(s)\n";
        delete[] data_;
        data_ = bigger;
        capacity_ *= 2;
    }
    int* data_;
    int  capacity_;
    int  top_;
    long copies_;                                       // total element copies so far
};
```

Kode 5.5 (Lanjutan) *03_growing_stack.cpp: stack yang menggandakan lariknya ketika penuh, sambil menghitung setiap penyalinan elemen.*

```
int main() {
    GrowingStack s;
    const int n = 17;
    cout << "pushing 1.." << n << " onto a stack that starts with capacity 1\n";
    for (int v = 1; v <= n; ++v) s.push(v);
    cout << "size " << s.size() << ", capacity " << s.capacity()
         << ", total copies " << s.copies() << " (less than 2n = " << 2 * n << ")\n";
    cout << "popping everything:";
    while (!s.isEmpty()) cout << " " << s.pop();
    cout << "\n";
    return 0;
}
```

```
pushing 1..17 onto a stack that starts with capacity 1
  grow: capacity 1 -> 2, copied 1 element(s)
  grow: capacity 2 -> 4, copied 2 element(s)
  grow: capacity 4 -> 8, copied 4 element(s)
  grow: capacity 8 -> 16, copied 8 element(s)
  grow: capacity 16 -> 32, copied 16 element(s)
size 17, capacity 32, total copies 31 (less than 2n = 34)
popping everything: 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

Satu *push* kini bisa mahal: *push* yang mendapati 16 elemen dalam larik 16 slot harus menyalin keenam belasnya sebelum dapat menulis elemen ketujuh belas. Jadi, apakah `push` masih $O(1)$? Tidak untuk setiap *push* secara individual — tetapi secara rata-rata atas barisan *push* apa pun, ya, dan keluarannya memperlihatkan alasannya.

Tujuh belas *push* menyebabkan penyalinan $1 + 2 + 4 + 8 + 16 = 31$ elemen. Secara umum, bila n elemen di-*push* mulai dari kapasitas 1, larik berlipat dua ketika ukurannya mencapai $1,2,4,\ldots,2^k$, dengan 2^k perpangkatan dua terbesar di bawah n. Banyaknya penyalinan adalah deret geometri:

$$1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 < 2n. \tag{5.5}$$

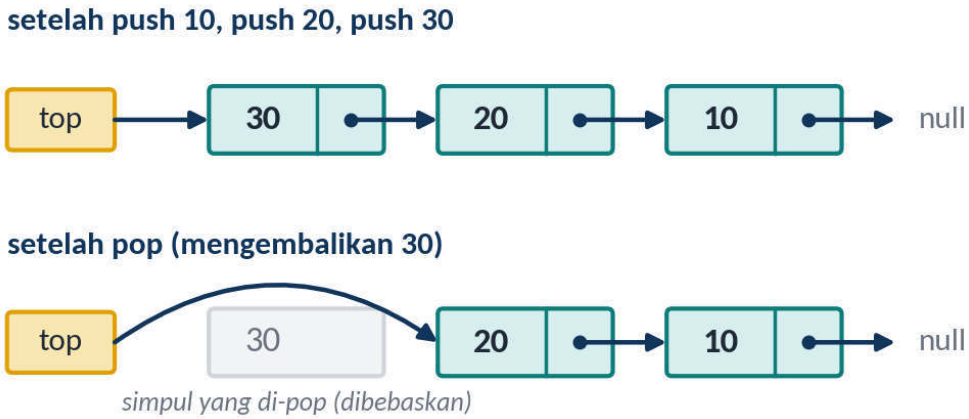
Ketaksamaan terakhir berlaku karena $2^k < n$ berakibat $2^{k+1} < 2n$. Jadi n *push* berbiaya paling banyak n operasi tulis ditambah kurang dari $2n$ penyalinan — total kurang dari $3n$ operasi elemen, atau rata-rata kurang dari 3 per *push*. Persamaan (5.5) adalah jantung dari apa yang disebut **analisis amortisasi** (*amortized analysis*): *push* mahal yang sesekali terjadi dibayar oleh banyak *push* murah sebelumnya, sehingga biaya *teramortisasi* per *push* adalah $O(1)$ walaupun biaya *kasus terburuk* satu *push* adalah $O(n)$. Faktornya penting: menumbuhkan larik dengan jumlah tetap (misalnya 10 slot setiap kali) alih-alih menggandakannya membuat total penyalinan sekitar $n^2/20$, yaitu rata-rata $O(n)$ per *push*. Sedgewick dan Wayne menyebut larik yang berlipat dua ini *resizing array* dan membuktikan batas yang sama [4].

Beginilah persisnya `std::vector`, `ArrayList` dan `ArrayDeque` di Java, serta `list` di Python tumbuh — dengan faktor pertumbuhan yang rinciannya berbeda, tetapi selalu dengan *faktor* konstan, bukan jumlah konstan.

5.5.2 Sekilas: stack bertaut

Obat kedua menyimpan setiap elemen dalam blok memori kecilnya sendiri, sebuah *simpul* (*node*), yang juga mencatat letak simpul berikutnya. Puncak stack adalah pointer ke simpul pertama; setiap simpul menunjuk simpul di bawahnya; simpul terbawah tidak menunjuk apa pun (`nullptr`). *Push* membuat simpul baru yang menunjuk puncak lama lalu menjadi puncak. *Pop*

memindahkan pointer puncak ke simpul berikutnya dan membebaskan simpul lama. **Gambar 5.4** memperlihatkan keduanya.



Gambar 5.4 Stack bertaut setelah *push* 10, 20, dan 30 (atas), dan setelah satu *pop* (bawah): pointer puncak melompati simpul pertama yang lama, yang kemudian dibebaskan.

Pointer adalah pokok bahasan Bab 6 dan *linked list* adalah pokok bahasan Bab 7, sehingga **Kode 5.6** hanyalah pratinjau: bacalah untuk menangkap gagasannya, lalu kembalilah setelah Bab 7.

Kode 5.6 `04_linked_stack.cpp`: pratinjau stack yang dibangun dari simpul-simpul bertaut.

```
// A preview of Chapters 6 and 7: a stack made of linked nodes instead of an array.
#include <iostream>
#include <stdexcept>
using namespace std;

struct Node {
    int   value;
    Node* next;                                     // the node underneath this one
};

class LinkedStack {
public:
    LinkedStack() : top_(nullptr), size_(0) {}
    ~LinkedStack() { while (!isEmpty()) pop(); }        // free every node
    LinkedStack(const LinkedStack&) = delete;
    LinkedStack& operator=(const LinkedStack&) = delete;

    void push(int v) {
        top_ = new Node{v, top_};                       // new node points at the old top
        ++size_;
    }
    int pop() {
        if (isEmpty()) throw underflow_error("pop from an empty stack");
        Node* old = top_;
        int v = old->value;
        top_ = old->next;                               // unlink first ...
        delete old;                                     // ... then free
        --size_;
        return v;
    }
    int  peek()    const { if (isEmpty()) throw underflow_error("peek at an empty stack");
return top_->value; }
    bool isEmpty() const { return top_ == nullptr; }
    int  size()    const { return size_; }
    void print()   const {
        cout << "top";
        for (Node* p = top_; p != nullptr; p = p->next) cout << " -> " << p->value;
        cout << " -> null   (size " << size_ << ")\n";
    }

private:
    Node* top_;
    int   size_;
};
```

Kode 5.6 (Lanjutan) *04_linked_stack.cpp: pratinjau stack yang dibangun dari simpul-simpul bertaut.*

```
int main() {
    LinkedStack s;
    s.print();
    for (int v : {10, 20, 30}) { s.push(v); cout << "push " << v << ": "; s.print(); }
    cout << "peek -> " << s.peek() << "\n";
    int v = s.pop();
    cout << "pop  -> " << v << ": "; s.print();
    s.push(40);
    cout << "push 40: "; s.print();
    cout << "(" << s.size() << " nodes remain; they are released when s goes out of
scope)\n";
    return 0;
}
```

```
top -> null   (size 0)
push 10: top -> 10 -> null   (size 1)
push 20: top -> 20 -> 10 -> null   (size 2)
push 30: top -> 30 -> 20 -> 10 -> null   (size 3)
peek -> 30
pop  -> 30: top -> 20 -> 10 -> null   (size 2)
push 40: top -> 40 -> 20 -> 10 -> null   (size 3)
(3 nodes remain; they are released when s goes out of scope)
```

Tiga baris memikul seluruh gagasannya. `top_ = new Node{v, top_};` membangun simpul yang `next`-nya adalah puncak lama lalu menjadikannya puncak baru, dalam satu pernyataan. Dalam `pop`, `top_ = old->next;` melepas tautan simpul *sebelum* `delete old;` membebaskannya — membalik kedua baris itu berarti membaca `old->next` dari memori yang sudah dibebaskan. Dan destruktornya melakukan *pop* sampai stack kosong, sehingga tidak ada simpul yang bocor. Kami mengompilasi program ini dengan `-fsanitize=address,undefined` dan menjalankannya di bawah Valgrind: nol galat, dan "All heap blocks were freed — no leaks are possible".

Stack bertaut tidak pernah meluap (sampai mesin kehabisan memori), dan setiap operasinya $O(1)$ pada kasus terburuk, bukan sekadar rata-rata. Biayanya adalah satu alokasi per *push*, satu pointer tambahan per elemen, dan simpul-simpul yang tersebar di memori, yang dibaca prosesor modern lebih lambat daripada larik yang bersebelahan. Dalam praktik, larik berlipat dua biasanya menang; pustaka standar ketiga bahasa memakai larik untuk stack-nya.

5.6 Aplikasi: Mencocokkan Tanda Kurung

Setiap kompilator, setiap penyunting kode yang mewarnai pasangan tanda kurung, setiap pengurai JSON dan HTML harus menjawab pertanyaan yang sama: apakah tanda kurung dalam teks ini *bersarang* dengan benar? Teks `{[()()]}` benar. Teks `([)]` tidak: `)` datang ketika tanda kurung buka terdalam adalah `[`. Teks `((a+b)*c` juga tidak: satu `(` tidak pernah ditutup.

Menghitung saja tidak cukup. `([)]` memiliki dua pembuka dan dua penutup untuk setiap jenis, tetapi tetap salah. Yang penting adalah bahwa **setiap tanda kurung tutup harus cocok dengan tanda kurung buka paling akhir yang belum dipasangkan** — dan "paling akhir yang belum dipasangkan" adalah definisi puncak sebuah stack.

5.6.1 Algoritmanya

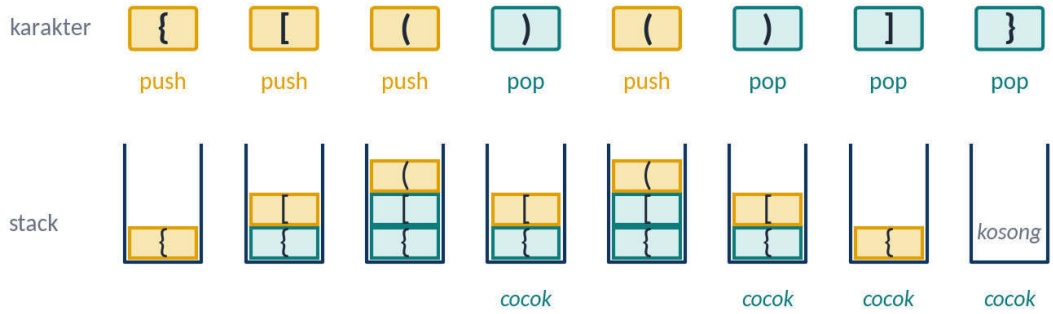
Pseudokode 5.2 memindai teks sekali dari kiri ke kanan.

Pseudokode 5.2 Memeriksa bahwa tanda kurung bersarang dengan benar menggunakan satu stack dan satu pemindaian kiri ke kanan.

```
BALANCED(text)
 1.  S ← stack kosong
 2.  for i ← 0 to length(text) - 1
 3.      c ← text[i]
 4.      if c salah satu dari ( [ {
 5.          PUSH(S, c)
 6.      else if c salah satu dari ) ] }
 7.          if STACK-EMPTY(S)
 8.              return "error at i: c has no opening partner"
 9.          if PEEK(S) bukan pasangan c
10.              return "error at i: c cannot close PEEK(S)"
11.          POP(S)
12.      // karakter lain diabaikan
13.  if not STACK-EMPTY(S)
14.      return "error at end: brackets never closed"
15.  return "balanced"
```

Dengan kata sederhana: lakukan *push* untuk setiap pembuka; pada setiap penutup, puncak stack harus pasangannya, dan jika demikian lakukan *pop*; di akhir, stack harus kosong. Ada tepat tiga cara gagal, dan masing-masing punya barisnya sendiri: penutup tanpa pembuka yang terbuka (baris 8), penutup yang tidak cocok dengan pembuka terdalam (baris 10), dan pembuka yang tidak pernah ditutup (baris 14).

Gambar 5.5 menelusuri kasus seimbang `{[()()]}` karakter demi karakter.



Gambar 5.5 Memeriksa `{[()()]}`: setiap pembuka di-*push*, setiap penutup mengambil pasangannya dari puncak, dan stack kosong di akhir, sehingga teksnya seimbang.

5.6.2 Programnya

Kode 5.7 mengimplementasikan **Pseudokode 5.2** dengan `std::stack<char>` dari pustaka standar — kita sudah membangun stack sendiri di Subbab 5.4, dan mulai sekarang kita boleh memakai milik pustaka. Program ini mencetak penelusuran untuk uji pertama dan vonis untuk enam teks.

Kode 5.7 `05_brackets.cpp`: pencocokan tanda kurung dengan `std::stack`, melaporkan posisi dan jenis galat pertama.

```
// Bracket matching with a stack: ( ) [ ] { } must nest properly.
#include <iomanip>
#include <iostream>
#include <stack>
#include <string>
using namespace std;

bool isOpen(char c)  { return c == '(' || c == '[' || c == '{'; }
bool isClose(char c) { return c == ')' || c == ']' || c == '}'; }
char partner(char close) { return close == ')' ? '(' : close == ']' ? '[' : '{'; }

string contents(stack<char> s) {                        // a copy, printed bottom -> top
    string r;
    while (!s.empty()) { r = string(1, s.top()) + (r.empty() ? "" : " ") + r; s.pop(); }
    return r.empty() ? "(empty)" : r;
}

// Returns "balanced" or a message naming the first problem and its position (0-based).
string check(const string& text, bool trace) {
    stack<char> open;
    for (size_t i = 0; i < text.size(); ++i) {
        char c = text[i];
        if (isOpen(c)) {
            open.push(c);
            if (trace) cout << "  " << c << "  push           stack: " << contents(open) <<
"\n";
        } else if (isClose(c)) {
            if (open.empty())
                return "error at " + to_string(i) + ": '" + c + "' has no opening partner";
            if (open.top() != partner(c))
                return "error at " + to_string(i) + ": '" + c + "' cannot close '" +
open.top() + "'";
            open.pop();
            if (trace) cout << "  " << c << "  pop, matches   stack: " << contents(open) <<
"\n";
        }                                                // every other character is ignored
    }
    if (!open.empty())
        return "error at end: " + to_string(open.size()) + " bracket(s) never closed, top
is '"
               + open.top() + "'";
    return "balanced";
}
```

Kode 5.7 (Lanjutan) *05_brackets.cpp: pencocokan tanda kurung dengan std::stack, melaporkan posisi dan jenis galat pertama.*

```
int main() {
    cout << "trace of {[()()]}:\n";
    string verdict = check("{[()()]}", true);
    cout << "  result: " << verdict << "\n\n";
    const string tests[] = {"{[()()]}", "([)]", "((a+b)*c", "a+b)*(c",
                            "if (a[i] == b[j]) { x(); }", ""};
    for (const string& t : tests)
        cout << left << setw(30) << ("\"" + t + "\"") << check(t, false) << "\n";
    return 0;
}
```

```
trace of {[()()]}:
  {  push           stack: {
  [  push           stack: { [
  (  push           stack: { [ (
  )  pop, matches   stack: { [
  (  push           stack: { [ (
  )  pop, matches   stack: { [
  ]  pop, matches   stack: {
  }  pop, matches   stack: (empty)
  result: balanced

"{[()()]}"                    balanced
"([)]"                        error at 2: ')' cannot close '['
"((a+b)*c"                    error at end: 1 bracket(s) never closed, top is '('
"a+b)*(c"                     error at 3: ')' has no opening partner
"if (a[i] == b[j]) { x(); }"  balanced
""                            balanced
```

Masing-masing dari ketiga cara gagal muncul sekali. Pada `([)]`, `)` di posisi 2 mendapati `[` di puncak: tidak cocok. Pada `((a+b)*c`, pemindaian berakhir dengan satu `(` masih di stack. Pada `a+b)*(c`, `)` di posisi 3 mendapati stack kosong. Baris `if (a[i] == b[j]) { x(); }` memperlihatkan bahwa karakter biasa dilewati begitu saja, dan string kosong seimbang, karena satu-satunya syarat adalah stack kosong di akhir.

Algoritma ini membaca setiap karakter sekali dan melakukan pekerjaan stack $O(1)$ per karakter, sehingga berjalan dalam waktu $O(n)$ untuk teks sepanjang n. Pada kasus terburuk — teks berisi n pembuka — stack memuat n karakter, sehingga ia juga memerlukan ruang tambahan $O(n)$.

5.7 Aplikasi: Undo dan Redo

Tekan Ctrl+Z di penyunting mana pun dan perubahan terakhir lenyap. Tekan lagi dan perubahan sebelumnya lenyap. Perubahan yang *terakhir* dibuat adalah yang *pertama* dibatalkan: *undo* bersifat LIFO. Tekan Ctrl+Y (atau Ctrl+Shift+Z) dan perubahan yang terakhir dibatalkan kembali — *redo* juga LIFO. Karena itu sebuah penyunting menyimpan dua stack.

- Sebelum setiap suntingan, **lakukan *push* keadaan saat ini ke stack undo**, lakukan suntingannya, lalu **kosongkan stack redo** — begitu Anda mengetik sesuatu yang baru, perubahan yang tadi dibatalkan tidak lagi punya tempat untuk kembali.
- Untuk **undo**, *push* keadaan saat ini ke stack redo dan *pop* stack undo menjadi keadaan saat ini.
- Untuk **redo**, *push* keadaan saat ini ke stack undo dan *pop* stack redo menjadi keadaan saat ini.

Kode 5.8 menerapkan hal ini pada catatan satu baris yang diketik pustakawan ke katalog Pustaka Ceria. Demi kejelasan, program ini menyimpan salinan utuh teksnya (teknik yang disebut *memento*); penyunting sungguhan menyimpan operasi suntingannya, yang lebih hemat untuk dokumen panjang tetapi mengikuti disiplin dua stack yang sama.

Kode 5.8 `06_undo_redo.cpp`: *undo* dan *redo* dengan dua stack berisi keadaan yang disimpan.

```
// Undo and redo with two stacks, for a one-line note in the Pustaka Ceria catalog.
#include <iomanip>
#include <iostream>
#include <stack>
#include <string>
using namespace std;

class Editor {
public:
    void type(const string& s) {
        undo_.push(text_);                              // remember the state before the
change
        text_ += s;
        while (!redo_.empty()) redo_.pop();             // a new edit makes old redos
meaningless
        report("type \"" + s + "\"");
    }
    void erase(int k) {                                 // delete the last k characters
        undo_.push(text_);
        text_.erase(text_.size() - static_cast<size_t>(k));
        while (!redo_.empty()) redo_.pop();
        report("erase " + to_string(k));
    }
    void undo() {
        if (undo_.empty()) { report("undo (nothing to undo)"); return; }
        redo_.push(text_);
        text_ = undo_.top(); undo_.pop();
        report("undo");
    }
    void redo() {
        if (redo_.empty()) { report("redo (nothing to redo)"); return; }
        undo_.push(text_);
        text_ = redo_.top(); redo_.pop();
        report("redo");
    }

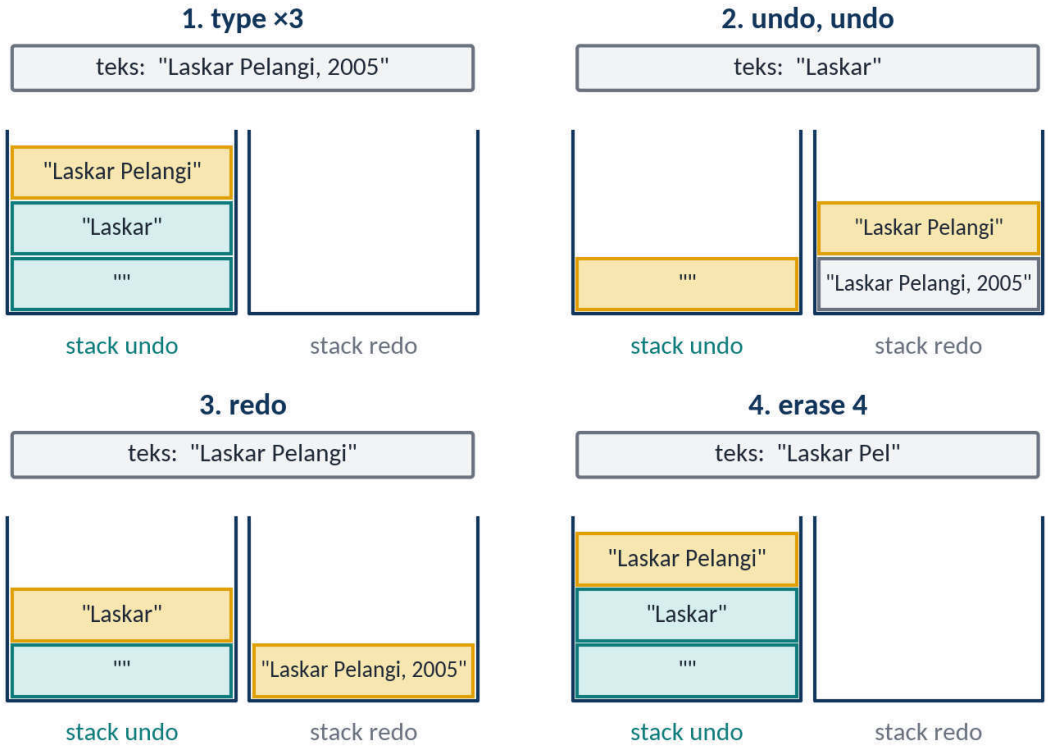
private:
    void report(const string& action) const {
        cout << left << setw(24) << action << setw(26) << ("\"" + text_ + "\"")
             << "undo " << undo_.size() << "  redo " << redo_.size() << "\n";
    }
    string text_;
    stack<string> undo_, redo_;
};
```

Kode 5.8 (Lanjutan) *06_undo_redo.cpp: undo dan redo dengan dua stack berisi keadaan yang disimpan.*

```
int main() {
    Editor e;
    cout << left << setw(24) << "action" << setw(26) << "text" << "stack sizes\n";
    e.type("Laskar");
    e.type(" Pelangi");
    e.type(", 2005");
    e.undo();
    e.undo();
    e.redo();
    e.erase(4);
    e.redo();
    e.undo();
    e.undo();
    e.undo();
    e.undo();
    return 0;
}
```

```
action                 text                    stack sizes
type "Laskar"          "Laskar"                undo 1  redo 0
type " Pelangi"        "Laskar Pelangi"        undo 2  redo 0
type ", 2005"          "Laskar Pelangi, 2005"  undo 3  redo 0
undo                   "Laskar Pelangi"        undo 2  redo 1
undo                   "Laskar"                undo 1  redo 2
redo                   "Laskar Pelangi"        undo 2  redo 1
erase 4                "Laskar Pel"            undo 3  redo 0
redo (nothing to redo) "Laskar Pel"            undo 3  redo 0
undo                   "Laskar Pelangi"        undo 2  redo 1
undo                   "Laskar"                undo 1  redo 2
undo                   ""                      undo 0  redo 3
undo (nothing to undo) ""                      undo 0  redo 3
```

Gambar 5.6 menggambar kedua stack pada empat saat dalam eksekusi itu. Setelah tiga suntingan (panel 1), stack undo memuat tiga keadaan sebelumnya. Dua *undo* (panel 2) memindahkan dua keadaan ke stack redo. Satu *redo* (panel 3) memindahkan satu kembali. Lalu datang kasus yang penting: `erase 4` adalah suntingan baru, sehingga ia melakukan *push* ke stack undo **dan mengosongkan stack redo** (panel 4). Itulah sebabnya `redo` berikutnya pada keluaran melaporkan "nothing to redo". Jika stack redo tidak dikosongkan, sebuah *redo* akan menghidupkan kembali "Laskar Pelangi, 2005" dari riwayat yang sudah ditimpa oleh suntingan baru — *bug* yang sungguh membingungkan pada sebagian penyunting generasi awal.



Gambar 5.6 Stack undo dan redo pada empat saat dalam **Kode 5.8**. Suntingan baru (panel 4) melakukan *push* ke stack undo dan mengosongkan stack redo.

Setiap *undo* dan *redo* berupa satu *pop* dan satu *push*: operasi stack $O(1)$, ditambah biaya menyalin teksnya. Dua stack yang sama menggerakkan tombol Back dan Forward pada peramban web, dengan "suntingan baru" berpadanan dengan mengeklik tautan baru — yang, seperti mungkin sudah Anda perhatikan, menghapus riwayat Forward Anda.

5.8 Aplikasi: Infiks ke Postfiks dengan Algoritma Shunting-Yard

Subbab ini dan berikutnya membangun kalkulator ilmiah kecil, dan bersama-sama keduanya adalah argumen terbaik yang pernah dibuat untuk stack.

5.8.1 Tiga cara menulis jumlah yang sama

Kita menulis aritmetika dalam notasi **infiks**: setiap operator biner berdiri *di antara* kedua operannya, seperti `3 + 2 * 5`. Infiks mudah bagi manusia dan canggung bagi mesin, karena urutan operator dalam teks bukanlah urutan penerapannya. Dalam `3 + 2 * 5`, `*` harus diterapkan lebih dulu walaupun `+` ditulis lebih dulu; untuk mengubahnya kita memerlukan tanda kurung, seperti `(3 + 2) * 5`. Program yang mengevaluasi infiks karenanya harus memahami presedensi, asosiativitas, dan tanda kurung.

Pada tahun 1920-an ahli logika Polandia Jan Łukasiewicz mengamati bahwa jika setiap operator ditulis *sebelum* operannya — notasi **prefiks**, `+ 3 * 2 5` — tanda kurung tidak pernah diperlukan. Bayangan cerminnya, dengan setiap operator ditulis *sesudah* operannya, adalah notasi **postfiks**, `3 2 5 * +`, yang juga disebut *reverse Polish notation* (RPN). Charles Hamblin di Australia serta Friedrich Bauer dan Edsger Dijkstra di Eropa secara terpisah menyadari pada akhir 1950-an bahwa postfiks adalah persis yang dapat dievaluasi mesin yang memiliki stack dalam satu kali pemindaian kiri ke kanan [5], [6]. **Tabel 5.2** membandingkan ketiga notasi.

Tabel 5.2 Ekspresi yang sama dalam notasi infiks, prefiks, dan postfiks. Postfiks tidak memerlukan tanda kurung, dan operatornya muncul persis dalam urutan penerapannya.

Infiks	Prefiks (Polish)	Postfiks (reverse Polish)	Nilai
`3 + 2 * 5`	`+ 3 * 2 5`	`3 2 5 * +`	13
`(3 + 2) * 5`	`* + 3 2 5`	`3 2 + 5 *`	25
`a + b * c - d`	`- + a * b c d`	`a b c * + d -`	9 bila $a=2, b=3, c=4, d=5$
`8 - 3 - 2`	`- - 8 3 2`	`8 3 - 2 -`	3
`2 ^ 3 ^ 2`	`^ 2 ^ 3 2`	`2 3 2 ^ ^`	512

Bacalah kolom postfiks dengan cermat. Operan muncul dalam urutan yang sama dengan kolom infiks — hanya operatornya yang berpindah. Dan operator muncul **persis dalam urutan penerapannya**: dalam `3 2 5 * +` perkalian datang lebih dulu karena dikerjakan lebih dulu. Itulah sebabnya tanda kurung tidak diperlukan: urutan teksnya *adalah* urutan evaluasinya.

5.8.2 Presedensi dan asosiativitas

Untuk mengubah infiks menjadi postfiks, kita harus tahu operator mana yang mengikat lebih kuat. **Tabel 5.3** memberikan aturan yang dipakai dalam bab ini; aturannya adalah aritmetika sekolah biasa, dengan `^` untuk perpangkatan.

Tabel 5.3 Presedensi dan asosiativitas operator yang dipakai kalkulator.

Operator	Arti	Presedensi	Asosiativitas
`^`	pangkat	3 (tertinggi)	kanan: `2^3^2` berarti `2^(3^2)`
`*`, `/`	kali, bagi	2	kiri: `100/10/5` berarti `(100/10)/5`
`+`, `-`	tambah, kurang	1 (terendah)	kiri: `8-3-2` berarti `(8-3)-2`

Presedensi memutuskan antara operator yang berbeda: `*` mengalahkan `+`. *Asosiativitas* memutuskan antara operator yang setara: pengurangan dikelompokkan dari kiri, sehingga `8-3-2`

adalah $(8-3)-2=3$ dan bukan $8-(3-2)=7$; perpangkatan dikelompokkan dari kanan, sehingga `2^3^2` adalah $2^{9}=512$ dan bukan $8^{2}=64$.

5.8.3 Algoritmanya

Konversi ini diterbitkan Dijkstra pada tahun 1961 [6], yang menamainya menurut *shunting yard* (emplasemen langsir kereta api): operan melaju lurus dari masukan ke keluaran, sedangkan operator dibelokkan ke jalur samping — sebuah stack — dan menunggu di sana sampai tiba gilirannya bergabung kembali dengan rangkaian. Aturan kapan sebuah operator boleh meninggalkan jalur samping adalah seluruh algoritmanya.

Pseudokode 5.3 Algoritma *shunting-yard* Dijkstra: infiks ke postfiks dengan satu stack operator.

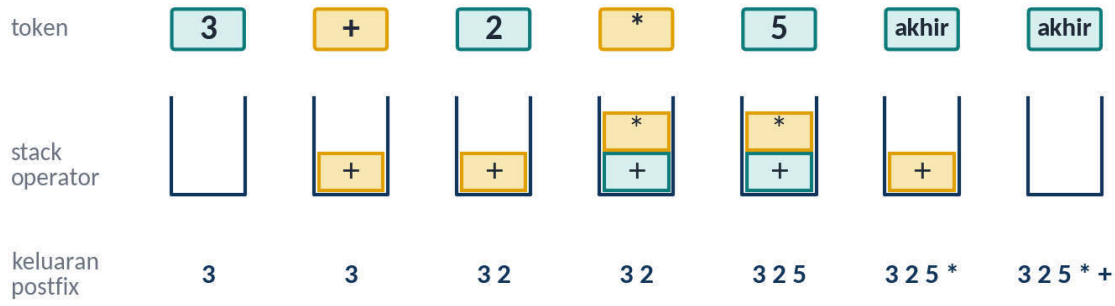
```
INFIX-TO-POSTFIX(tokens)
 1.  S ← stack kosong;  out ← daftar kosong
 2.  for setiap token t, dari kiri ke kanan
 3.      if t adalah operan (bilangan atau variabel)
 4.          tambahkan t ke out
 5.      else if t = "("
 6.          PUSH(S, t)
 7.      else if t = ")"
 8.          while PEEK(S) ≠ "("
 9.              tambahkan POP(S) ke out
10.          POP(S)                          // buang "("; galat bila S habis
11.      else                                // t adalah operator
12.          while S tidak kosong and PEEK(S) ≠ "(" and
                   ( prec(PEEK(S)) > prec(t) or
                     ( prec(PEEK(S)) = prec(t) and t asosiatif-kiri ) )
13.              tambahkan POP(S) ke out
14.          PUSH(S, t)
15.  while S tidak kosong
16.      tambahkan POP(S) ke out             // galat bila tersisa "("
17.  return out
```

Dengan kata sederhana: operan langsung ke keluaran. Tanda kurung buka masuk ke stack sebagai pembatas. Tanda kurung tutup mengeluarkan operator ke keluaran sampai bertemu pembatasnya, yang kemudian dibuang. Operator yang datang terlebih dahulu mengeluarkan setiap operator di stack yang harus diterapkan *sebelum* dirinya — yang presedensinya lebih tinggi, atau yang presedensinya sama bila operator yang datang bersifat asosiatif-kiri — lalu ia sendiri menunggu di stack. Di akhir, apa pun yang masih menunggu dikeluarkan ke keluaran.

Baris 12 adalah satu-satunya baris yang halus, jadi berikut sekali lagi dengan contoh. Ketika `-` datang dalam `a+b*c-d`, stack berisi `+ *`. `*` presedensinya lebih tinggi daripada `-`, sehingga harus diterapkan lebih dulu: keluarkan. Lalu `+` presedensinya *sama* dengan `-`, dan `-` asosiatif-kiri, sehingga `a+b*c` harus diselesaikan sebelum `-` diterapkan: keluarkan `+` juga. Barulah `-` di-*push*. Untuk `^` yang asosiatif-kanan, kasus presedensi sama **tidak** mengeluarkan apa pun, dan itulah yang membuat `2^3^2` dikelompokkan dari kanan.

5.8.4 Menelusuri `3+2*5`

Gambar 5.7 menelusuri contoh menarik yang terkecil, token demi token.



Gambar 5.7 Mengubah `3+2*5` menjadi postfiks. `*` di-*push* di atas `+` karena mengikat lebih kuat; di akhir stack dikosongkan dari puncak, menghasilkan `3 2 5 * +`.

1. `3` adalah operan: keluaran `3`.
2. `+` bertemu stack kosong: *push*.
3. `2`: keluaran `3 2`.
4. `*` bertemu `+` di puncak. Apakah `prec(+) > prec(*)`? Tidak, $1 < 2$. Jadi tidak ada yang dikeluarkan, dan `*` di-*push* di atas `+`. Perbandingan tunggal inilah yang membuat perkalian dikerjakan sebelum penjumlahan.
5. `5`: keluaran `3 2 5`.
6. Akhir masukan: *pop* `*`, keluaran `3 2 5 *`; *pop* `+`, keluaran `3 2 5 * +`.

5.8.5 Menelusuri `a+b*c-d`

Tabel 5.4 menelusuri contoh yang lebih panjang dengan variabel, termasuk dua kali *pop* yang dijelaskan setelah **Pseudokode 5.3**.

Tabel 5.4 Mengubah `a+b*c-d` menjadi postfiks, satu token per baris.

Token	Tindakan	Stack (dasar → puncak)	Keluaran sejauh ini
`a`	keluarkan operan	(kosong)	`a`
`+`	*push* `+`	`+`	`a`
`b`	keluarkan operan	`+`	`a b`
`*`	prec(+) < prec(∗): *push* `*`	`+ *`	`a b`
`c`	keluarkan operan	`+ *`	`a b c`
`-`	*pop* `*` (lebih tinggi), *pop* `+` (sama, asosiatif-kiri), *push* `-`	`-`	`a b c * +`
`d`	keluarkan operan	`-`	`a b c * + d`
akhir	*pop* `-`	(kosong)	`a b c * + d -`

Hasil `a b c * + d -` dibaca: kalikan `b` dan `c`; tambahkan `a`; kurangi `d`. Itu persis $(a + (b \times c)) - d$, makna teks infiksnya.

5.9 Aplikasi: Mengevaluasi Ekspresi Postfiks

Begitu sebuah ekspresi berada dalam bentuk postfiks, mengevaluasinya hanya memerlukan satu stack berisi *bilangan* dan satu aturan.

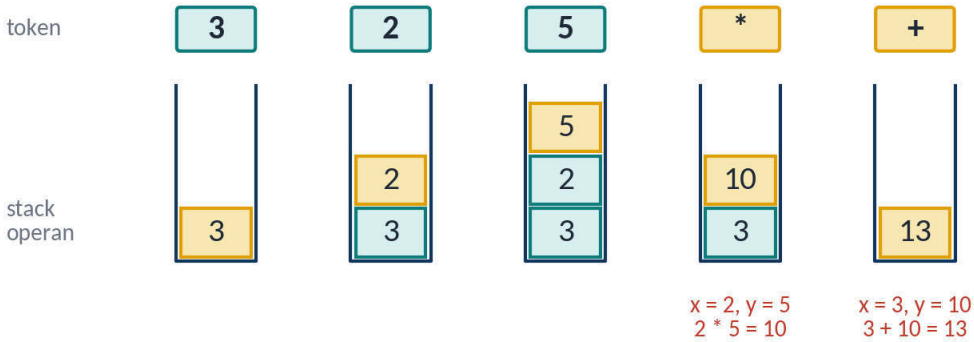
Pseudokode 5.4 Mengevaluasi ekspresi postfiks dengan satu stack operan.

```
EVALUATE-POSTFIX(tokens)
 1.  S ← stack kosong
 2.  for setiap token t, dari kiri ke kanan
 3.      if t adalah operan
 4.          PUSH(S, nilai t)
 5.      else                                  // t adalah operator biner
 6.          if S memuat kurang dari 2 nilai
 7.              error "operator needs two operands"
 8.          y ← POP(S)                        // POP PERTAMA adalah operan KANAN
 9.          x ← POP(S)
10.          PUSH(S, x t y)
11.  if S tidak memuat tepat satu nilai
12.      error "malformed expression"
13.  return POP(S)
```

Dengan kata sederhana: *push* bilangan; ketika operator datang, dua bilangan yang dibutuhkannya adalah dua bilangan paling akhir, jadi *pop* keduanya, terapkan operatornya, dan *push* hasilnya, yang mungkin menjadi operan bagi operator berikutnya. Di akhir harus tersisa tepat satu bilangan.

Baris 8 dan 9 menyembunyikan jebakan klasik. Untuk `8 3 -`, stack berisi `8 3` dengan 3 di puncak, sehingga *pop* pertama mengembalikan `3` dan yang kedua mengembalikan `8`. Operatornya harus menghitung $x - y = 8 - 3 = 5$. Menghitung $y - x$ akan menghasilkan -5. Penjumlahan dan perkalian tidak peduli urutan, dan justru itulah sebabnya *bug* ini lolos uji pada `3 2 5 * +` lalu muncul pada pengurangan atau pembagian pertama.

Gambar 5.8 mengevaluasi `3 2 5 * +`.



Gambar 5.8 Mengevaluasi `3 2 5 * +`. Operan di-*push*; setiap operator mengambil y (puncak) lalu x, dan melakukan *push* x op y. Satu nilai yang tersisa, 13, adalah hasilnya.

5.9.1 Kalkulator lengkap

Kode 5.9 menggabungkan kedua algoritma. Program ini memecah masukan menjadi token (agar bilangan bermulti-digit seperti `100` dapat dipakai), mengubahnya menjadi postfiks dengan **Pseudokode 5.3**, mengevaluasinya dengan **Pseudokode 5.4**, mencetak penelusuran lengkap untuk kedua contoh kerja, lalu menjalankan sekumpulan kasus yang lebih sulit — termasuk empat masukan yang sengaja salah.

Kode 5.9 `07_calculator.cpp`: kalkulator yang mengubah infiks menjadi postfiks dengan algoritma *shunting-yard* dan mengevaluasi postfiks dengan stack operan.

```cpp
// The scientific calculator: infix -> postfix (shunting-yard), then postfix evaluation.
#include <cctype>
#include <iomanip>
#include <iostream>
#include <map>
#include <sstream>
#include <stdexcept>
#include <string>
#include <vector>
using namespace std;

map<char, long long> vars = {{'a', 2}, {'b', 3}, {'c', 4}, {'d', 5}};

int  prec(const string& op)       { return op == "^" ? 3 : (op == "*" || op == "/") ? 2 :
1; }
bool rightAssoc(const string& op) { return op == "^"; }
bool isOperator(const string& t)  { return t == "+" || t == "-" || t == "*" || t == "/" ||
t == "^"; }

vector<string> tokenize(const string& s) {            // "12+x" -> {"12", "+", "x"}
    vector<string> out;
    for (size_t i = 0; i < s.size();) {
        if (isspace(static_cast<unsigned char>(s[i]))) { ++i; continue; }
        if (isdigit(static_cast<unsigned char>(s[i]))) {
            size_t j = i;
            while (j < s.size() && isdigit(static_cast<unsigned char>(s[j]))) ++j;
            out.push_back(s.substr(i, j - i));
            i = j;
        } else {
            out.push_back(string(1, s[i++]));
        }
    }
    return out;
}

string join(const vector<string>& v) {
    string r;
    for (size_t i = 0; i < v.size(); ++i) r += (i ? " " : "") + v[i];
    return r;
}
```

Kode 5.9 (Lanjutan) *07_calculator.cpp: kalkulator yang mengubah infiks menjadi postfiks dengan algoritma shunting-yard dan mengevaluasi postfiks dengan stack operan.*

```
void row(const string& tok, const string& action, const string& stack, const string& out) {
    ostringstream line;
    line << "  " << left << setw(6) << tok << setw(24) << action << setw(10) << stack <<
out;
    string text = line.str();
    while (!text.empty() && text.back() == ' ') text.pop_back();   // no trailing blanks
    cout << text << "\n";
}

vector<string> toPostfix(const string& infix, bool trace) {
    vector<string> out, ops;                          // ops is the operator stack (back
= top)
    if (trace) row("token", "action", "stack", "postfix");
    for (const string& t : tokenize(infix)) {
        string action;
        if (isalnum(static_cast<unsigned char>(t[0]))) {
            out.push_back(t);
            action = "output operand";
        } else if (t == "(") {
            ops.push_back(t);
            action = "push (";
        } else if (t == ")") {
            while (!ops.empty() && ops.back() != "(") {
                action += "pop " + ops.back() + ", ";
                out.push_back(ops.back());
                ops.pop_back();
            }
            if (ops.empty()) throw runtime_error("unmatched ')'");
            ops.pop_back();                           // discard the "("
            action += "drop (";
        } else if (isOperator(t)) {
            while (!ops.empty() && ops.back() != "(" &&
                   (prec(ops.back()) > prec(t) || (prec(ops.back()) == prec(t) &&
!rightAssoc(t)))) {
                action += "pop " + ops.back() + ", ";
                out.push_back(ops.back());
                ops.pop_back();
            }
            ops.push_back(t);
            action += "push " + t;
        } else {
            throw runtime_error("unknown symbol '" + t + "'");
        }
        if (trace) row(t, action, join(ops), join(out));
    }
    while (!ops.empty()) {                            // drain what is left
        if (ops.back() == "(") throw runtime_error("unmatched '('");
        out.push_back(ops.back());
        ops.pop_back();
```

Kode 5.9 (Lanjutan) *07_calculator.cpp: kalkulator yang mengubah infiks menjadi postfiks dengan algoritma shunting-yard dan mengevaluasi postfiks dengan stack operan.*

```
        if (trace) row("end", "pop " + out.back(), join(ops), join(out));
    }
    return out;
}

long long apply(long long x, const string& op, long long y) {
    if (op == "+") return x + y;
    if (op == "-") return x - y;
    if (op == "*") return x * y;
    if (op == "/") { if (y == 0) throw runtime_error("division by zero"); return x / y; }
    long long r = 1;                                    // "^": y >= 0 in this book
    for (long long i = 0; i < y; ++i) r *= x;
    return r;
}

long long evaluate(const vector<string>& postfix, bool trace) {
    vector<long long> st;                               // the operand stack (back = top)
    auto show = [&]() { string r; for (size_t i = 0; i < st.size(); ++i) r += (i ? " " :
"") + to_string(st[i]); return r; };
    if (trace) row("token", "action", "stack", "");
    for (const string& t : postfix) {
        string action;
        if (isOperator(t)) {
            if (st.size() < 2) throw runtime_error("operator " + t + " needs two
operands");
            long long y = st.back(); st.pop_back();     // the FIRST pop is the right
operand
            long long x = st.back(); st.pop_back();
            st.push_back(apply(x, t, y));
            action = to_string(x) + " " + t + " " + to_string(y) + " = " +
to_string(st.back());
        } else {
            long long v = isdigit(static_cast<unsigned char>(t[0])) ? stoll(t) :
vars.at(t[0]);
            st.push_back(v);
            action = "push " + to_string(v);
        }
        if (trace) row(t, action, show(), "");
    }
    if (st.size() != 1) throw runtime_error("malformed expression");
    return st.back();
}
```

Kode 5.9 (Lanjutan) *07_calculator.cpp: kalkulator yang mengubah infiks menjadi postfiks dengan algoritma shunting-yard dan mengevaluasi postfiks dengan stack operan.*

```
int main() {
    for (const string e : {"3+2*5", "a+b*c-d"}) {
        cout << "infix " << e << "  (a=2, b=3, c=4, d=5)\n";
        vector<string> p = toPostfix(e, true);
        cout << "postfix: " << join(p) << "\n";
        long long v = evaluate(p, true);
        cout << "value: " << v << "\n\n";
    }
    for (const string e : {"(7+3)*(10-4)/5", "2^3^2", "(2^3)^2", "100/10/5", "8-3-2",
                           "(1+2", "1+2)", "1+", "7/(3-3)"}) {
        cout << left << setw(16) << e;
        try {
            vector<string> p = toPostfix(e, false);
            cout << "postfix " << setw(22) << join(p);
            long long v = evaluate(p, false);
            cout << "= " << v << "\n";
        } catch (const exception& ex) {
            cout << "error: " << ex.what() << "\n";
        }
    }
    return 0;
}
```

```
infix 3+2*5  (a=2, b=3, c=4, d=5)
  token action                  stack     postfix
  3     output operand                    3
  +     push +                  +         3
  2     output operand          +         3 2
  *     push *                  + *       3 2
  5     output operand          + *       3 2 5
  end   pop *                   +         3 2 5 *
  end   pop +                               3 2 5 * +
postfix: 3 2 5 * +
  token action                  stack
  3     push 3                  3
  2     push 2                  3 2
  5     push 5                  3 2 5
  *     2 * 5 = 10              3 10
  +     3 + 10 = 13             13
value: 13

infix a+b*c-d  (a=2, b=3, c=4, d=5)
  token action                  stack     postfix
  a     output operand                    a
  +     push +                  +         a
  b     output operand          +         a b
  *     push *                  + *       a b
  c     output operand          + *       a b c
  -     pop *, pop +, push -    -         a b c * +
  d     output operand          -         a b c * + d
  end   pop -                             a b c * + d -
postfix: a b c * + d -
  token action                  stack
  a     push 2                  2
  b     push 3                  2 3
  c     push 4                  2 3 4
  *     3 * 4 = 12              2 12
  +     2 + 12 = 14             14
  d     push 5                  14 5
  -     14 - 5 = 9              9
value: 9

(7+3)*(10-4)/5  postfix 7 3 + 10 4 - * 5 /    = 12
2^3^2           postfix 2 3 2 ^ ^             = 512
(2^3)^2         postfix 2 3 ^ 2 ^             = 64
100/10/5        postfix 100 10 / 5 /          = 2
8-3-2           postfix 8 3 - 2 -             = 3
(1+2            error: unmatched '('
1+2)            error: unmatched ')'
1+              postfix 1 +                   error: operator + needs two operands
7/(3-3)         postfix 7 3 3 - /             error: division by zero
```

Periksa penelusurannya terhadap **Gambar 5.7**, **Tabel 5.4**, dan **Gambar 5.8**: semuanya cocok token demi token, dan kedua nilainya, 13 dan 9, sama dengan hasil $3 + 2 \cdot 5$ dan $2 + 3 \cdot 4 - 5$ yang dihitung dengan tangan. Kumpulan kasus berikutnya menegaskan sisa teorinya:

- `(7+3)*(10-4)/5` memperlihatkan tanda kurung bekerja — setiap `)` mengeluarkan operator sampai `(`-nya — dan bilangan bermulti-digit: $10 \cdot 6/5 = 12$.
- `2^3^2` menghasilkan 512 dan `(2^3)^2` menghasilkan 64: asosiativitas kanan, dan tanda kurung yang mengesampingkannya.
- `100/10/5` menghasilkan 2 dan `8-3-2` menghasilkan 3: asosiativitas kiri, dengan x dan y dalam urutan yang benar.
- `(1+2` dan `1+2)` tertangkap saat konversi (tanda kurung tanpa pasangan adalah kegagalan pencocokan tanda kurung, Subbab 5.6); `1+` tertangkap saat evaluasi karena `+` hanya menemukan satu operan; dan `7/(3-3)` tertangkap sebagai pembagian dengan nol.

Kedua fase membaca setiap token sekali dan melakukan pekerjaan stack $O(1)$ per token di luar *pop*, dan setiap token di-*push* serta di-*pop* paling banyak sekali. Karena itu konversi dan evaluasi masing-masing $O(n)$ untuk n token. Versi Java dan Python dari kalkulator ini di folder Code mencetak keluaran yang identik; versi Python memerlukan satu baris tambahan, karena `//` di Python membulatkan ke arah minus tak hingga, sedangkan C++ dan Java memotong ke arah nol.

5.10 Aplikasi: Call Stack Fungsi

Stack terpenting di komputer Anda adalah stack yang telah Anda pakai dalam setiap program yang pernah Anda tulis, tanpa pernah melihatnya.

5.10.1 Yang terjadi ketika fungsi dipanggil

Ketika `main` memanggil `lateFee(7, "reference")`, prosesor harus mengingat di mana melanjutkan `main` ketika `lateFee` kembali, dan `lateFee` memerlukan tempat untuk menyimpan parameter dan variabel lokalnya. Ketika `lateFee` pada gilirannya memanggil `dailyRate`, hal yang sama terjadi lagi, satu tingkat lebih dalam. Ketika `dailyRate` kembali, prosesor harus kembali ke `lateFee` — pemanggil paling akhir — bukan ke `main`. Fungsi yang terakhir dipanggil adalah yang pertama kembali: pemanggilan fungsi bersifat LIFO.

Karena itu sistem eksekusi memelihara sebuah **call stack**. Setiap pemanggilan melakukan *push* sebuah *stack frame* (disebut juga *activation record*) yang memuat:

- **alamat kembali** — tempat melanjutkan di pemanggil;
- **parameter** fungsi;
- **variabel lokal**-nya;
- register prosesor yang disimpan dan data pembukuan lain.

Setiap *return* melakukan *pop* terhadap *frame* itu, dan bersamanya setiap variabel lokal dari pemanggilan tersebut. Inilah sebabnya variabel lokal "lenyap" ketika fungsinya kembali, dan mengapa sebuah fungsi tidak boleh mengembalikan pointer atau referensi ke salah satu variabel lokalnya sendiri: *frame* tempat variabel itu hidup sudah di-*pop* dan akan ditimpa oleh pemanggilan berikutnya.

5.10.2 Menyaksikannya terjadi

Call stack itu sendiri dikelola oleh kompilator dan prosesor, sehingga kita tidak dapat mencetaknya secara langsung. Namun kita dapat memelihara salinannya sendiri. Dalam **Kode 5.10**, setiap fungsi membuat objek `Frame` sebagai pernyataan pertamanya; konstruktornya melakukan *push* nama fungsi ke sebuah vektor, dan destruktornya — yang dijalankan C++ secara otomatis ketika fungsi kembali, melalui jalur mana pun — melakukan *pop*.

Kode 5.10 `08_call_stack.cpp`: menampakkan call stack dengan melakukan *push* nama pada setiap masuk fungsi dan *pop* pada setiap kembali.

```cpp
// Making the call stack visible: every function pushes a frame on entry and pops it on return.
#include <iostream>
#include <string>
#include <vector>
using namespace std;

vector<string> callStack;                                   // our own copy of what the machine keeps

string picture() {                                          // bottom -> top
    string s;
    for (size_t i = 0; i < callStack.size(); ++i) s += (i ? " | " : "") + callStack[i];
    return s;
}

struct Frame {                                              // constructor = push, destructor = pop
    explicit Frame(const string& name) {
        callStack.push_back(name);
        cout << "push  " << picture() << "\n";
    }
    ~Frame() {
        callStack.pop_back();
        cout << "pop   " << (callStack.empty() ? "(empty)" : picture()) << "\n";
    }
    Frame(const Frame&) = delete;
    Frame& operator=(const Frame&) = delete;
};

int dailyRate(const string& category) {
    Frame f("dailyRate(" + category + ")");
    return category == "reference" ? 5000 : 1000;           // rupiah per day
}

int capFee(int fee, int cap) {
    Frame f("capFee(" + to_string(fee) + ")");
    return fee < cap ? fee : cap;
}

int lateFee(int days, const string& category) {
    Frame f("lateFee(" + to_string(days) + ")");
    int fee = days * dailyRate(category);                   // a call inside an expression
    return capFee(fee, 25000);
}
```

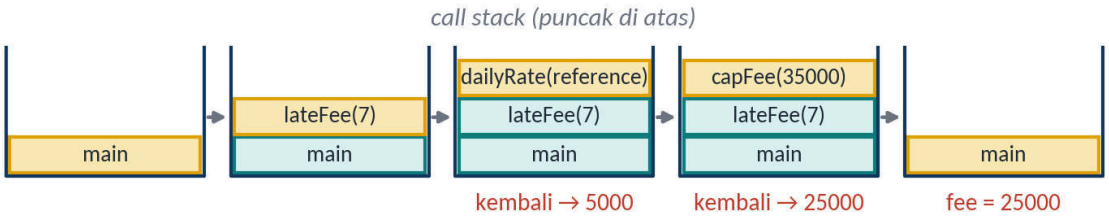
Kode 5.10 (Lanjutan) *08_call_stack.cpp: menampakkan call stack dengan melakukan push nama pada setiap masuk fungsi dan pop pada setiap kembali.*

```cpp
int countdown(int n) {                                      // a preview of Chapter 10
    Frame f("countdown(" + to_string(n) + ")");
    if (n == 0) return 0;
    return 1 + countdown(n - 1);
}

int main() {
    Frame f("main");
    int fee = lateFee(7, "reference");
    cout << "        late fee = Rp" << fee << "\n";
    int steps = countdown(2);
    cout << "        countdown made " << steps << " recursive calls\n";
    return 0;
}
```

```
push  main
push  main | lateFee(7)
push  main | lateFee(7) | dailyRate(reference)
pop   main | lateFee(7)
push  main | lateFee(7) | capFee(35000)
pop   main | lateFee(7)
pop   main
        late fee = Rp25000
push  main | countdown(2)
push  main | countdown(2) | countdown(1)
push  main | countdown(2) | countdown(1) | countdown(0)
pop   main | countdown(2) | countdown(1)
pop   main | countdown(2)
pop   main
        countdown made 2 recursive calls
pop   (empty)
```

Gambar 5.9 menggambar paruh pertama eksekusi itu. Perhatikan urutan kejadian dalam `lateFee`: ekspresi `days * dailyRate(category)` memanggil `dailyRate` *di tengah* penghitungan `fee`, sehingga *frame* `dailyRate` di-*push* di atas *frame* `lateFee`, mengembalikan 5000, dan di-*pop* bahkan sebelum `capFee` dipanggil. Kemudian `capFee` mendapat *frame*-nya sendiri di tempat yang sama, mengembalikan 25000 (biaya keterlambatan 7 × 5000 = 35000 dibatasi Rp25.000), dan di-*pop*. Akhirnya `lateFee` kembali dan hanya `main` yang tersisa.



Gambar 5.9 Call stack selama `lateFee(7, "reference")`: setiap pemanggilan melakukan *push* sebuah *frame* di puncak, dan setiap *return* melakukan *pop*, menyerahkan nilainya kepada *frame* di bawahnya.

Paruh kedua keluaran adalah pratinjau Bab 10. `countdown(2)` memanggil `countdown(1)`, yang memanggil `countdown(0)`: tiga *frame* dari fungsi yang *sama* berada di stack sekaligus, masing-masing dengan `n`-nya sendiri. Rekursi tidak lebih dari fungsi yang *frame*-nya di-*push* di atas *frame*-nya sendiri. Ini juga menjelaskan bahayanya. Setiap *frame* memakan memori, dan call stack itu kecil — secara bawaan 1 MB untuk program Windows yang dibangun dengan *linker* Microsoft [7], umumnya 8 MB di Linux (`ulimit -s` mencetak 8192 pada mesin yang dipakai untuk buku ini), dan Python menolak masuk lebih dalam dari 1.000 pemanggilan bersarang kecuali diperintahkan lain. Rekursi yang tidak pernah mencapai kasus dasarnya terus melakukan *push* *frame* sampai memori itu habis: **stack overflow**, galat yang memberi nama sebuah situs pemrograman terkenal.

Versi Java dan Python program ini memakai `try`/`finally` dan *context manager* (`with`) untuk mendapatkan perilaku *push* saat masuk dan *pop* saat keluar yang sama, dan mencetak baris-baris yang sama.

5.11 Queue sebagai Tipe Data Abstrak

Sekarang beralihlah ke disiplin lain pada **Gambar 5.1**.

> **DEFINISI.** *Queue* (antrean) adalah kumpulan elemen yang penyisipannya terjadi di satu ujung, *rear* (belakang, disebut juga *tail*), dan pengambilannya terjadi di ujung lain, *front* (depan,

disebut juga *head*). Elemen yang diambil selalu yang paling lama berada di dalam queue. Disiplin ini disebut *first in, first out* (FIFO): pertama masuk, pertama keluar.

Queue adalah struktur paling adil dalam komputasi, dan kehidupan sehari-hari penuh dengannya: antrean di teller bank atau ATM, mobil di gerbang tol Surabaya–Gempol, penumpang di pemeriksaan keamanan bandara, pembeli di kasir swalayan. Di dalam komputer, queue bahkan lebih umum lagi:

- **Pekerjaan cetak.** Beberapa orang mengirim dokumen A.doc, B.doc, C.doc, dan D.doc ke satu pencetak bersama. Pencetak tidak dapat mencetak empat sekaligus, jadi ia mengantrekannya menurut urutan kedatangan dan mencetaknya satu per satu.
- **Proses dan *thread*.** Sistem operasi menyimpan proses yang siap dijalankan dalam *ready queue*. Pada penjadwalan *round-robin*, proses di depan berjalan selama satu irisan waktu yang singkat lalu, jika belum selesai, kembali ke belakang.
- **Paket jaringan.** Router yang menerima paket lebih cepat daripada kemampuan tautan keluarnya mengirim menyimpan paket-paket itu dalam queue — sebuah *buffer* — dan meneruskannya secara berurutan.
- **Masukan dan keluaran.** Tombol yang diketik lebih cepat daripada program membacanya menunggu dalam *buffer* papan ketik; *message queue* Windows menampung klik tetikus dan tekanan tombol bagi *event loop* sebuah aplikasi.
- **Penelusuran graf.** *Breadth-first search* (Bab 13) mengunjungi graf tingkat demi tingkat dengan menyimpan simpul yang sudah ditemukan tetapi belum dijelajahi dalam sebuah queue.

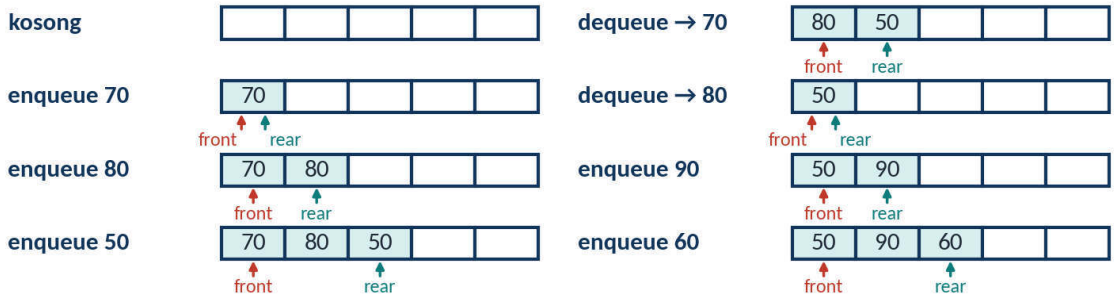
5.11.1 Operasi-operasinya

Tabel 5.5 mendaftar operasi queue. Seperti pada stack, namanya berbeda-beda antarbuku dan antarpustaka — `enqueue` disebut juga *add*, *offer*, *push*, atau *put*; `dequeue` disebut juga *remove*, *poll*, *pop*, atau *get* — tetapi perilakunya tidak.

Tabel 5.5 Operasi ADT queue dan biayanya pada implementasi larik sirkular di Subbab 5.13.

Operasi	Yang dilakukan	Prasyarat	Biaya (larik sirkular)
`enqueue(x)`	Menambahkan x di belakang	Queue tidak penuh (khusus queue berkapasitas tetap)	$O(1)$
`dequeue()`	Mengambil elemen depan dan mengembalikannya	Queue tidak kosong	$O(1)$
`peek()` / `front()`	Mengembalikan elemen depan tanpa mengambilnya	Queue tidak kosong	$O(1)$
`isEmpty()`	Bernilai benar bila queue tidak memuat elemen	—	$O(1)$
`isFull()`	Bernilai benar bila queue berkapasitas tetap tidak punya slot kosong	—	$O(1)$
`size()`	Banyaknya elemen	—	$O(1)$

Gambar 5.10 menjalankan barisan pendek pada sebuah queue: *enqueue* 70, 80, dan 50; *dequeue* dua kali; *enqueue* 90 dan 60. Perhatikan elemen mana yang keluar. Walaupun 50 adalah kedatangan terbaru pada saat *dequeue* pertama, yang keluar adalah 70 — elemen yang paling lama menunggu; lalu 80. Ketika 90 dan 60 tiba, keduanya bergabung di belakang 50.



Gambar 5.10 Sebuah queue melalui tujuh operasi, digambar dengan bagian depan di kiri. Elemen keluar persis dalam urutan kedatangannya: 70, lalu 80.

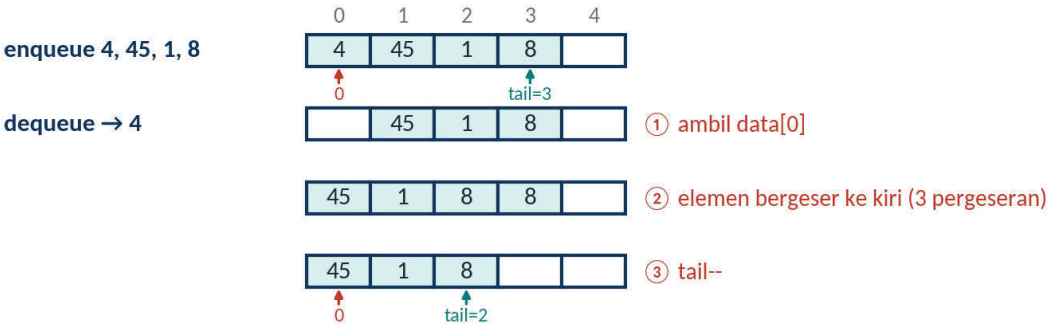
Gambar itu sengaja menyembunyikan *bagaimana* elemen disimpan: queue digambar ulang dari tepi kiri setiap kali. Dua subbab berikutnya memperlihatkan bahwa menyimpan queue dalam larik lebih sulit daripada menyimpan stack, karena queue berubah di **kedua** ujungnya.

5.12 Queue dalam Larik Linear, dan Masalah Pergeseran

Stack cukup memerlukan satu indeks karena ia berubah di satu ujung. Queue berubah di dua ujung, sehingga implementasi larik yang paling jelas memegang dua indeks: `head` untuk bagian depan dan `tail` untuk bagian belakang. Ada dua cara yang jelas untuk memakainya, dan keduanya cacat dengan cara yang mendidik.

5.12.1 Percobaan pertama: geser semuanya pada setiap *dequeue*

Pertahankan bagian depan secara permanen di slot 0. *Enqueue* menulis di `++tail`, persis seperti *push*. *Dequeue* mengambil `data[0]`, lalu menggeser setiap elemen yang tersisa satu slot ke kiri untuk menutup celahnya, lalu mengurangi `tail`. **Gambar 5.11** memperlihatkan satu *dequeue* dari queue `4 45 1 8`.



Gambar 5.11 Queue bergeser: *dequeue* mengambil slot 0 (langkah 1), menggeser setiap elemen lain satu tempat ke kiri (langkah 2), dan mengurangi `tail` (langkah 3). Queue berisi n elemen memerlukan $n - 1$ pergeseran per *dequeue*.

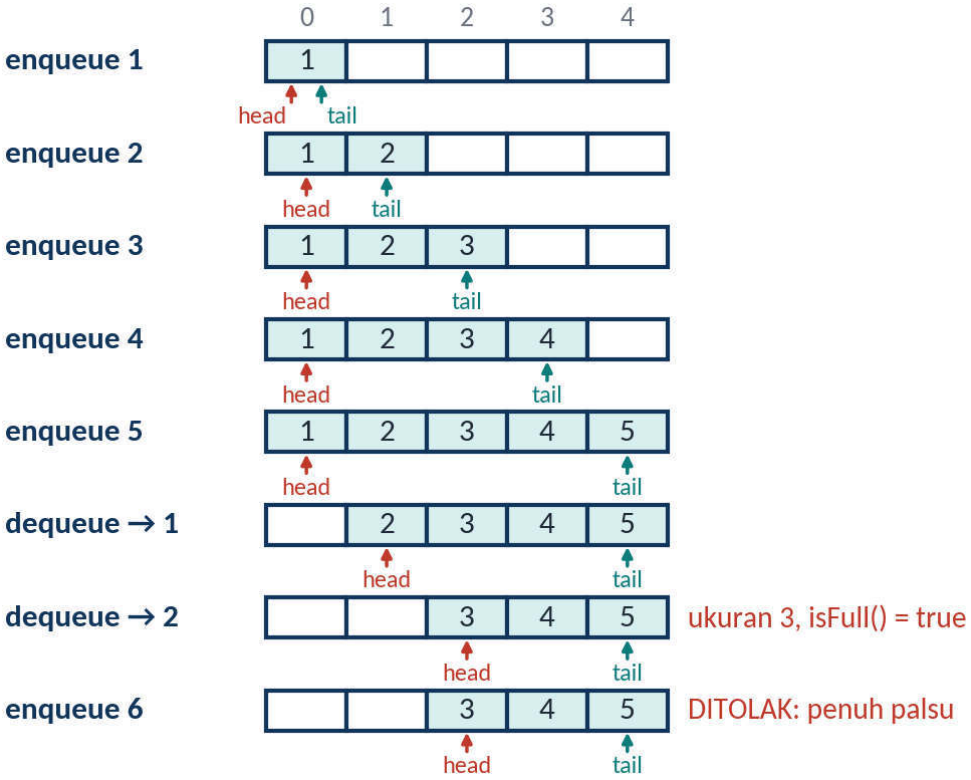
Versi ini benar dan mudah dipahami, dan tidak pernah memboroskan slot. Namun langkah 2 adalah perulangan atas seluruh queue: *dequeue* memerlukan $n - 1$ pergeseran elemen untuk queue berisi n elemen, sehingga biayanya $O(n)$, bukan $O(1)$. Antrean cetak berisi sepuluh dokumen tidak peduli; antrean router berisi seratus ribu paket sangat peduli.

Langkah 3 juga langkah yang paling mudah terlupa. Tanpa `tail--`, queue mengira masih memuat empat elemen setelah *dequeue* — dengan elemen terakhir kini terduplikasi — seperti yang terlihat jelas pada baris tengah **Gambar 5.11** (`45 1 8 8`). *Dequeue* yang menggeser tetapi tidak

mengurangi `tail`, dan perulangan pencetakan yang berhenti pada `i < tail` alih-alih `i <= tail`, adalah dua keteledoran yang muncul berulang-ulang pada percobaan pertama menulis kode ini.

5.12.2 Percobaan kedua: biarkan indeksnya bergerak

Agar *dequeue* menjadi $O(1)$, berhentilah menggeser. Biarkan *dequeue* sekadar mengambil `data[head]` lalu menaikkan `head`. Sekarang kedua indeks bergerak ke kanan: *enqueue* menggerakkan `tail`, *dequeue* menggerakkan `head`, dan elemen tetap di tempat ia ditulis. Queue adalah bentangan slot dari `head` sampai `tail`, dan **Gambar 5.12** memperlihatkannya bergeser melintasi larik.



Gambar 5.12 Queue yang bergeser. Kedua indeks hanya bergerak ke kanan. Setelah dua *dequeue*, slot 0 dan 1 kosong, tetapi `tail` sudah mencapai slot terakhir, sehingga `isFull()` melaporkan benar dan *enqueue* berikutnya ditolak: *penuh palsu*.

Setiap operasi kini $O(1)$. Namun lihat dua baris terakhir. Setelah lima *enqueue* dan dua *dequeue*, queue memuat tiga elemen dalam larik lima slot, tetapi `tail = 4 = MAX - 1`, dan uji `isFull()` — "apakah `tail` sudah mencapai slot terakhir?" — menjawab ya. *Enqueue* angka 6 ditolak padahal dua slot kosong. Inilah **masalah pergeseran** (*drifting problem*), disebut juga *penuh palsu* (*false full*): indeks hanya pernah bergerak ke kanan, sehingga setiap slot di sebelah kiri `head` hilang selamanya. Dengan lalu lintas yang cukup, queue bergeser berkapasitas 1.000 akan menolak elemen berikutnya setelah *enqueue* keseribu, bahkan ketika queue itu kosong.

Kode 5.11 menjalankan kedua percobaan dan mencetak keadaannya.

Kode 5.11 `09_linear_queue.cpp`: queue bergeser-elemen (benar tetapi $O(n)$ per *dequeue*) dan queue yang indeksnya bergeser ($O(1)$ tetapi penuh palsu).

```cpp
// Two linear-array queues: one that shifts on every dequeue, one whose indices drift.
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

const int CAP = 5;

string pad(const string& label) {                       // left-justify a label in 15
columns
    ostringstream o;
    o << left << setw(15) << label;
    return o.str();
}

string cells(const int a[], int lo, int hi) {           // print slots, "." for unused
    string s = "[";
    for (int i = 0; i < CAP; ++i) s += " " + (i >= lo && i <= hi ? to_string(a[i]) :
string("."));
    return s + " ]";
}

// Version 1: the front is always slot 0; dequeue shifts everything one place left.
struct ShiftQueue {
    int data[CAP];
    int tail = -1;                                      // index of the last element
    long moves = 0;                                     // element moves done by shifting
    bool isEmpty() const { return tail == -1; }
    bool isFull()  const { return tail == CAP - 1; }
    bool enqueue(int v) { if (isFull()) return false; data[++tail] = v; return true; }
    int dequeue() {                                     // caller checks isEmpty() first
        int front = data[0];
        for (int i = 0; i < tail; ++i) { data[i] = data[i + 1]; ++moves; }
        --tail;                                         // the step that is easy to forget
        return front;
    }
};
```

Kode 5.11 (Lanjutan) *09_linear_queue.cpp: queue bergeser-elemen (benar tetapi O(n) per dequeue) dan queue yang indeksnya bergeser (O(1) tetapi penuh palsu).*

```
// Version 2: head and tail both move right and never come back.
struct DriftQueue {
    int data[CAP];
    int head = 0, tail = -1;
    bool isEmpty() const { return head > tail; }
    bool isFull()  const { return tail == CAP - 1; }   // asks the wrong question
    int  size()    const { return tail - head + 1; }
    bool enqueue(int v) { if (isFull()) return false; data[++tail] = v; return true; }
    int  dequeue()      { return data[head++]; }
};

int main() {
    cout << "--- shifting queue, capacity " << CAP << " ---\n";
    ShiftQueue s;
    for (int v : {4, 45, 1, 8}) {
        s.enqueue(v);
        cout << pad("enqueue " + to_string(v)) << cells(s.data, 0, s.tail) << "  tail=" <<
s.tail << "\n";
    }
    for (int k = 0; k < 2; ++k) {
        long before = s.moves;
        int v = s.dequeue();
        cout << pad("dequeue -> " + to_string(v)) << cells(s.data, 0, s.tail) << "  tail="
<< s.tail
             << "  (" << s.moves - before << " elements shifted)\n";
    }

    cout << "\n--- drifting queue, capacity " << CAP << " ---\n";
    DriftQueue d;
    for (int v = 1; v <= 5; ++v) {
        d.enqueue(v);
        cout << pad("enqueue " + to_string(v)) << cells(d.data, d.head, d.tail)
             << "  head=" << d.head << " tail=" << d.tail << "\n";
    }
    for (int k = 0; k < 2; ++k) {
        int v = d.dequeue();
        cout << pad("dequeue -> " + to_string(v)) << cells(d.data, d.head, d.tail)
             << "  head=" << d.head << " tail=" << d.tail << "\n";
    }
    cout << "size " << d.size() << " of " << CAP << ", but isFull() says "
         << (d.isFull() ? "true" : "false") << "\n";
    cout << "enqueue 6 -> " << (d.enqueue(6) ? "ok" : "REJECTED: false full") << "\n";
    return 0;
}
```

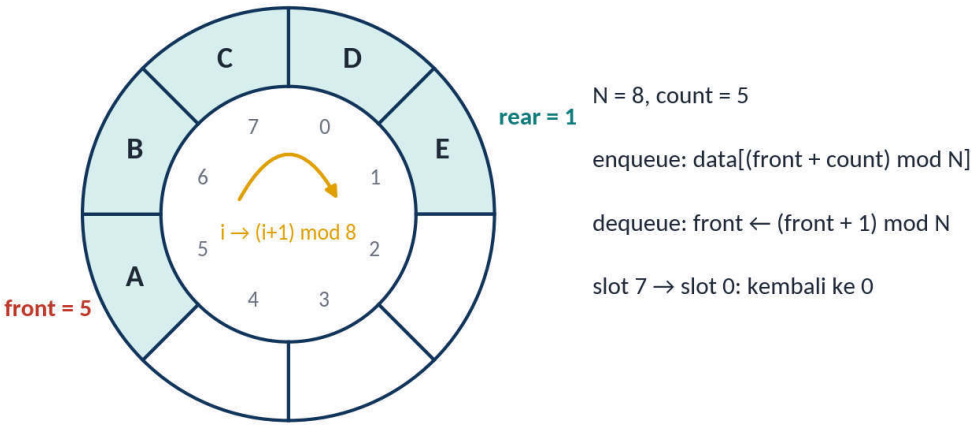
```
--- shifting queue, capacity 5 ---
enqueue 4      [ 4 . . . . ]  tail=0
enqueue 45     [ 4 45 . . . ]  tail=1
enqueue 1      [ 4 45 1 . . ]  tail=2
enqueue 8      [ 4 45 1 8 . ]  tail=3
dequeue -> 4   [ 45 1 8 . . ]  tail=2  (3 elements shifted)
dequeue -> 45  [ 1 8 . . . ]  tail=1  (2 elements shifted)

--- drifting queue, capacity 5 ---
enqueue 1      [ 1 . . . . ]  head=0 tail=0
enqueue 2      [ 1 2 . . . ]  head=0 tail=1
enqueue 3      [ 1 2 3 . . ]  head=0 tail=2
enqueue 4      [ 1 2 3 4 . ]  head=0 tail=3
enqueue 5      [ 1 2 3 4 5 ]  head=0 tail=4
dequeue -> 1   [ . 2 3 4 5 ]  head=1 tail=4
dequeue -> 2   [ . . 3 4 5 ]  head=2 tail=4
size 3 of 5, but isFull() says true
enqueue 6 -> REJECTED: false full
```

Keluaran queue pertama menghitung pekerjaannya: tiga pergeseran untuk *dequeue* pertama, dua untuk yang kedua. Keluaran queue kedua berakhir dengan kontradiksi yang hitam di atas putih: "size 3 of 5, but isFull() says true", disusul *enqueue* yang ditolak. Kita menginginkan operasi $O(1)$ dari percobaan kedua tanpa slot yang hilang. Perbaikannya adalah salah satu gagasan paling elegan dalam buku ini.

5.13 Queue Sirkular

Ketika `tail` mencapai slot terakhir, biarkan ia **melingkar kembali** (*wrap around*) ke slot 0 — yang kemungkinan besar sudah dikosongkan oleh sebuah *dequeue* — lalu lanjutkan. Tekuklah larik menjadi cincin, seperti pada **Gambar 5.13**, sehingga slot $N-1$ disusul slot 0. Indeks tetap hanya bergerak maju, tetapi "maju" kini berputar terus, memakai ulang setiap slot.



Gambar 5.13 Queue sirkular berkapasitas $N = 8$ yang memuat A sampai E. Queue dimulai pada `front` = 5 dan berjalan searah jarum jam melalui slot 6, 7, 0, dan 1: setelah slot 7 ia melingkar ke slot 0.

5.13.1 Aritmetika modular

Alat yang mewujudkan lingkaran itu adalah operator sisa bagi: $a \bmod N$ adalah sisa pembagian a oleh N, selalu berupa bilangan dari 0 sampai $N-1$. Jadi, alih-alih `i = i + 1`, queue sirkular memajukan indeks dengan

$$i \leftarrow (i + 1) \bmod N. \tag{5.6}$$

Untuk $N = 8$, Persamaan (5.6) membawa $0 \to 1 \to \cdots \to 6 \to 7$, lalu $(7 + 1) \bmod 8 = 8 \bmod 8 = 0$. Indeks tidak pernah keluar dari larik, dan tidak diperlukan pernyataan `if` untuk menangkap ujungnya. Dalam C++, Java, dan Python operator sisa bagi ditulis `%`.

5.13.2 Rancangan pertama: front ditambah pencacah

Rancangan yang paling bersih menyimpan `front` (indeks elemen tertua) dan `count` (banyaknya elemen). Bagian belakang tidak disimpan melainkan dihitung. Elemen-elemen menempati `count` slot berturutan mengelilingi cincin mulai dari `front`, sehingga slot **kosong** pertama — tempat *enqueue* berikutnya menulis — adalah

$$\text{free} = (\text{front} + \text{count}) \bmod N, \tag{5.7}$$

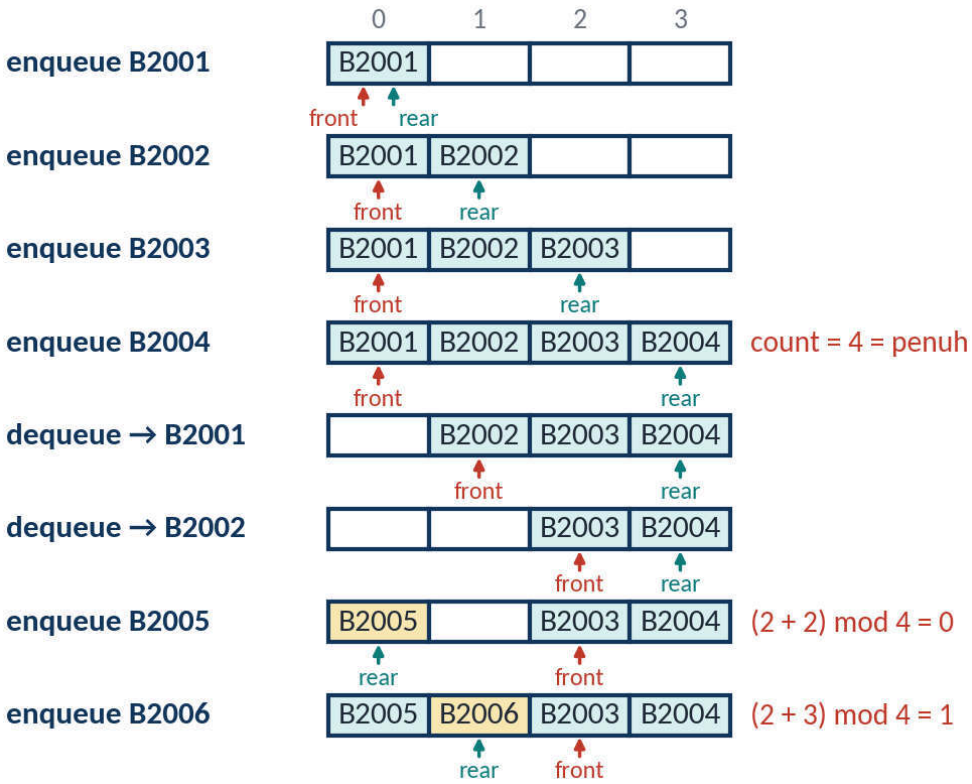
dan elemen terakhir berada satu slot sebelumnya, di $(\text{front} + \text{count} - 1) \bmod N$. Kosong dan penuh menjadi sepele: kosong bila `count = 0`, penuh bila `count = N`. **Pseudokode 5.5** menuliskan kedua operasi utamanya.

Pseudokode 5.5 `ENQUEUE` dan `DEQUEUE` pada larik sirkular berisi *N* slot dengan indeks `front` dan pencacah `count`.

```
ENQUEUE(Q, x)
 1.  if Q.count = N
 2.      error "overflow"
 3.  Q[(Q.front + Q.count) mod N] ← x     // slot kosong pertama, melingkar
 4.  Q.count ← Q.count + 1

DEQUEUE(Q)
 5.  if Q.count = 0
 6.      error "underflow"
 7.  x ← Q[Q.front]
 8.  Q.front ← (Q.front + 1) mod N        // maju dan melingkar
 9.  Q.count ← Q.count - 1
10.  return x
```

Gambar 5.14 memutar ulang meja buku baru dengan kapasitas 4. Buku B2001 sampai B2004 memenuhi larik; B2005 ditolak dan menunggu di meja; dua buku dikatalogkan, yang memajukan `front` ke 2; lalu B2005 dan B2006 tiba dan **melingkar** ke slot 0 dan 1, persis slot yang dikosongkan kedua *dequeue* tadi. Menurut Persamaan (5.7), slot kosong pertama untuk B2005 adalah $(2 + 2) \bmod 4 = 0$, dan untuk B2006 adalah $(2 + 3) \bmod 4 = 1$.



Gambar 5.14 Queue sirkular pada **Kode 5.12** dengan kapasitas 4. Dua *enqueue* terakhir melingkar ke slot 0 dan 1, sehingga bagian belakang kini berada di kiri bagian depan — dan queue tetap berada dalam urutan FIFO yang sempurna.

Perhatikan baris terakhir: urutan fisik dalam larik adalah B2005 B2006 B2003 B2004, tetapi urutan *queue*, dibaca dari `front` mengelilingi cincin, adalah B2003 B2004 B2005 B2006. Lariknya telah dipotong dan disambung kembali, dan queue tidak peduli.

5.13.3 Queue sirkular dalam C++, Java, dan Python

Kode 5.12 mengimplementasikan **Pseudokode 5.5** sebagai *class template*. Metode `slots()` mencetak larik mentahnya, dengan `.....` untuk slot kosong, sehingga kita dapat membandingkan eksekusinya dengan **Gambar 5.14**.

Kode 5.12 `10_circular_queue.cpp`: queue sirkular dengan pencacah, sebagai antrean buku baru Pustaka Ceria.

```
// A circular queue with a count: Pustaka Ceria's new-arrivals cataloging line.
#include <iomanip>
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;

template <typename T>
class CircularQueue {
public:
    explicit CircularQueue(int capacity)
        : data_(new T[capacity]), capacity_(capacity), front_(0), count_(0) {}
    ~CircularQueue() { delete[] data_; }
    CircularQueue(const CircularQueue&) = delete;
    CircularQueue& operator=(const CircularQueue&) = delete;

    bool isEmpty() const { return count_ == 0; }
    bool isFull()  const { return count_ == capacity_; }
    int  size()    const { return count_; }
    int  front()   const { return front_; }
    int  rear()    const { return (front_ + count_ - 1 + capacity_) % capacity_; }

    bool enqueue(const T& value) {
        if (isFull()) return false;
        data_[(front_ + count_) % capacity_] = value;   // the slot just after the rear,
wrapped
        ++count_;
        return true;
    }
    T dequeue() {
        if (isEmpty()) throw underflow_error("dequeue from an empty queue");
        T value = data_[front_];
        front_ = (front_ + 1) % capacity_;              // advance and wrap
        --count_;
        return value;
    }
    const T& peek() const {
        if (isEmpty()) throw underflow_error("peek at an empty queue");
        return data_[front_];
    }
    string slots() const {                              // the raw array, "." for a free
slot
        string s = "[";
        for (int i = 0; i < capacity_; ++i) {
            int offset = (i - front_ + capacity_) % capacity_;   // how far i is behind the
front
            s += " " + (offset < count_ ? data_[i] : string("."));
        }
        return s + " ]";
```

Kode 5.12 (Lanjutan) *10_circular_queue.cpp: queue sirkular dengan pencacah, sebagai antrean buku baru Pustaka Ceria.*

```
    }

private:
    T*  data_;
    int capacity_;
    int front_;                                          // index of the oldest element
    int count_;                                          // how many elements are stored
};

void state(const string& what, const CircularQueue<string>& q) {
    cout << left << setw(26) << what << q.slots() << "  front=" << q.front()
         << " rear=" << q.rear() << " count=" << q.size() << "\n";
}

int main() {
    cout << "=== Pustaka Ceria: new arrivals (circular queue, capacity 4) ===\n";
    CircularQueue<string> q(4);
    for (const string id : {"B2001", "B2002", "B2003", "B2004", "B2005"}) {
        if (q.enqueue(id)) state("enqueue " + id, q);
        else cout << "enqueue " << id << " -> REJECTED, full (it waits at the desk)\n";
    }
    for (int k = 0; k < 2; ++k) { string id = q.dequeue(); state("dequeue -> " + id, q); }
    for (const string id : {"B2005", "B2006"}) { q.enqueue(id); state("enqueue " + id, q);
}
    cout << "peek -> " << q.peek() << "\n";
    cout << "cataloging order:";
    while (!q.isEmpty()) cout << " " << q.dequeue();
    cout << "\n";
    try {
        q.dequeue();
    } catch (const underflow_error& e) {
        cout << "dequeue on the empty queue -> error: " << e.what() << "\n";
    }
    return 0;
}
```

```
=== Pustaka Ceria: new arrivals (circular queue, capacity 4) ===
enqueue B2001              [ B2001 ..... ..... ..... ]  front=0 rear=0 count=1
enqueue B2002              [ B2001 B2002 ..... ..... ]  front=0 rear=1 count=2
enqueue B2003              [ B2001 B2002 B2003 ..... ]  front=0 rear=2 count=3
enqueue B2004              [ B2001 B2002 B2003 B2004 ]  front=0 rear=3 count=4
enqueue B2005 -> REJECTED, full (it waits at the desk)
dequeue -> B2001           [ ..... B2002 B2003 B2004 ]  front=1 rear=3 count=3
dequeue -> B2002           [ ..... ..... B2003 B2004 ]  front=2 rear=3 count=2
enqueue B2005              [ B2005 ..... B2003 B2004 ]  front=2 rear=0 count=3
enqueue B2006              [ B2005 B2006 B2003 B2004 ]  front=2 rear=1 count=4
peek -> B2003
cataloging order: B2003 B2004 B2005 B2006
dequeue on the empty queue -> error: dequeue from an empty queue
```

Eksekusinya cocok dengan gambar baris demi baris, dan baris penutup “cataloging order: B2003 B2004 B2005 B2006” menegaskan bahwa urutan FIFO bertahan melewati lingkaran. Satu baris layak diperhatikan lebih saksama. `rear()` menghitung `(front_ + count_ - 1 + capacity_) % capacity_`. Mengapa ditambah `capacity_`? Karena ketika queue kosong dan `front_` bernilai 0, `front_ + count_ - 1` bernilai −1, dan **dalam C++ dan Java, `-1 % 4` bernilai `-1`**, bukan 3. Kedua bahasa mendefinisikan `%` sehingga hasilnya bertanda sama dengan operan kirinya. Menambahkan *N* lebih dulu menjaga operan kiri tetap nonnegatif dan hasilnya tetap dalam rentang. Python berbeda — `%`-nya selalu menghasilkan nilai bertanda sama dengan pembaginya, sehingga `-1 % 4 == 3` — itulah sebabnya versi Python dapat menghilangkan `+ N`. Melupakan perbedaan ini adalah salah satu *bug* klasik pada *buffer* sirkular; Subbab 5.19 kembali membahasnya.

Kode 5.13 dan **Kode 5.14** adalah versi Java dan Python.

Kode 5.13 `CircularQueueDemo.java`: queue sirkular sebagai kelas generik Java.

```java
// A circular queue with a count: Pustaka Ceria's new-arrivals cataloging line.
public class CircularQueueDemo {
    static class CircularQueue<T> {
        private final T[] data;
        private int front = 0;                      // index of the oldest element
        private int count = 0;                      // how many elements are stored

        @SuppressWarnings("unchecked")
        CircularQueue(int capacity) { data = (T[]) new Object[capacity]; }

        boolean isEmpty() { return count == 0; }
        boolean isFull()  { return count == data.length; }
        int size()        { return count; }
        int front()       { return front; }
        int rear()        { return (front + count - 1 + data.length) % data.length; }

        boolean enqueue(T value) {
            if (isFull()) return false;
            data[(front + count) % data.length] = value;   // the slot just after the rear, wrapped
            ++count;
            return true;
        }
        T dequeue() {
            if (isEmpty()) throw new IllegalStateException("dequeue from an empty queue");
            T value = data[front];
            data[front] = null;
            front = (front + 1) % data.length;      // advance and wrap
            --count;
            return value;
        }
        T peek() {
            if (isEmpty()) throw new IllegalStateException("peek at an empty queue");
            return data[front];
        }
        String slots() {                            // the raw array, "." for a free slot
            StringBuilder s = new StringBuilder("[");
            for (int i = 0; i < data.length; ++i) {
                int offset = (i - front + data.length) % data.length;   // how far i is behind the front
                s.append(' ').append(offset < count ? data[i] : ".....");
            }
            return s.append(" ]").toString();
        }
    }
```

Kode 5.13 (Lanjutan) *CircularQueueDemo.java: queue sirkular sebagai kelas generik Java.*

```
    static void state(String what, CircularQueue<String> q) {
        System.out.printf("%-26s%s  front=%d rear=%d count=%d%n", what, q.slots(),
q.front(), q.rear(), q.size());
    }

    public static void main(String[] args) {
        System.out.println("=== Pustaka Ceria: new arrivals (circular queue, capacity 4)
===");
        CircularQueue<String> q = new CircularQueue<>(4);
        for (String id : new String[]{"B2001", "B2002", "B2003", "B2004", "B2005"}) {
            if (q.enqueue(id)) state("enqueue " + id, q);
            else System.out.println("enqueue " + id + " -> REJECTED, full (it waits at the
desk)");
        }
        for (int k = 0; k < 2; ++k) { String id = q.dequeue(); state("dequeue -> " + id,
q); }
        for (String id : new String[]{"B2005", "B2006"}) { q.enqueue(id); state("enqueue "
+ id, q); }
        System.out.println("peek -> " + q.peek());
        StringBuilder order = new StringBuilder("cataloging order:");
        while (!q.isEmpty()) order.append(' ').append(q.dequeue());
        System.out.println(order);
        try {
            q.dequeue();
        } catch (IllegalStateException e) {
            System.out.println("dequeue on the empty queue -> error: " + e.getMessage());
        }
    }
}
```

Kode 5.14 `10_circular_queue.py`: queue sirkular dalam Python.

```
# A circular queue with a count: Pustaka Ceria's new-arrivals cataloging line.


class CircularQueue:
    def __init__(self, capacity):
        self._data = [None] * capacity
        self._front = 0                  # index of the oldest element
        self._count = 0                  # how many elements are stored

    def is_empty(self):
        return self._count == 0

    def is_full(self):
        return self._count == len(self._data)

    def size(self):
        return self._count

    def front(self):
        return self._front

    def rear(self):
        return (self._front + self._count - 1) % len(self._data)

    def enqueue(self, value):
        if self.is_full():
            return False
        self._data[(self._front + self._count) % len(self._data)] = value   # just after
the rear, wrapped
        self._count += 1
        return True

    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from an empty queue")
        value = self._data[self._front]
        self._data[self._front] = None
        self._front = (self._front + 1) % len(self._data)                   # advance and
wrap
        self._count -= 1
        return value

    def peek(self):
        if self.is_empty():
            raise IndexError("peek at an empty queue")
        return self._data[self._front]
```

Kode 5.14 (Lanjutan) *10_circular_queue.py: queue sirkular dalam Python.*

```
    def slots(self):                    # the raw array, "." for a free slot
        n = len(self._data)
        cells = []
        for i in range(n):
            offset = (i - self._front) % n   # how far i is behind the front
            cells.append(self._data[i] if offset < self._count else ".....")
        return "[ " + " ".join(cells) + " ]"


def state(what, q):
    print(f"{what:<26}{q.slots()}  front={q.front()} rear={q.rear()} count={q.size()}")


def main():
    print("=== Pustaka Ceria: new arrivals (circular queue, capacity 4) ===")
    q = CircularQueue(4)
    for book_id in ("B2001", "B2002", "B2003", "B2004", "B2005"):
        if q.enqueue(book_id):
            state(f"enqueue {book_id}", q)
        else:
            print(f"enqueue {book_id} -> REJECTED, full (it waits at the desk)")
    for _ in range(2):
        book_id = q.dequeue()
        state(f"dequeue -> {book_id}", q)
    for book_id in ("B2005", "B2006"):
        q.enqueue(book_id)
        state(f"enqueue {book_id}", q)
    print(f"peek -> {q.peek()}")
    order = []
    while not q.is_empty():
        order.append(q.dequeue())
    print("cataloging order: " + " ".join(order))
    try:
        q.dequeue()
    except IndexError as e:
        print(f"dequeue on the empty queue -> error: {e}")


if __name__ == "__main__":
    main()
```

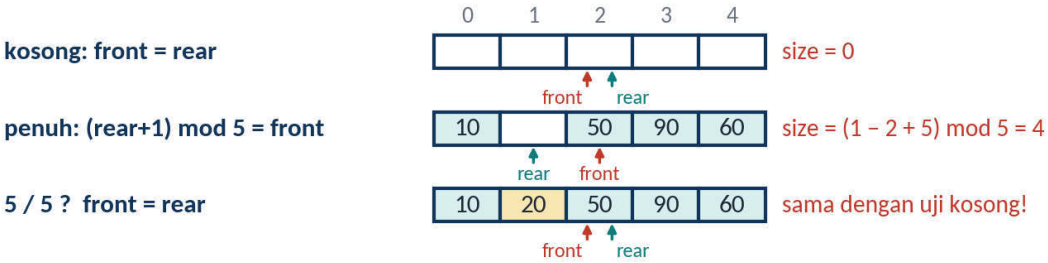
Ketiganya mencetak empat belas baris yang sama.

5.13.4 Rancangan kedua: sisakan satu slot kosong

Banyak buku teks dan banyak kode nyata — termasuk penyajian klasik Cormen dkk. [1] — menyimpan dua indeks tanpa pencacah: `front`, indeks elemen tertua, dan `rear`, indeks **slot kosong berikutnya**. *Enqueue* menulis di `rear` lalu memajukannya dengan Persamaan (5.6), dan *dequeue* membaca di `front` lalu memajukannya. Banyaknya elemen adalah jarak dari `front` ke `rear` mengelilingi cincin:

$$\text{size} = (\text{rear} - \text{front} + N) \bmod N. \tag{5.8}$$

`+ N` pada Persamaan (5.8) lagi-lagi melindungi dari operan kiri negatif ketika `rear` sudah melingkar ke kiri `front`. Namun kini muncul masalah nyata, dan **Gambar 5.15** memperlihatkannya. Queue kosong memiliki `front = rear` (keduanya menunjuk slot kosong yang sama). Isilah seluruh N slot, maka `rear` berkeliling penuh dan mendarat lagi di `front` — sehingga queue yang **penuh** juga memiliki `front = rear`. Kedua indeks tidak dapat membedakan queue penuh dari queue kosong.



Gambar 5.15 Tanpa pencacah, `front = rear` akan berarti "kosong" (atas) sekaligus "kelima slot terpakai" (bawah). Menyisakan satu slot yang selalu kosong (tengah) membuat keadaan penuh dapat dibedakan: queue penuh bila memajukan `rear` akan mencapai `front`.

Obat bakunya adalah **tidak pernah mengisi slot terakhir**. Nyatakan queue penuh bila satu *enqueue* lagi akan membuat `rear` menyusul `front`:

$$\text{full} \Leftrightarrow (\text{rear} + 1) \bmod N = \text{front}, \qquad \text{empty} \Leftrightarrow \text{rear} = \text{front}. \tag{5.9}$$

Dengan Persamaan (5.9), larik berisi N slot memuat paling banyak $N - 1$ elemen. Itulah harganya: satu slot. **Tabel 5.6** membandingkan kedua rancangan.

Tabel 5.6 Dua cara membedakan queue sirkular yang penuh dari yang kosong.

	Rancangan pencacah (Subbab 5.13.2)	Rancangan satu-slot-kosong (Subbab 5.13.4)
Field yang disimpan	`front`, `count`	`front`, `rear`
Makna *rear*	dihitung: (front + count − 1) mod N adalah elemen terakhir	disimpan: `rear` adalah slot kosong berikutnya
Uji kosong	count = 0	rear = front
Uji penuh	count = N	(rear + 1) mod N = front
Ukuran	`count`	(rear − front + N) mod N
Slot terpakai maksimum	N	$N - 1$
Pemakaian umum	kode pembelajaran, sebagian besar queue pustaka	*buffer* tanpa kunci, dengan satu *thread* menulis `rear` dan *thread* lain menulis `front`

Rancangan satu-slot-kosong memiliki keunggulan nyata yang disiratkan baris terakhir tabel: bila satu *thread* hanya melakukan *enqueue* dan *thread* lain hanya melakukan *dequeue*, setiap *thread* hanya menulis indeksnya sendiri, sehingga keduanya tidak pernah menulis variabel yang sama. Dengan `count` bersama, keduanya harus memperbaruinya. Itulah sebabnya *ring buffer* pada *driver* audio, kartu jaringan, dan pengiriman pesan antar-*thread* sering memakai rancangan ini.

Kode 5.15 mengimplementasikannya dan memutar ulang barisan *enqueue* 70/80/50 dari **Gambar 5.10**, kali ini dengan indeks yang sesungguhnya, lalu melanjutkan sampai queue penuh.

Kode 5.15 `11_circular_queue_slot.cpp`: queue sirkular yang menyisakan satu slot kosong dan menghitung ukurannya dari kedua indeks.

```
// A circular queue WITHOUT a count: one slot is always left empty.
#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

const int N = 5;                                        // array length; holds at most N -
1

struct SlotQueue {
    int data[N];
    int front = 0;                                      // index of the oldest element
    int rear  = 0;                                      // index of the NEXT free slot
    bool isEmpty() const { return front == rear; }
    bool isFull()  const { return (rear + 1) % N == front; }
    int  size()    const { return (rear - front + N) % N; }
    bool enqueue(int v) {
        if (isFull()) return false;
        data[rear] = v;
        rear = (rear + 1) % N;
        return true;
    }
    int dequeue() {                                     // caller checks isEmpty() first
        int v = data[front];
        front = (front + 1) % N;
        return v;
    }
};

void state(const string& what, const SlotQueue& q) {
    string s = "[";
    for (int i = 0; i < N; ++i) {
        bool used = (i - q.front + N) % N < q.size();
        s += " " + (used ? to_string(q.data[i]) : string(".."));
    }
    cout << left << setw(14) << what << s << " ]  front=" << q.front << " rear=" << q.rear
         << "  size=(" << q.rear << "-" << q.front << "+" << N << ") mod " << N << " = " <<
q.size()
         << (q.isFull() ? "  FULL" : q.isEmpty() ? "  EMPTY" : "") << "\n";
}
```

Kode 5.15 (Lanjutan) *11_circular_queue_slot.cpp: queue sirkular yang menyisakan satu slot kosong dan menghitung ukurannya dari kedua indeks.*

```
int main() {
    SlotQueue q;
    state("start", q);
    for (int v : {70, 80, 50}) { q.enqueue(v); state("enqueue " + to_string(v), q); }
    for (int k = 0; k < 2; ++k) { int v = q.dequeue(); state("dequeue -> " + to_string(v),
q); }
    for (int v : {90, 60, 10}) { q.enqueue(v); state("enqueue " + to_string(v), q); }
    cout << "enqueue 20    -> " << (q.enqueue(20) ? "ok" : "REJECTED: full with 4 of 5
slots used") << "\n";
    while (!q.isEmpty()) { int v = q.dequeue(); state("dequeue -> " + to_string(v), q); }
    return 0;
}
```

```
start         [ .. .. .. .. .. ]  front=0 rear=0  size=(0-0+5) mod 5 = 0  EMPTY
enqueue 70    [ 70 .. .. .. .. ]  front=0 rear=1  size=(1-0+5) mod 5 = 1
enqueue 80    [ 70 80 .. .. .. ]  front=0 rear=2  size=(2-0+5) mod 5 = 2
enqueue 50    [ 70 80 50 .. .. ]  front=0 rear=3  size=(3-0+5) mod 5 = 3
dequeue -> 70 [ .. 80 50 .. .. ]  front=1 rear=3  size=(3-1+5) mod 5 = 2
dequeue -> 80 [ .. .. 50 .. .. ]  front=2 rear=3  size=(3-2+5) mod 5 = 1
enqueue 90    [ .. .. 50 90 .. ]  front=2 rear=4  size=(4-2+5) mod 5 = 2
enqueue 60    [ .. .. 50 90 60 ]  front=2 rear=0  size=(0-2+5) mod 5 = 3
enqueue 10    [ 10 .. 50 90 60 ]  front=2 rear=1  size=(1-2+5) mod 5 = 4  FULL
enqueue 20    -> REJECTED: full with 4 of 5 slots used
dequeue -> 50 [ 10 .. .. 90 60 ]  front=3 rear=1  size=(1-3+5) mod 5 = 3
dequeue -> 90 [ 10 .. .. .. 60 ]  front=4 rear=1  size=(1-4+5) mod 5 = 2
dequeue -> 60 [ 10 .. .. .. .. ]  front=0 rear=1  size=(1-0+5) mod 5 = 1
dequeue -> 10 [ .. .. .. .. .. ]  front=1 rear=1  size=(1-1+5) mod 5 = 0  EMPTY
```

Setiap baris mencetak Persamaan (5.8) dengan angka sesungguhnya, sehingga Anda dapat memeriksa aritmetikanya: setelah `enqueue 60`, `rear` sudah melingkar ke 0 dan ukurannya $(0 - 2 + 5) \bmod 5 = 3$. Setelah `enqueue 10` ukurannya 4 dan $(1 + 1) \bmod 5 = 2 = \text{front}$, sehingga Persamaan (5.9) melaporkan FULL, dan `enqueue 20` ditolak walaupun slot 1 masih kosong — kali ini memang disengaja. Empat *dequeue* kemudian mengosongkan queue dalam urutan FIFO, 50, 90, 60, 10, dan baris terakhir memperlihatkan `front = rear = 1`: kosong.

5.14 Deque

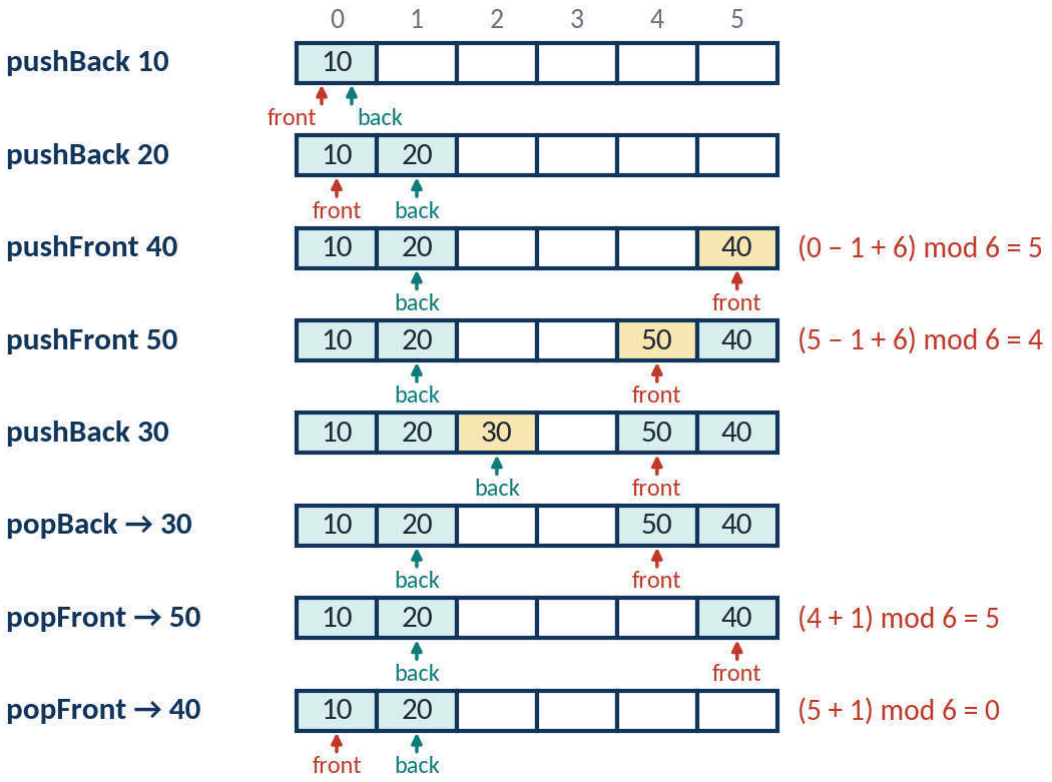
Stack bekerja di satu ujung; queue menambah di satu ujung dan mengambil di ujung lain. Perumuman yang wajar mengizinkan penambahan **dan** pengambilan di **kedua** ujung. Struktur ini disebut *double-ended queue*, disingkat *deque* dan dilafalkan seperti "dek"; Knuth menyebut E. J. Schweppe sebagai pencetus namanya [2].

> **DEFINISI.** *Deque* mendukung `pushFront`, `pushBack`, `popFront`, `popBack`, `peekFront`, dan `peekBack`, beserta `isEmpty` dan `size`. Bila hanya dipakai dengan `pushBack` dan `popBack`, ia adalah stack; bila hanya dipakai dengan `pushBack` dan `popFront`, ia adalah queue.

Larik sirkular mengimplementasikan deque dengan satu gagasan baru: `pushFront` harus memindahkan `front` satu slot **mundur**, melingkar dari slot 0 ke slot $N - 1$. Mundur berarti mengurangi 1, dan agar operan kiri `%` tetap nonnegatif dalam C++ dan Java, kita tambahkan N lebih dulu:

$$\text{front} \leftarrow (\text{front} - 1 + N) \bmod N. \tag{5.10}$$

`popBack` tidak memerlukan perubahan indeks selain mengurangi `count`, karena posisi elemen belakang, $(\text{front} + \text{count} - 1) \bmod N$, dihitung dari `count`. **Gambar 5.16** menelusuri deque berkapasitas 6.



Gambar 5.16 Deque berbasis larik sirkular berkapasitas 6. `pushFront` memindahkan `front` mundur dengan pelingkaran, dari 0 ke 5 lalu ke 4; `popFront` memajukannya kembali.

`pushFront` pertama adalah yang menarik: dengan `front = 0`, Persamaan (5.10) memberikan $(0 - 1 + 6) \bmod 6 = 5$, sehingga 40 mendarat di slot *terakhir* larik, yang secara logis berada di depan 10. **Kode 5.16** mengimplementasikan deque dan memakainya untuk tugas klasik: menguji apakah sebuah frasa adalah palindrom dengan berulang kali membandingkan lalu membuang kedua ujungnya.

Kode 5.16 `12_array_deque.cpp`: deque berbasis larik sirkular, dan uji palindrom yang mengambil dari kedua ujung.

```
// A double-ended queue on a circular array, and a palindrome test that uses it.
#include <cctype>
#include <iomanip>
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;

template <typename T>
class ArrayDeque {
public:
    explicit ArrayDeque(int capacity)
        : data_(new T[capacity]), cap_(capacity), front_(0), count_(0) {}
    ~ArrayDeque() { delete[] data_; }
    ArrayDeque(const ArrayDeque&) = delete;
    ArrayDeque& operator=(const ArrayDeque&) = delete;

    bool isEmpty() const { return count_ == 0; }
    bool isFull()  const { return count_ == cap_; }
    int  size()    const { return count_; }
    int  front()   const { return front_; }

    void pushFront(const T& v) {
        if (isFull()) throw overflow_error("deque is full");
        front_ = (front_ - 1 + cap_) % cap_;           // step back one slot, wrapping to
the end
        data_[front_] = v;
        ++count_;
    }
    void pushBack(const T& v) {
        if (isFull()) throw overflow_error("deque is full");
        data_[(front_ + count_) % cap_] = v;
        ++count_;
    }
    T popFront() {
        if (isEmpty()) throw underflow_error("deque is empty");
        T v = data_[front_];
        front_ = (front_ + 1) % cap_;
        --count_;
        return v;
    }
    T popBack() {
        if (isEmpty()) throw underflow_error("deque is empty");
        --count_;
        return data_[(front_ + count_) % cap_];         // the old back slot
    }
    string slots() const {
```

Kode 5.16 (Lanjutan) *12_array_deque.cpp: deque berbasis larik sirkular, dan uji palindrom yang mengambil dari kedua ujung.*

```
        string s = "[";
        for (int i = 0; i < cap_; ++i) {
            bool used = (i - front_ + cap_) % cap_ < count_;
            s += " " + (used ? to_string(data_[i]) : string(".."));
        }
        return s + " ]";
    }

private:
    T*  data_;
    int cap_, front_, count_;
};

bool isPalindrome(const string& text) {
    ArrayDeque<char> d(64);
    for (char c : text)
        if (isalpha(static_cast<unsigned char>(c)))
            d.pushBack(static_cast<char>(tolower(static_cast<unsigned char>(c))));
    while (d.size() > 1)
        if (d.popFront() != d.popBack()) return false;  // compare the two ends, then
discard them
    return true;
}

int main() {
    ArrayDeque<int> d(6);
    auto show = [&](const string& what) {
        cout << left << setw(16) << what << d.slots() << "  front=" << d.front()
             << " size=" << d.size() << "\n";
    };
    show("start");
    d.pushBack(10);  show("pushBack 10");
    d.pushBack(20);  show("pushBack 20");
    d.pushFront(40); show("pushFront 40");
    d.pushFront(50); show("pushFront 50");
    d.pushBack(30);  show("pushBack 30");
    int b = d.popBack();  show("popBack -> " + to_string(b));
    int f = d.popFront(); show("popFront -> " + to_string(f));
    f = d.popFront();     show("popFront -> " + to_string(f));

    cout << "\n";
    for (const string t : {"kasur rusak", "Taat", "racecar", "stack", "A man, a plan, a
canal: Panama"})
        cout << left << setw(34) << ("\"" + t + "\"") << (isPalindrome(t) ? "palindrome" :
"not a palindrome") << "\n";
    return 0;
}
```

```
start           [ .. .. .. .. .. .. ]  front=0 size=0
pushBack 10     [ 10 .. .. .. .. .. ]  front=0 size=1
pushBack 20     [ 10 20 .. .. .. .. ]  front=0 size=2
pushFront 40    [ 10 20 .. .. .. 40 ]  front=5 size=3
pushFront 50    [ 10 20 .. .. 50 40 ]  front=4 size=4
pushBack 30     [ 10 20 30 .. 50 40 ]  front=4 size=5
popBack -> 30   [ 10 20 .. .. 50 40 ]  front=4 size=4
popFront -> 50  [ 10 20 .. .. .. 40 ]  front=5 size=3
popFront -> 40  [ 10 20 .. .. .. .. ]  front=0 size=2

"kasur rusak"                     palindrome
"Taat"                            palindrome
"racecar"                         palindrome
"stack"                           not a palindrome
"A man, a plan, a canal: Panama"  palindrome
```

Uji palindrom mengabaikan spasi, tanda baca, dan huruf besar-kecil, sehingga "kasur rusak" (palindrom bahasa Indonesia yang terkenal) dan "A man, a plan, a canal: Panama" sama-sama lolos, sedangkan "stack" gagal pada perbandingan pertama antara `s` dan `k`. Setiap karakter di-*push* sekali dan di-*pop* sekali, sehingga uji ini berjalan dalam $O(n)$.

Deque bersinar setiap kali pekerjaan datang di satu ujung dan diambil dari kedua ujung. Penjadwal *work-stealing* memberi setiap inti prosesor sebuah deque berisi tugas: inti itu melakukan *push* dan *pop* tugasnya sendiri di belakang (seperti stack, demi lokalitas *cache*), sedangkan inti yang menganggur mencuri dari depan. Masalah *sliding-window maximum* — nilai terbesar pada setiap jendela k pembacaan berturutan, misalnya untuk melaporkan lalu lintas puncak dalam setiap jendela lima menit — diselesaikan dalam $O(n)$ dengan deque berisi indeks kandidat (Soal 5.14).

5.15 Priority Queue: Sekilas

Unit gawat darurat rumah sakit tidak melayani pasien menurut urutan kedatangan. Pasien serangan jantung yang tiba semenit lalu ditangani sebelum pasien keseleo yang sudah menunggu sejam. Setiap pasien memiliki **prioritas**, dan yang dilayani berikutnya adalah yang prioritasnya tertinggi, kapan pun ia tiba. Inilah *priority queue* (antrean prioritas): ADT dengan `insert(x, priority)`, `extractMax()` (atau `extractMin()`), `peek`, dan `isEmpty`.

Meski namanya demikian, priority queue bukanlah queue dalam arti FIFO: urutan kedatangan hanya penting untuk memecah seri, itu pun jika ada. Pertanyaan menariknya adalah bagaimana membuat kedua operasi cepat. **Tabel 5.7** membandingkan implementasi yang paling jelas dengan yang dibangun di Bab 14.

Tabel 5.7 Tiga cara mengimplementasikan priority queue. *Heap* pada Bab 14 adalah satu-satunya yang membuat kedua operasi cepat.

Implementasi	`insert`	`extractMax`	Gagasan
Larik tak terurut	$O(1)$	$O(n)$	tambahkan di akhir; pindai untuk mencari maksimum
Larik terurut	$O(n)$	$O(1)$	geser untuk menjaga urutan; maksimum ada di akhir
Binary heap (Bab 14)	$O(\lg n)$	$O(\lg n)$	pohon yang tersembunyi dalam larik

Tidak satu pun versi naif memuaskan: salah satu dari dua operasinya linear. *Binary heap* menyeimbangkan keduanya pada $O(\lg n)$ masing-masing, yang untuk sejuta elemen berarti sekitar 20 langkah alih-alih sejuta. Ketiga pustaka standar menyediakan priority queue berbasis *heap*, dan Subbab 5.16 memperlihatkan cara memakainya sekarang, sebelum Bab 14 memperlihatkan cara kerjanya.

5.16 Stack dan Queue dalam Pustaka Standar

Setelah membangun setiap struktur dengan tangan, Anda berhak memakai versi pustakanya — dan Anda akan jauh lebih sering memakainya daripada buatan sendiri. Setiap bahasa menyediakannya dengan cara yang sedikit berbeda, dan masing-masing memiliki satu-dua jebakan.

5.16.1 C++: *container adaptor*

Dalam C++, `std::stack`, `std::queue`, dan `std::priority_queue` adalah *container adaptor*: pembungkus tipis yang mengambil sebuah kontainer dasar dan hanya memperlihatkan operasi

stack, queue, atau priority queue. Secara bawaan `std::stack` dan `std::queue` membungkus `std::deque`, dan `std::priority_queue` membungkus `std::vector` yang diatur sebagai *max-heap* [8], [9]. `std::deque` sendiri adalah *double-ended queue* utuh dengan penyisipan dan pengambilan $O(1)$ di kedua ujung. **Kode 5.17** mencoba keempatnya.

Kode 5.17 `13_library.cpp`: `std::stack`, `std::queue`, `std::deque`, dan `std::priority_queue`.

```cpp
// The standard library's ready-made stack, queue, deque and priority queue.
#include <deque>
#include <iostream>
#include <queue>
#include <stack>
#include <vector>
using namespace std;

int main() {
    stack<int> s;                                   // adapter over deque<int> by
default
    for (int v : {1, 2, 3}) s.push(v);
    cout << "stack  : push 1 2 3 -> top " << s.top() << ", pop order:";
    while (!s.empty()) { cout << " " << s.top(); s.pop(); }   // pop() returns nothing
    cout << "\n";

    queue<int> q;                                   // adapter over deque<int> by
default
    for (int v : {1, 2, 3}) q.push(v);              // push = enqueue at the back
    cout << "queue  : push 1 2 3 -> front " << q.front() << ", back " << q.back() << ", pop
order:";
    while (!q.empty()) { cout << " " << q.front(); q.pop(); }
    cout << "\n";

    deque<int> d;
    d.push_back(1); d.push_back(2); d.push_front(0); d.push_front(-1);
    cout << "deque  : front to back:";
    for (int v : d) cout << " " << v;
    d.pop_front(); d.pop_back();
    cout << " | after pop_front, pop_back:";
    for (int v : d) cout << " " << v;
    cout << "\n";

    priority_queue<int> pq;                         // a max-heap underneath (Chapter
14)
    for (int v : {5, 1, 4, 2, 3}) pq.push(v);
    cout << "pqueue : push 5 1 4 2 3 -> pop order:";
    while (!pq.empty()) { cout << " " << pq.top(); pq.pop(); }
    cout << "\n";

    priority_queue<int, vector<int>, greater<int>> minpq;   // smallest first
    for (int v : {5, 1, 4, 2, 3}) minpq.push(v);
    cout << "min-pq : push 5 1 4 2 3 -> pop order:";
    while (!minpq.empty()) { cout << " " << minpq.top(); minpq.pop(); }
    cout << "\n";
    return 0;
}
```

```
stack  : push 1 2 3 -> top 3, pop order: 3 2 1
queue  : push 1 2 3 -> front 1, back 3, pop order: 1 2 3
deque  : front to back: -1 0 1 2 | after pop_front, pop_back: 0 1
pqueue : push 5 1 4 2 3 -> pop order: 5 4 3 2 1
min-pq : push 5 1 4 2 3 -> pop order: 1 2 3 4 5
```

Tiga fakta tentang *adaptor* C++ mengejutkan hampir semua orang pada kali pertama:

- **`pop()` mengembalikan `void`.** Untuk mengambil dan memakai elemen puncak Anda menulis `x = s.top(); s.pop();`. Pustaka standar dirancang begini agar `pop` tidak pernah dapat menghilangkan elemen bila penyalinannya melempar eksepsi di tengah jalan.
- **`top()`, `front()`, dan `pop()` pada kontainer kosong adalah perilaku tak terdefinisi**, bukan eksepsi. Pustaka menganggap Anda sudah memeriksa `empty()`. Kelas buatan kita pada Subbab 5.4 dan 5.13 lebih ketat.
- **`std::priority_queue` secara bawaan adalah *max-heap*.** Untuk urutan terkecil-dahulu, berikan `std::greater<int>` sebagai pembanding, seperti pada baris `min-pq`.

5.16.2 Java: `ArrayDeque` untuk keduanya

Dalam Java, pakailah `ArrayDeque` sebagai stack sekaligus queue. Dokumentasi resminya menyatakan bahwa kelas ini "is likely to be faster than `Stack` when used as a stack, and faster than `LinkedList` when used as a queue" [10]. Kelas lama `java.util.Stack` merupakan turunan `Vector`, yang seluruh metodenya tersinkronisasi dan yang memungkinkan penyisipan ke tengah "stack"; dokumentasinya sendiri menyarankan antarmuka `Deque` "in preference to this class" [11]. **Kode 5.18** menghasilkan keluaran yang sama dengan **Kode 5.17**.

Kode 5.18 `Library.java`: `ArrayDeque` sebagai stack, sebagai queue, dan sebagai deque, serta `PriorityQueue` dalam kedua urutan.

```java
// The standard library's ready-made stack, queue, deque and priority queue.
import java.util.ArrayDeque;
import java.util.Collections;
import java.util.PriorityQueue;

public class Library {
    public static void main(String[] args) {
        ArrayDeque<Integer> s = new ArrayDeque<>();     // used as a stack: push/pop/peek
at the head
        for (int v : new int[]{1, 2, 3}) s.push(v);
        StringBuilder line = new StringBuilder("stack  : push 1 2 3 -> top " + s.peek() +
", pop order:");
        while (!s.isEmpty()) line.append(' ').append(s.pop());
        System.out.println(line);

        ArrayDeque<Integer> q = new ArrayDeque<>();     // used as a queue: offer at the
tail, poll at the head
        for (int v : new int[]{1, 2, 3}) q.offer(v);
        line = new StringBuilder("queue  : push 1 2 3 -> front " + q.peek() + ", back " +
q.peekLast() + ", pop order:");
        while (!q.isEmpty()) line.append(' ').append(q.poll());
        System.out.println(line);

        ArrayDeque<Integer> d = new ArrayDeque<>();
        d.addLast(1); d.addLast(2); d.addFirst(0); d.addFirst(-1);
        line = new StringBuilder("deque  : front to back:");
        for (int v : d) line.append(' ').append(v);
        d.pollFirst(); d.pollLast();
        line.append(" | after pop_front, pop_back:");
        for (int v : d) line.append(' ').append(v);
        System.out.println(line);

        PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());   //
largest first
        for (int v : new int[]{5, 1, 4, 2, 3}) pq.add(v);
        line = new StringBuilder("pqueue : push 5 1 4 2 3 -> pop order:");
        while (!pq.isEmpty()) line.append(' ').append(pq.poll());
        System.out.println(line);

        PriorityQueue<Integer> minpq = new PriorityQueue<>();                           //
smallest first
        for (int v : new int[]{5, 1, 4, 2, 3}) minpq.add(v);
        line = new StringBuilder("min-pq : push 5 1 4 2 3 -> pop order:");
        while (!minpq.isEmpty()) line.append(' ').append(minpq.poll());
        System.out.println(line);
    }
}
```

Dua rincian penting. Sebagai stack, `ArrayDeque` melakukan *push* dan *pop* di **kepalanya** — sehingga iterasi atasnya berjalan dari puncak ke dasar — sedangkan sebagai queue ia melakukan *offer* di ekor dan *poll* di kepala. Selain itu `ArrayDeque` menolak elemen `null`, sehingga `poll()` yang mengembalikan `null` dapat berarti "kosong" tanpa keraguan. `PriorityQueue` di Java secara bawaan adalah *min-heap*; `Collections.reverseOrder()` membaliknya.

5.16.3 Python: `list`, `collections.deque`, `queue`, dan `heapq`

Python tidak memiliki kelas bernama `Stack`. Sebuah `list` adalah stack: `append` melakukan *push* dan `pop()` melakukan *pop*, keduanya di akhir *list* dan keduanya $O(1)$ teramortisasi. Untuk queue, **jangan** memakai `list.pop(0)`: mengambil elemen pertama menggeser setiap elemen lain ke kiri, persis seperti queue bergeser pada Subbab 5.12.1, dan dokumentasi Python memperingatkan bahwa `pop(0)` dan `insert(0, v)` menimbulkan "O(n) memory movement costs". Pakailah `collections.deque`, yang mendukung penambahan dan pengambilan "from either side of the

deque with approximately the same O(1) performance in either direction" [12]. Untuk queue yang dipakai bersama antar-*thread*, modul `queue` menyediakan `Queue`, `LifoQueue`, dan `PriorityQueue` dengan "all the required locking semantics" [13]; dan modul `heapq` mengubah *list* menjadi *binary min-heap*. **Kode 5.19** adalah versi Python dari tur pustaka ini.

Kode 5.19 `13_library.py`: *list* sebagai stack, `collections.deque` sebagai queue dan deque, serta `heapq` sebagai priority queue.

```
# The standard library's ready-made stack, queue, deque and priority queue.
import heapq
from collections import deque


def main():
    s = []                                  # a list used as a stack: append/pop at the end
    for v in (1, 2, 3):
        s.append(v)
    out = []
    top = s[-1]
    while s:
        out.append(s.pop())
    print(f"stack  : push 1 2 3 -> top {top}, pop order: " + " ".join(map(str, out)))

    q = deque()                             # a deque used as a queue: append right, popleft
left
    for v in (1, 2, 3):
        q.append(v)
    front, back = q[0], q[-1]
    out = []
    while q:
        out.append(q.popleft())
    print(f"queue  : push 1 2 3 -> front {front}, back {back}, pop order: " + "
".join(map(str, out)))

    d = deque()
    d.append(1)
    d.append(2)
    d.appendleft(0)
    d.appendleft(-1)
    line = "deque  : front to back: " + " ".join(map(str, d))
    d.popleft()
    d.pop()
    print(line + " | after pop_front, pop_back: " + " ".join(map(str, d)))

    pq = []                                 # heapq is a min-heap: store -v to get largest
first
    for v in (5, 1, 4, 2, 3):
        heapq.heappush(pq, -v)
    out = []
    while pq:
        out.append(-heapq.heappop(pq))
    print("pqueue : push 5 1 4 2 3 -> pop order: " + " ".join(map(str, out)))
```

Kode 5.19 (Lanjutan) *13_library.py: list sebagai stack, collections.deque sebagai queue dan deque, serta heapq sebagai priority queue.*

```
    minpq = []                              # smallest first, heapq's natural order
    for v in (5, 1, 4, 2, 3):
        heapq.heappush(minpq, v)
    out = []
    while minpq:
        out.append(heapq.heappop(minpq))
    print("min-pq : push 5 1 4 2 3 -> pop order: " + " ".join(map(str, out)))


if __name__ == "__main__":
    main()
```

Karena `heapq` hanya menyediakan *min-heap*, baris terbesar-dahulu melakukan *push* $-v$ dan menegasikannya kembali saat *pop* — idiom baku. Ketiga program mencetak lima baris yang ditampilkan setelah **Kode 5.17**.

Tabel 5.8 menghimpun kosakatanya.

Tabel 5.8 Operasi yang sama dalam ketiga pustaka standar.

Operasi	C++	Java	Python
stack push / pop / top	`s.push(x)` / `s.pop()` / `s.top()`	`d.push(x)` / `d.pop()` / `d.peek()`	`l.append(x)` / `l.pop()` / `l[-1]`
queue enqueue / dequeue / front	`q.push(x)` / `q.pop()` / `q.front()`	`d.offer(x)` / `d.poll()` / `d.peek()`	`d.append(x)` / `d.popleft()` / `d[0]`
deque kedua ujung	`push_front`, `push_back`, `pop_front`, `pop_back`	`addFirst`, `addLast`, `pollFirst`, `pollLast`	`appendleft`, `append`, `popleft`, `pop`
priority queue	`std::priority_queue` (maksimum dahulu)	`PriorityQueue` (minimum dahulu)	`heapq` pada *list* (minimum dahulu)
uji kosong	`s.empty()`	`d.isEmpty()`	`not l` atau `len(l) == 0`
pop saat kosong	perilaku tak terdefinisi	`pop()` melempar eksepsi, `poll()` mengembalikan `null`	`IndexError`
queue aman-*thread*	tidak ada di pustaka standar	`java.util.concurrent.ArrayBlockingQueue`	`queue.Queue`

5.17 Memilih di Antara Mereka: Ringkasan Biaya

Tabel 5.9 menghimpun biaya setiap struktur dalam bab ini. “Teramortisasi” berarti dirata-ratakan atas barisan operasi yang panjang, sebagaimana diturunkan dalam Persamaan (5.5).

Tabel 5.9 Biaya setiap struktur dalam bab ini.

Struktur	Tambah	Ambil	Intip	Ruang untuk n butir	Kapasitas tetap?
Stack larik (Subbab 5.4)	$O(1)$	$O(1)$	$O(1)$	N slot	ya
Stack larik berlipat dua (Subbab 5.5.1)	$O(1)$ teramortisasi, $O(n)$ terburuk	$O(1)$	$O(1)$	paling banyak $2n$ slot	tidak
Stack bertaut (Subbab 5.5.2)	$O(1)$	$O(1)$	$O(1)$	n simpul + n pointer	tidak
Queue linear bergeser-elemen (Subbab 5.12.1)	$O(1)$	$O(n)$	$O(1)$	N slot	ya
Queue linear yang indeksnya bergeser (Subbab 5.12.2)	$O(1)$	$O(1)$	$O(1)$	slot hilang selamanya	ya, dan menyusut
Queue sirkular (Subbab 5.13)	$O(1)$	$O(1)$	$O(1)$	N (atau $N-1$) slot	ya
Deque sirkular (Subbab 5.14)	$O(1)$ di kedua ujung	$O(1)$ di kedua ujung	$O(1)$	N slot	ya
Priority queue, *binary heap* (Bab 14)	$O(\lg n)$	$O(\lg n)$	$O(1)$	n slot	tidak

Tabel ini menuntun ke aturan keputusan yang ringkas. Jika masalahnya memerlukan "yang terbaru lebih dulu" — *undo*, *backtracking*, struktur bersarang, pemanggilan fungsi — pakailah stack. Jika memerlukan "yang tertua lebih dulu" — keadilan, *buffering*, penelusuran tingkat demi tingkat — pakailah queue, dan implementasikan sebagai *buffer* sirkular, jangan pernah sebagai queue yang bergeser. Jika memerlukan kedua ujung, pakailah deque. Jika memerlukan "yang terpenting lebih dulu", pakailah priority queue. Dan dalam kode produksi, pakailah versi pustaka kecuali Anda sudah mengukur alasan untuk tidak melakukannya.

5.18 Profitina dalam Praktik

Profitina adalah platform AI Visibility (GEO) yang berjalan di produksi di app.profitina.com: ia memindai situs web sebuah merek, mengukur seberapa siap situs itu dikutip oleh mesin penjawab berbasis AI, dan membantu memperbaiki temuannya [14]. Produk ini ditulis dalam Python di atas FastAPI, menjalankan pekerjaan AI-nya pada model Qwen2.5 lokal, dan menyajikan umpan komunitas waktu-nyata melalui Server-Sent Events. Tidak ada bagian darinya yang merupakan soal buku teks, tetapi struktur-struktur dalam bab ini muncul di mana-mana, karena masalah yang dipecahkannya memiliki bentuk yang sama dengan yang telah kita pelajari. Berikut uraian bentuk-bentuk itu dan golongan solusi yang dituntut masing-masing.

Data terstruktur harus seimbang. Mesin penjawab membaca data terstruktur schema.org, yang disisipkan situs sebagai blok JSON-LD. Satu `}` yang hilang dalam blok itu membuat seluruh blok tak terbaca, dan data terstruktur situs itu diam-diam lenyap dari pandangan mesin. Hal pertama yang dilakukan setiap pengurai JSON adalah pemeriksaan **Pseudokode 5.2**: setiap `{` dan `[` harus ditutup oleh pasangannya, dalam urutan yang benar. Fitur bantuan teknis yang menolong pemilik situs menulis data terstruktur harus melaporkan galat semacam itu setepat **Kode 5.7** — tanda kurung mana, di posisi mana, gagal cocok dengan apa.

Perayapan adalah queue. Untuk memindai sebuah situs, perayap (*crawler*) mulai dari halaman beranda, menemukan tautan di dalamnya, dan harus memutuskan halaman mana yang diambil berikutnya. Menyimpan halaman yang sudah ditemukan tetapi belum dikunjungi dalam queue FIFO akan mengambilnya tingkat demi tingkat — beranda, lalu semua yang berjarak satu klik, lalu semua yang berjarak dua klik — yang kemungkinan besar juga urutan nilai halaman-halaman itu

bagi perayap mesin penjawab sendiri. Inilah *breadth-first search*, yang dibangun di atas graf pada Bab 13; queue di jantungnya adalah queue sirkular Subbab 5.13 atau deque pustaka.

Pekerjaan lambat masuk ke antrean kerja. Menjalankan model bertujuh miliar parameter secara lokal lambat dibandingkan dengan menjawab permintaan web. Pekerjaan seperti menyusun konten GEO atau memberi pendapat AI opsional tentang tautan yang diunggah tidak dapat dikerjakan sementara permintaan pengguna menunggu; pekerjaan itu dimasukkan ke dalam queue dan diambil oleh sebuah *worker* menurut urutan kedatangan. Disiplin FIFO-lah yang membuatnya adil — tidak ada permintaan yang disalip permintaan yang datang belakangan — dan batas kapasitas queue-lah yang melindungi server: ketika queue penuh, aplikasi harus mengatakannya alih-alih menerima pekerjaan yang tidak sanggup diselesaikannya, pilihan jujur yang sama dengan yang dibuat `push` dan `enqueue` kita ketika mengembalikan `false`.

Pemindaian ulang terjadwal adalah priority queue. Setiap merek dipindai ulang secara otomatis dengan irama yang ditentukan tingkat langganannya. Pertanyaan penjadwal selalu "pemindaian mana yang paling cepat jatuh tempo?", dan jawabannya adalah minimum dari himpunan waktu jatuh tempo yang terus berubah — priority queue berkunci waktu, struktur yang diperkenalkan sekilas di Subbab 5.15 dan dibangun di Bab 14. Queue FIFO biasa keliru di sini: pemindaian mingguan yang diantrekan lebih awal tidak boleh menunda pemindaian harian yang jatuh temponya lebih dulu.

Umpan langsung memerlukan *buffer* berbatas. Umpan komunitas waktu-nyata mengirimkan setiap unggahan baru ke setiap peramban yang terhubung. Jika koneksi seorang pembaca lambat, unggahan yang menunggu dikirim kepadanya harus ditampung di suatu tempat — dan ditampung dalam *buffer* berukuran *terbatas*, atau satu ponsel yang lambat dapat membuat server kehabisan memori. *Ring buffer* dengan aturan penuh/kosong Persamaan (5.9) adalah struktur klasik untuk situasi persis ini, dan ia juga menyisakan satu keputusan kebijakan yang jelas bagi server ketika *buffer* penuh: membuang butir tertua yang belum terkirim, atau meminta pembaca menyambung ulang lalu menyusul.

Unggahan optimistis adalah *undo* satu tingkat. Ketika pengguna mengunggah ke komunitas, kotak tulisnya langsung dikosongkan dan unggahannya langsung tampil, sebelum server mengonfirmasinya; jika pengiriman gagal, teksnya dikembalikan. Itulah pola *undo* Subbab 5.7 dengan stack sedalam satu: teks disimpan sebelum perubahan optimistis, lalu di-*pop* kembali jika perubahan itu harus dibatalkan.

Alur umpan balik adalah queue dari queue. Saran seorang pengguna melewati tahap kirim, kurasi, dataset, latih-dan-ukur, dan rilis. Di dalam setiap tahap, butir ditangani menurut urutan kedatangannya, dan tidak ada butir yang melompati tahap — barisan queue FIFO yang keluaran satu tahapnya menjadi masukan tahap berikutnya.

Latihan praktik yang dapat dijalankan untuk gagasan-gagasan ini memakai studi kasus pembelajaran Profitina Laundry: Soal 5.15 meminta Anda merancang meja pesanannya.

5.19 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. **Menaikkan indeks dan menulis dalam urutan yang salah.** `data[top++] = x` dengan *top* bermula dari −1 menulis ke `data[-1]`. Dengan konvensi "−1 berarti kosong", *push* adalah `data[++top] = x` dan *pop* adalah `return data[top--]`. Tulislah kedua baris **Pseudokode 5.1** itu sekali, uji pada stack berkapasitas 1, lalu salin dari sana seterusnya.

2. **Mencampur konvensi `top`.** Menginisialisasi `top = 0` lalu menguji kekosongan dengan `top == -1` (atau sebaliknya) menghasilkan stack yang tidak pernah kosong atau tidak pernah penuh. Pilih satu konvensi, tuliskan Persamaan (5.4) sebagai komentar di samping field-nya, dan patuhi.
3. ***Pop* tanpa memeriksa.** `std::stack::top()` dan `pop()` pada stack kosong adalah perilaku tak terdefinisi dalam C++; programnya bisa *crash*, atau lebih buruk, tampak bekerja. Periksa `empty()` lebih dulu, atau bungkus stack pustaka dalam fungsi yang memeriksanya untuk Anda (Soal 5.18).
4. **Menukar operan saat mengevaluasi postfiks.** *Pop* pertama adalah operan *kanan*. `8 3 -` adalah $8 - 3$, bukan $3 - 8$. Selalu uji evaluator pada pengurangan dan pembagian, jangan hanya pada `+` dan `*`.
5. **Memperlakukan `^` sebagai asosiatif-kiri dalam algoritma *shunting-yard*.** Mengeluarkan operator berpresedensi sama untuk `^` mengubah `2^3^2` menjadi $(2^3)^2 = 64$ alih-alih 512. *Pop* pada presedensi sama di baris 12 **Pseudokode 5.3** hanya berlaku untuk operator asosiatif-kiri.
6. **Melupakan pembatas tanda kurung.** Operator yang datang tidak boleh mengeluarkan `(`; tanda `)` harus mengeluarkan operator *sampai* `(`-nya lalu membuangnya. Pemeriksaan yang hilang mengubah `(3+2)*5` menjadi `3 2 5 * +`.
7. **Melupakan `tail--` pada queue bergeser-elemen**, dan mencetak dengan `i < tail` alih-alih `i <= tail`. Yang pertama menduplikasi elemen terakhir; yang kedua menyembunyikannya. Telusuri satu *dequeue* dengan tangan, seperti pada **Gambar 5.11**.
8. **Merilis queue yang indeksnya bergeser.** Queue yang `head` dan `tail`-nya hanya bertambah akan menyatakan dirinya penuh setelah `capacity` kali *enqueue*, berapa pun *dequeue* yang terjadi. Setiap queue larik harus melingkar: pakailah Persamaan (5.6).
9. **Sisa bagi negatif.** Dalam C++ dan Java, `(i - 1) % N` bernilai -1 ketika `i` bernilai 0, dan `data[-1]` berada di luar batas. Tulislah `(i - 1 + N) % N`, seperti pada Persamaan (5.10). `%` di Python selalu nonnegatif, sehingga kode yang diporting *dari* Python paling berisiko.
10. **Menguji `front == rear` untuk penuh sekaligus kosong.** Dengan dua indeks tanpa pencacah, larik yang penuh sama sekali tampak persis seperti yang kosong (**Gambar 5.15**). Simpan pencacah, atau sisakan satu slot kosong dan pakai Persamaan (5.9).
11. **Memakai `list.pop(0)` sebagai queue Python atau `java.util.Stack` sebagai stack Java.** Yang pertama $O(n)$ per *dequeue*; yang kedua kelas lama tersinkronisasi yang juga mengizinkan akses ke tengah. Pakailah `collections.deque` dan `ArrayDeque`.
12. **Mempertahankan stack redo setelah suntingan baru.** Suntingan baru harus mengosongkan stack redo; jika tidak, *redo* menghidupkan kembali keadaan dari riwayat yang sudah ditinggalkan.
13. **Mengembalikan alamat variabel lokal.** Variabel lokal hidup di *stack frame* fungsinya, yang di-*pop* ketika fungsi kembali. Pointer atau referensi ke variabel itu langsung menggantung, dan pemanggilan fungsi berikutnya menimpa memorinya.
14. **Menyalin kelas yang memiliki larik.** Kelas stack atau queue yang mengalokasikan dengan `new[]` lalu disalin anggota demi anggota berakhir dengan dua objek yang menghapus larik yang sama. Hapuslah operasi salinnya (seperti pada kelas-kelas kita) atau implementasikan dengan benar, sebagaimana dijelaskan Bab 6.

5.20 Ringkasan Bab

- **Stack** menambah dan mengambil di satu ujung, puncak: terakhir masuk, pertama keluar. Operasi `push`, `pop`, `peek`, `isEmpty`, dan `isFull` masing-masing berjalan dalam waktu $O(1)$ pada larik dengan indeks `top`, dengan $top = -1$ berarti kosong dan size $= top + 1$. *Overflow* (*push* saat penuh) dan *underflow* (*pop* saat kosong) harus ditangani dengan sengaja.
- **Menara Hanoi** adalah tiga stack dengan aturan ukuran. Banyaknya langkah minimum memenuhi $M(n) = 2M(n-1) + 1$, sehingga $M(n) = 2^n - 1$; teka-teki ini dapat diselesaikan dengan perulangan yang hanya memakai operasi stack, dan Bab 10 menyelesaikannya dengan rekursi.
- **Larik berlipat dua** menghapus batas kapasitas dengan biaya teramortisasi $O(1)$ per *push*, karena $1 + 2 + \cdots + 2^k < 2n$. **Stack bertaut** menghapusnya dengan satu simpul per elemen.
- Stack **mencocokkan tanda kurung** (*push* pembuka, *pop* dan bandingkan pada penutup, berakhir kosong), mengimplementasikan ***undo* dan *redo*** (dua stack; suntingan baru mengosongkan redo), **mengubah infiks menjadi postfiks** (algoritma *shunting-yard* Dijkstra, digerakkan presedensi dan asosiativitas) dan **mengevaluasi postfiks** (*push* operan; operator mengambil y lalu x dan melakukan *push* x op y), serta menjadi mesin **pemanggilan fungsi** (satu *frame* per pemanggilan; rekursi adalah *frame-frame* dari fungsi yang sama; rekursi yang lepas kendali meluapkan stack).
- **Queue** menambah di belakang dan mengambil di depan: pertama masuk, pertama keluar. Implementasi larik linear entah **menggeser elemen** pada setiap *dequeue* ($O(n)$) atau **indeksnya bergeser** sampai melaporkan penuh palsu.
- **Queue sirkular** melingkarkan indeksnya dengan $i \leftarrow (i + 1) \bmod N$, memberikan operasi $O(1)$ tanpa slot yang hilang. Ia membedakan penuh dari kosong entah dengan **pencacah** atau dengan menyisakan **satu slot kosong** (penuh bila $(\text{rear} + 1) \bmod N = \text{front}$). Dalam C++ dan Java, jagalah operan kiri `%` tetap nonnegatif.
- **Deque** menambah dan mengambil di kedua ujung; pada larik sirkular, `pushFront` memakai $(\text{front} - 1 + N) \bmod N$. **Priority queue** melayani prioritas tertinggi lebih dulu; *heap* pada Bab 14 membuatnya $O(\lg n)$ per operasi.
- Dalam praktik: `std::stack`, `std::queue`, `std::deque`, dan `std::priority_queue` di C++ (`pop` mengembalikan `void`; akses saat kosong tak terdefinisi); `ArrayDeque` dan `PriorityQueue` di Java; `list`, `collections.deque`, `queue.Queue`, dan `heapq` di Python.

5.21 Soal Latihan

5.1 Mulai dari stack larik kosong berkapasitas 5, lakukan: *push* 5, *push* 3, *pop*, *push* 7, *push* 2, *pop*, *pop*, *push* 9, *push* 4, *peek*. Sebutkan nilai yang dikembalikan setiap *pop* dan *peek*, isi stack dari dasar ke puncak pada akhirnya, dan nilai `top` pada akhirnya.

5.2 (a) Berapa langkah yang diperlukan Menara Hanoi untuk 20 cakram, dan berapa lama waktunya dengan satu langkah per detik? (b) Dalam penyelesaian untuk $n = 4$ cakram, pada langkah bernomor berapa saja cakram 1 berpindah, dan pada satu langkah yang mana cakram 4 berpindah? (c) Umumkan: pada langkah-langkah mana cakram k berpindah?

5.3 Bilangan 1, 2, 3, 4 di-*push* ke stack kosong dengan urutan itu, tetapi *pop* boleh dilakukan kapan saja, dan setiap bilangan yang di-*pop* dicetak. Manakah barisan keluaran berikut yang mungkin? Untuk yang mungkin, berikan barisan *push* dan *pop*-nya; untuk yang tidak mungkin, jelaskan alasannya. (a) 1 2 3 4 (b) 4 3 2 1 (c) 2 4 3 1 (d) 3 1 4 2 (e) 1 4 2 3.

5.4 Tambahkan tiga fungsi anggota ke `ArrayStack` pada **Kode 5.2**: `search(x)`, yang mengembalikan jarak salinan `x` teratas dari puncak (0 untuk puncak itu sendiri) atau -1; `edit(k, x)`, yang mengganti elemen k posisi di bawah puncak; dan `stats()`, yang mencetak total, rata-rata, elemen terbesar, dan elemen terkecil sebuah stack bilangan bulat. Apakah fungsi-fungsi ini semestinya ada pada stack? Diskusikan.

5.5 Telusuri **Pseudokode 5.2** pada `{()}`, pada `(((`, dan pada `)(`. Berikan pesan yang dihasilkan setiap masukan.

5.6 Ubahlah menjadi postfiks dengan **Pseudokode 5.3**, sambil memperlihatkan stack setelah setiap token, lalu evaluasi yang pertama: (a) `9-2*3+1` (b) `(a+b)*(c-d)/e` (c) `a^b^c*d`.

5.7 Evaluasilah ekspresi postfiks `6 2 3 + - 3 8 2 / + * 2 ^ 3 +` dengan **Pseudokode 5.4**, sambil memperlihatkan stack setelah setiap token.

5.8 Dalam notasi prefiks operator datang lebih dulu. Jelaskan cara mengevaluasi ekspresi prefiks dengan satu stack dengan memindainya dari **kanan ke kiri**, sebutkan *pop* mana yang menjadi operan kiri, lalu evaluasi `- * 2 3 4` dan `* - 7 3 + 1 1`.

5.9 Dua stack harus berbagi satu larik berisi N slot sedemikian sehingga tidak satu pun meluap sebelum seluruh N slot terpakai. Rancanglah representasinya dan keempat operasi `push1`, `push2`, `pop1`, `pop2`, lalu nyatakan uji luapannya.

5.10 Implementasikan queue menggunakan dua stack tanpa penyimpanan lain. Tunjukkan bahwa walaupun satu `dequeue` dapat berbiaya $O(n)$, setiap barisan m operasi berbiaya total $O(m)$, sehingga setiap operasi $O(1)$ teramortisasi.

5.11 Sebuah queue sirkular dengan rancangan pencacah pada Subbab 5.13.2 memiliki $N = 6$, `front` = 4, dan `count` = 0. Lakukan: *enqueue* A, *enqueue* B, *enqueue* C, *dequeue*, *enqueue* D, *enqueue* E, *enqueue* F, *enqueue* G, *enqueue* H. Setelah setiap operasi berikan `front`, `count`, slot yang ditulis atau dibaca, dan apakah operasinya berhasil.

5.12 Sebuah queue sirkular dengan rancangan satu-slot-kosong memiliki $N = 8$, `front` = 6, dan `rear` = 2. (a) Berapa elemen yang dimuatnya? (b) Slot mana yang ditempati, dalam urutan queue? (c) Berapa kali lagi *enqueue* berhasil sebelum penuh? (d) Berapa `front` dan `rear` setelah dua *dequeue* disusul *enqueue* sebanyak yang masih berhasil?

5.13 Seorang mahasiswa mengusulkan obat bagi queue bergeser pada Subbab 5.12.2 sebagai berikut: setiap kali sebuah *dequeue* membuat queue kosong, setel ulang `head = 0` dan `tail = -1`. Apakah ini menyembuhkan penuh palsu? Berikan barisan operasi yang tetap gagal, atau buktikan bahwa tidak ada.

5.14 *Sliding-window maximum* dari larik $A[0..n-1]$ untuk ukuran jendela k adalah daftar $\max(A[i..i+k-1])$ untuk $i = 0, \ldots, n-k$. Jelaskan algoritma $O(n)$ yang memelihara deque berisi indeks, dan telusuri pada $A = [1,3,-1,-3,5,3,6,7]$ dengan $k = 3$.

5.15 Di meja Profitina Laundry, pesanan dilayani menurut urutan kedatangan, tetapi pesanan *ekspres* harus dilayani sebelum semua pesanan reguler, dan di antara pesanan ekspres urutan kedatangan tetap harus dihormati. (a) Jelaskan mengapa satu queue FIFO tidak cukup. (b) Rancanglah solusi dengan dua queue. (c) Rancanglah solusi dengan satu priority queue, dan jelaskan apa yang harus dimuat prioritasnya agar urutan kedatangan terjaga di dalam setiap golongan.

5.16 Rancanglah stack yang mendukung `push`, `pop`, dan `getMin` (elemen terkecil yang sedang berada di stack), semuanya dalam waktu $O(1)$.

5.17 *Stack frame* sebuah fungsi rekursif menempati 64 byte. Kira-kira seberapa dalam rekursinya dapat berjalan sebelum meluapkan call stack 1 MB? Dan call stack 8 MB? Mengapa jawaban sesungguhnya lebih kecil?

5.18 `std::stack::pop` mengembalikan `void`, dan `top()` pada stack kosong adalah perilaku tak terdefinisi. Tulislah fungsi C++ kecil `template <typename T> T popValue(std::stack<T>& s)` yang mengembalikan elemen yang diambil dan melempar `std::underflow_error` pada stack kosong. Mengapa perancang pustaka standar tidak membuat `pop` berperilaku demikian?

5.22 Penyelesaian Soal Latihan

5.1 Operasi dan hasilnya, dengan stack ditulis dari dasar ke puncak:

- *push* 5 → [5], $top = 0$; *push* 3 → [5 3], $top = 1$.
- *pop* → mengembalikan **3**; [5], $top = 0$.
- *push* 7 → [5 7]; *push* 2 → [5 7 2], $top = 2$.
- *pop* → mengembalikan **2**; *pop* → mengembalikan **7**; [5], $top = 0$.
- *push* 9 → [5 9]; *push* 4 → [5 9 4], $top = 2$.
- *peek* → mengembalikan **4**; tidak ada yang berubah.

Isi akhir, dari dasar ke puncak: 5 9 4, dengan $top = 2$ dan ukuran $top + 1 = 3$ menurut Persamaan (5.4). Kapasitas 5 tidak pernah tercapai.

5.2 (a) Menurut Persamaan (5.2), $M(20) = 2^{20} - 1 = 1{,}048{,}575$ langkah. Dengan satu langkah per detik itu sama dengan $1{,}048{,}575 / 86{,}400 \approx 12.1$ hari.

(b) Untuk $n = 4$ (genap), cakram 1 berpindah pada setiap langkah ganjil: 1, 3, 5, 7, 9, 11, 13, 15 — delapan kali. Cakram 4 berpindah tepat sekali, pada langkah $2^3 = 8$, di tengah-tengah 15 langkah: $7 = M(3)$ langkah pertama menyingkirkan tiga cakram, langkah 8 memindahkan cakram 4, dan 7 langkah terakhir membangun kembali menara tiga cakram di atasnya.

(c) Cakram k pertama kali berpindah pada langkah 2^{k-1} lalu setiap 2^k langkah: pada langkah $2^{k-1}, 3 \cdot 2^{k-1}, 5 \cdot 2^{k-1}, \ldots$ Jadi cakram 1 berpindah pada langkah ganjil, cakram 2 pada langkah 2,6,10, …, cakram 3 pada langkah 4,12,20, … Secara keseluruhan cakram k berpindah 2^{n-k} kali, dan $\sum_{k=1}^{n} 2^{n-k} = 2^n - 1$, yaitu Persamaan (5.2) lagi.

5.3 Sebuah bilangan hanya dapat dicetak ketika berada di puncak, dan bilangan masuk dalam urutan menaik.

(a) 1 2 3 4: mungkin — *push* 1, *pop*; *push* 2, *pop*; *push* 3, *pop*; *push* 4, *pop*.

(b) 4 3 2 1: mungkin — *push* 1, 2, 3, 4, lalu *pop* empat kali.

(c) 2 4 3 1: mungkin — *push* 1, *push* 2, *pop* (2); *push* 3, *push* 4, *pop* (4); *pop* (3); *pop* (1).

(d) 3 1 4 2: **tidak mungkin.** Untuk mencetak 3 lebih dulu, 1 dan 2 sudah harus di-*push*, dan keduanya berada di bawah 3 dengan 2 di atas 1. Setelah 3 di-*pop*, puncaknya 2, sehingga 1 tidak dapat keluar sebelum 2.

(e) 1 4 2 3: **tidak mungkin.** Setelah 1 dan 4 dicetak, 2 dan 3 sama-sama berada di stack dengan 3 di atas 2, sehingga 3 harus dicetak sebelum 2.

Aturan umumnya: sebuah barisan tidak mungkin tepat ketika memuat tiga nilai $a < b < c$ yang dicetak dalam urutan c, a, b.

5.4 Satu implementasi yang mungkin, ditulis sebagai anggota `ArrayStack<T>`:

```
int search(const T& x) const {              // distance from the top, or -1
    for (int i = top_; i >= 0; --i)
        if (data_[i] == x) return top_ - i;
    return -1;
}
bool edit(int k, const T& x) {              // k = 0 is the top
    if (k < 0 || k > top_) return false;
    data_[top_ - k] = x;
    return true;
}
// stats() for ArrayStack<int>: walk i = 0..top_, accumulating
// total, max and min; average = total / (double)(top_ + 1),
// guarding the empty case first.
```

`search` dan `stats` bernilai $O(n)$ dan `edit` bernilai $O(1)$. Fungsi-fungsi ini berguna untuk *debugging* dan latihan, tetapi secara ketat melanggar ADT stack: stack hanya menjanjikan akses ke puncak. Kelas yang menyediakannya sebenarnya adalah larik dengan antarmuka mirip stack. Pustaka nyata mengikuti garis yang sama — `std::stack` sama sekali tidak menyediakan pencarian, sedangkan `Stack.search` pada Java yang lama ada justru karena `Stack` dibangun di atas `Vector`. Jika Anda perlu mencari, tanyakan apakah stack memang struktur yang tepat.

5.5

- `{(})`: *push* `{`, *push* `(`; di posisi 2 penutup `}` mendapati `(` di puncak: **“error at 2: ‘}’ cannot close ‘(’”**.
- `(((`: tiga *push*, pemindaian berakhir dengan tiga tanda kurung di stack: **“error at end: 3 bracket(s) never closed, top is ‘(’”**.
- `)(`: di posisi 0 penutup `)` mendapati stack kosong: **“error at 0: ‘)’ has no opening partner”**. Algoritma berhenti pada galat pertama, sehingga `(` yang tak tertutup tidak pernah dilaporkan.

5.6 (a) `9-2*3+1`: `9` → keluaran `9`; `-` → *push*; `2` → keluaran `9 2`; `*` (di atas `-`, lebih tinggi) → *push*, stack `- *`; `3` → keluaran `9 2 3`; `+` → *pop* `*` (lebih tinggi), *pop* `-` (sama, asosiatif-kiri), *push* `+`, keluaran `9 2 3 * -`; `1` → keluaran `9 2 3 * - 1`; akhir → *pop* `+`. Postfiks **`9 2 3 * - 1 +`**. Evaluasi: *push* 9, 2, 3; `*` → $2 \cdot 3 = 6$, stack `9 6`; `-` → $9 - 6 = 3$; *push* 1; `+` → \$3 + 1 = \$ **4**, sesuai dengan $9 - 6 + 1$.

(b) `(a+b)*(c-d)/e`: `(` *push*; `a` keluar; `+` *push* (di atas `(`); `b` keluar; `)` mengeluarkan `+`, membuang `(` → keluaran `a b +`; `*` *push*; `(` *push*; `c` keluar; `-` *push*; `d` keluar; `)` mengeluarkan `-` → keluaran `a b + c d -`; `/` → *pop* `*` (sama, kiri) lalu *push* `/`, keluaran `a b + c d - *`; `e` keluar; akhir mengeluarkan `/`. Postfiks **`a b + c d - * e /`**.

(c) `a^b^c*d`: `a` keluar; `^` *push*; `b` keluar; `^` bertemu `^` (sama, asosiatif-**kanan**) → tidak ada *pop*, *push*, stack `^ ^`; `c` keluar; `*` → *pop* `^`, *pop* `^` (keduanya lebih tinggi), *push* `*`, keluaran `a b c ^ ^`; `d` keluar; akhir mengeluarkan `*`. Postfiks **`a b c ^ ^ d *`**, yang berarti $(a^{(b^c)}) \cdot d$.

5.7 Stack setelah setiap token (dasar → puncak):

Token	Stack	Token	Stack
6	6	2	1 3 8 2
2	6 2	/	1 3 4
3	6 2 3	+	1 7
+	6 5	*	7
-	1	2	7 2
3	1 3	^	49
8	1 3 8	3	49 3
		+	52

Hasilnya **52**. Pada `-`, *pop* memberikan $y = 5$, $x = 6$, dan $x - y = 1$; pada `/`, $y = 2$, $x = 8$, $8/2 = 4$.

5.8 Memindai prefiks dari kanan ke kiri menjumpai setiap operan sebelum operatornya, sama seperti pemindaian kiri ke kanan pada postfiks. Jadi: *push* operan; pada operator, *pop* dua kali dan *push* hasilnya. Karena pemindaiannya terbalik, *pop* **pertama** kini adalah operan **kiri** x dan yang kedua adalah operan kanan y.

`- * 2 3 4` dari kanan: *push* 4, *push* 3, *push* 2; `*` mengambil $x = 2$, $y = 3$, melakukan *push* 6; `-` mengambil $x = 6$, $y = 4$, melakukan *push* **2**. (Infiks: $2 \cdot 3 - 4$.)

`* - 7 3 + 1 1` dari kanan: *push* 1, *push* 1; `+` → 2; *push* 3, *push* 7; `-` mengambil $x = 7$, $y = 3 \to 4$; `*` mengambil $x = 4$, $y = 2 \to$ **8**. (Infiks: $(7 - 3)(1 + 1)$.)

5.9 Biarkan stack 1 tumbuh ke atas dari slot 0 dengan indeks `top1` (awalnya -1) dan stack 2 tumbuh ke bawah dari slot $N - 1$ dengan indeks `top2` (awalnya N). Maka `push1(x)`: `data[++top1] = x`; `pop1()`: `return data[top1--]`; `push2(x)`: `data[--top2] = x`; `pop2()`: `return data[top2++]`. Masing-masing stack kosong ketika `top1 = -1` dan `top2 = N`, dan larik penuh — untuk **kedua** *push* — tepat ketika `top1 + 1 == top2`. Tidak ada slot yang terbuang, dan tidak satu pun stack meluap selama yang lain masih menyisakan ruang. Setiap operasi $O(1)$.

5.10 Simpan stack *in* dan stack *out*. `enqueue(x)` melakukan *push* x ke *in*. `dequeue()`: jika *out* kosong, *pop* setiap elemen *in* dan *push* ke *out* — yang membalik urutannya, menempatkan elemen tertua di puncak — lalu *pop out*. Satu *dequeue* yang memicu pemindahan berbiaya $O(n)$. Namun ikutilah satu elemen sepanjang hidupnya: ia di-*push* ke *in* sekali, di-*pop* dari *in* sekali, di-*push* ke *out* sekali, dan di-*pop* dari *out* sekali. Empat langkah $O(1)$ per elemen, bagaimanapun urutan operasinya berselang-seling. Jadi m operasi berbiaya paling banyak $4m$ langkah — $O(1)$ teramortisasi per operasi, dengan pembukuan yang sama seperti Persamaan (5.5).

5.11 Dengan $N = 6$, slot kosong pertama adalah (front + count) mod 6 menurut Persamaan (5.7):

Operasi	Slot	`front`	`count`	Hasil
awal	—	4	0	kosong
enqueue A	$(4+0) \bmod 6 = 4$	4	1	berhasil
enqueue B	$(4+1) \bmod 6 = 5$	4	2	berhasil
enqueue C	$(4+2) \bmod 6 = 0$	4	3	berhasil, melingkar
dequeue	baca slot 4 → A	5	2	berhasil
enqueue D	$(5+2) \bmod 6 = 1$	5	3	berhasil
enqueue E	$(5+3) \bmod 6 = 2$	5	4	berhasil
enqueue F	$(5+4) \bmod 6 = 3$	5	5	berhasil
enqueue G	$(5+5) \bmod 6 = 4$	5	6	berhasil, kini penuh
enqueue H	—	5	6	ditolak: `count = N`

Urutan queue pada akhirnya, dari `front`: B (5), C (0), D (1), E (2), F (3), G (4).

5.12 (a) Menurut Persamaan (5.8), ukuran $= (2 - 6 + 8) \bmod 8 = 4$. (b) Slot 6, 7, 0, 1, dalam urutan queue itu; `rear` = 2 adalah slot kosong berikutnya. (c) Queue memuat paling banyak $N - 1 = 7$, sehingga **3** *enqueue* lagi berhasil (ke slot 2, 3, 4); lalu `rear` = 5 dan $(5 + 1) \bmod 8 = 6 =$ front: penuh. (d) Dua *dequeue* membaca slot 6 dan 7, sehingga `front` = 0. Mulai dari `rear` = 2 dengan 2 elemen, lima *enqueue* berhasil (slot 2, 3, 4, 5, 6), menyisakan `rear` = 7 dan $(7 + 1) \bmod 8 = 0 =$ front: penuh dengan 7 elemen. Jadi `front` = 0, `rear` = 7.

5.13 Tidak menyembuhkan. Penyetelan ulang hanya membantu ketika queue benar-benar kosong. Contoh penyangkal dengan kapasitas 5: *enqueue* 1, 2, 3, 4, 5 (`tail` = 4); *dequeue* sekali (`head` = 1, satu elemen keluar, queue tidak kosong, jadi tidak ada penyetelan ulang); *enqueue* 6 → ditolak, padahal slot 0 kosong. Pola lalu lintas apa pun yang membuat queue tidak pernah benar-benar kosong — keadaan normal pencetak atau router yang sibuk — bergeser menuju penuh palsu. Hanya pelingkaran yang memperbaiki masalah ini.

5.14 Simpan deque berisi **indeks** yang nilainya menurun dari depan ke belakang. Untuk setiap i dari 0 sampai $n - 1$: (1) *pop* dari depan setiap indeks $\leq i - k$ (sudah keluar dari jendela); (2) *pop* dari belakang setiap indeks yang nilainya $\leq A[i]$ (tidak mungkin lagi menjadi maksimum selama $A[i]$ berada di jendela); (3) *push* i di belakang; (4) begitu $i \geq k - 1$, keluarkan A[front]. Setiap indeks di-*push* sekali dan di-*pop* paling banyak sekali, sehingga total pekerjaannya $O(n)$.

Penelusuran pada $[1,3,-1,-3,5,3,6,7]$, $k = 3$ (deque memperlihatkan indeks, dengan nilai dalam kurung siku):

i	$A[i]$	Deque setelah langkah	Keluaran
0	1	0 [1]	—
1	3	1 [3]	—
2	−1	1 [3], 2 [−1]	3
3	−3	1 [3], 2 [−1], 3 [−3]	3
4	5	4 [5]	5
5	3	4 [5], 5 [3]	5
6	6	6 [6]	6
7	7	7 [7]	7

Maksimum jendelanya adalah **3 3 5 5 6 7**. Pada $i = 4$, indeks 1 di-*pop* dari depan karena $1 \leq 4 - 3$, dan indeks 3 dan 2 di-*pop* dari belakang karena nilainya di bawah 5.

5.15 (a) Satu queue FIFO melayani ketat menurut kedatangan, sehingga pesanan ekspres yang tiba setelah lima pesanan reguler menunggu di belakang kelimanya.

(b) Simpan dua queue FIFO, `express` dan `regular`. Masukkan setiap pesanan ke queue golongannya. Untuk melayani: jika `express` tidak kosong, lakukan *dequeue* darinya; jika kosong, lakukan *dequeue* dari `regular`. Kedua operasi tetap $O(1)$, dan urutan kedatangan terjaga di dalam setiap golongan karena setiap golongan adalah queue FIFO-nya sendiri.

(c) Pakailah priority queue yang kuncinya berupa pasangan (golongan, nomor kedatangan), dengan ekspres diurutkan sebelum reguler dan sebuah pencacah yang bertambah pada setiap kedatangan. Membandingkan pasangan secara leksikografis melayani semua pesanan ekspres lebih dulu dan, di dalam satu golongan, nomor kedatangan terkecil lebih dulu. Nomor kedatangan itu penting: *heap* tidak *stabil*, sehingga tanpanya dua pesanan segolongan dapat keluar dalam urutan mana pun. Solusi (b) lebih sederhana dan lebih cepat bila hanya ada dua golongan; (c) dapat diperluas ke berapa pun tingkat prioritas.

5.16 Simpan stack kedua, `mins`, di samping stack utama. Pada `push(x)`: *push* x ke stack utama, dan *push* min(x,mins.top()) ke `mins` (atau cukup x bila `mins` kosong). Pada `pop()`: *pop* kedua stack. Maka `getMin()` adalah `mins.top()`. Invariannya: puncak `mins` selalu merupakan minimum elemen yang sedang berada di stack utama, karena `mins` mencatat minimum *pada saat setiap push* dan *pop* memulihkan persis keadaan sebelumnya. Ketiga operasi $O(1)$; ruang tambahannya $O(n)$. (Varian hemat ruang hanya melakukan *push* ke `mins` bila $x \leq$ minimum saat ini.)

5.17 Stack 1 MB berukuran 1,048,576 byte, sehingga $1{,}048{,}576/64 = 16{,}384$ *frame*. Stack 8 MB memungkinkan $8 \times 16{,}384 = 131{,}072$ *frame*. Batas sesungguhnya lebih rendah karena `main`, *runtime* C++, dan kode awal sistem operasi sudah menempati sebagian stack, kompilator dapat menambahkan *padding* dan register yang disimpan pada setiap *frame*, dan pada *build* untuk *debugging* *frame* biasanya lebih besar. Di Python, interpreter berhenti pada batas rekursinya (bawaannya 1.000) jauh sebelum stack sistem operasi penuh.

5.18 Satu versi:

```
template <typename T>
T popValue(std::stack<T>& s) {
    if (s.empty()) throw std::underflow_error("pop from an empty stack");
    T value = std::move(s.top());   // take the element out first ...
    s.pop();                        // ... then remove it
    return value;
}
```

Pustaka standar memisahkan `top` dan `pop` demi *exception safety*. Jika `pop` mengambil elemen lalu mengembalikannya sebagai nilai, penyalinan ke variabel pemanggil dapat melempar eksepsi (misalnya bila penyalinan `std::string` kehabisan memori) setelah elemennya terlanjur keluar dari stack — dan elemen itu hilang selamanya. Dengan `top` dan `pop` yang terpisah, pemanggil menyalin lebih dulu; jika penyalinan melempar eksepsi, stack tidak tersentuh. Pada fungsi bantu kita risiko yang sama ada pada `return value;`, tetapi C++ modern memindahkan atau meniadakan salinan itu, yang untuk tipe-tipe standar tidak melempar eksepsi. Pemeriksaan kekosongan diserahkan kepada pemanggil di pustaka demi kecepatan: program yang sudah tahu stack-nya tidak kosong tidak semestinya membayar uji kedua.

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, sec. 10.1.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, sec. 2.2.1.

[3] É. Lucas, *Récréations mathématiques*, vol. 3. Paris, France: Gauthier-Villars, 1893, pp. 55–59 ("La Tour d'Hanoï").

[4] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011, sec. 1.3.

[5] C. L. Hamblin, "Translation to and from Polish notation," *The Computer Journal*, vol. 5, no. 3, pp. 210–213, 1962, doi: 10.1093/comjnl/5.3.210.

[6] E. W. Dijkstra, "Algol 60 translation: An Algol 60 translator for the X1 and Making a translator for Algol 60," Mathematisch Centrum, Amsterdam, The Netherlands, Rep. MR 35/61, 1961.

[7] Microsoft, "/STACK (Stack allocations)," *Microsoft Learn — C/C++ Build Reference*. [Online]. Available: https://learn.microsoft.com/en-us/cpp/build/reference/stack-stack-allocations. [Accessed: 24-Sep-2026].

[8] ISO/IEC 14882:2024, *Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2024, subclause "Container adaptors".

[9] cppreference.com, "std::stack." [Online]. Available: https://en.cppreference.com/w/cpp/container/stack. [Accessed: 24-Sep-2026].

[10] Oracle, "Class ArrayDeque," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/ArrayDeque.html. [Accessed: 24-Sep-2026].

[11] Oracle, "Class Stack," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Stack.html. [Accessed: 24-Sep-2026].

[12] Python Software Foundation, "collections — Container datatypes," *The Python Standard Library*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/collections.html. [Accessed: 24-Sep-2026].

[13] Python Software Foundation, "queue — A synchronized queue class," *The Python Standard Library*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/queue.html. [Accessed: 24-Sep-2026].

[14] Profitina, "Profitina — AI Visibility (GEO) platform." [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

Bab 6

Pointer, Referensi, dan Fungsi

Di akhir bab ini Anda akan sanggup menggambar program C++ kecil mana pun sebagai kotak dan panah di memori, lalu membaca gambar itu kembali sebagai kode; mendeklarasikan *pointer* (penunjuk), mengambil alamat dengan `&`, mengikutinya dengan `*`, dan menyatakan dalam satu kalimat arti ketiga simbol itu di setiap posisi tempat ia bisa muncul; menggerakkan pointer menyusuri larik (*array*) dan menjelaskan mengapa `p + 1` melompat empat byte untuk sebuah `int` tetapi seratus empat byte untuk sebuah `Book`; menjangkau *field* sebuah struktur melalui pointer dengan `->` dan mengikuti rantai simpul dari satu ke berikutnya, persis yang dilakukan Bab 7 untuk membangun *linked list*; mengalokasikan memori saat program berjalan dengan `new` dan `new[]`, mengembalikannya dengan `delete` dan `delete[]`, serta membangun larik tumbuh yang menggandakan dirinya; mengenali lima bug pointer klasik (*dangling pointer*, kebocoran memori, *double free*, `delete` yang tidak berpasangan, dan dereferensi null), membuat AddressSanitizer dan valgrind menangkap masing-masing, dan membaca laporannya; memakai `nullptr`, referensi, dan `const` sehingga kompilator menangkap kesalahan sebelum program dijalankan; menuturkan kisah *swap* dengan tiga cara, berdasarkan nilai, melalui pointer, dan melalui referensi, serta menjelaskan mengapa hanya dua di antaranya yang berhasil; mengembalikan pointer dan referensi tanpa membuatnya menggantung; mengganti sebagian besar `delete` yang ditulis tangan dengan `std::unique_ptr` dan `std::shared_ptr`; dan menjelaskan dengan tepat apa yang dilakukan Java dan Python ketika sebuah objek dikirim ke metode, termasuk makna frasa *pass by object reference* dan perbedaan antara mengubah objek dan mengikat ulang sebuah nama.

6.1 Dari Mana Kita Melanjutkan

Bab 1 menggambarkan memori sebagai deretan panjang byte bernomor dan menunjukkan bahwa larik adalah sederet slot berukuran sama yang diletakkan berdampingan, sehingga komputer dapat menemukan slot ke-*i* dengan satu perkalian dan satu penjumlahan. Bab 2 dan 3 memakai larik untuk mencari dan mengurutkan. Bab 5 membangun *stack* (tumpukan) dan *queue* (antrean) di atas larik dan menemukan titik lemahnya: larik lahir dengan ukuran tetap, dan antrean yang harus tumbuh melampaui ukuran itu tidak punya tempat untuk pergi.

Sepanjang bab-bab itu kita memakai beberapa simbol tanpa menjelaskannya sampai tuntas. Kita menulis `&catalog[i]` untuk membicarakan tempat sebuah elemen tinggal. Kita menulis kepala fungsi seperti `void printBook(const Book& b)` dan percaya saja bahwa tanda `&` itu berbuat sesuatu yang berguna. Kita mengirim larik ke fungsi dan memperhatikan, barangkali dengan sedikit heran, bahwa fungsi itu dapat mengubah larik milik pemanggil walaupun tidak ada yang tampak mengizinkannya. Bab ini menjelaskan semua itu dari awal, lalu melangkah lebih jauh.

Alasan sebuah buku struktur data menyediakan satu bab penuh untuk ini sederhana saja. Larik menyimpan elemen-elemennya berdampingan, dan kedekatan itulah yang membuatnya cepat. Namun banyak struktur yang masih menanti kita, *linked list* di Bab 7 dan 9, pohon di Bab 11, graf di Bab 13, menyimpan bagian-bagiannya tersebar di mana pun memori kebetulan kosong, lalu menyambungkannya dengan *alamat*. Sebuah simpul dalam *linked list* tidak mengenal tetangganya dari posisi; ia mengenal alamat tetangganya. Segala sesuatu sejak Bab 7 bertumpu pada rasa nyaman sepenuhnya dengan tiga gagasan: alamat, variabel yang menyimpan alamat, dan

sepotong memori yang ada karena program memintanya ketika sedang berjalan. Itulah pointer dan memori dinamis yang menjadi pokok bab ini.

Bab ini juga menjawab pertanyaan yang cepat atau lambat diajukan setiap pemrogram tentang fungsi: *ketika saya menyerahkan sesuatu kepada fungsi, apakah fungsi itu menerima bendanya sendiri atau salinannya?* C++ membiarkan Anda memilih, dan pilihan itu tertulis di setiap kepala fungsi. Java dan Python tidak memberi pilihan, dan masing-masing mempunyai aturan tetapnya sendiri. Mengetahui ketiga aturan itu secara tepat akan menyelamatkan Anda dari sekelompok bug yang kalau tidak begitu sangat sulit dilihat.

Kita tetap memakai contoh berjalan dari bab-bab sebelumnya: *Pustaka Ceria*, perpustakaan komunitas kecil yang katalognya berisi catatan seperti ini.

```
struct Book {
    std::string id, title, author;
    int year;
};
```

Lima buku akan muncul berulang kali: *Laskar Pelangi* (B4001, Andrea Hirata, 2005), *Filosofi Kopi* (B4002, Dee Lestari, 2006), *Bumi Manusia* (B4003, Pramoedya Ananta Toer, 1980), *Negeri 5 Menara* (B4004, Ahmad Fuadi, 2009), dan *Cantik Itu Luka* (B4005, Eka Kurniawan, 2002).

Semua kode C++ dalam bab ini adalah C++17, dikompilasi dengan `g++ -std=c++17 -O2 -Wall -Wextra` tanpa satu peringatan pun; bahasanya didefinisikan oleh standar ISO [1], dan tur yang enak dibaca tentang C++ modern adalah buku Stroustrup [2]. Setiap program C++ juga tersedia dalam Java dan Python di folder kode bab ini, dan versi Java serta Python yang mengajarkan hal baru ditampilkan dalam teks, di Subbab 6.14.

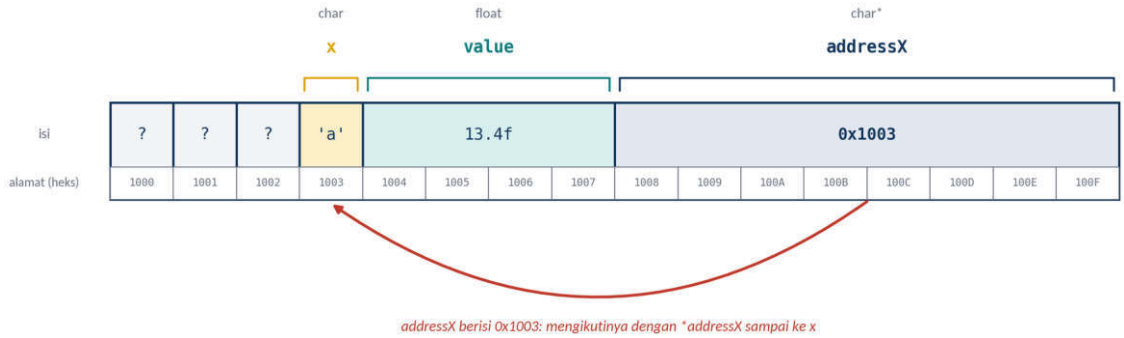
6.2 Memori, Alamat, dan Apa Itu Pointer

6.2.1 Memori sebagai jalan dengan rumah-rumah bernomor

Bayangkan sebuah jalan yang sangat panjang, tempat setiap rumah berukuran sama, masing-masing tepat menampung satu byte, dan setiap pintunya bercat nomor rumah. Nomornya dimulai dari nol dan naik satu demi satu dari rumah ke rumah. Jalan itu adalah memori komputer Anda, dan nomor rumah itu adalah *alamat*. Satu byte terdiri atas delapan bit, cukup untuk menampung bilangan dari 0 sampai 255 atau satu huruf Latin.

Variabel adalah nama yang diberikan kompilator kepada satu atau beberapa rumah yang berurutan. Sebuah `char` memerlukan satu rumah. Sebuah `int` pada mesin masa kini memerlukan empat rumah berurutan, `float` empat, `double` delapan. Ketika kita menyebut “alamat sebuah variabel”, yang kita maksud adalah nomor rumah pertamanya.

Gambar 6.1 memperlihatkan enam belas byte memori yang berurutan. Alamatnya ditulis dalam heksadesimal, basis enam belas, karena begitulah debugger dan C++ mencetaknya: digitnya berjalan dari 0 sampai 9 lalu A sampai F untuk sepuluh sampai lima belas, sehingga byte sesudah `1009` adalah `100A`, dan awalan `0x` di depan bilangan hanya mengumumkan bahwa bilangan itu heksadesimal. Tiga variabel tinggal di sini. `char x` menyimpan huruf `'a'` dalam satu byte di alamat `0x1003`. `float value` menyimpan 13.4 dalam empat byte `0x1004` sampai `0x1007`. Variabel ketiga, `addressX`, menempati delapan byte, dan isinya bukan huruf dan bukan ukuran. Isinya bilangan `0x1003`, yaitu alamat `x`.



Gambar 6.1 Enam belas byte memori. Setiap variabel menempati byte-byte yang berurutan; pointer `addressX` menyimpan alamat `0x1003`, sehingga mengikutinya membawa kita kembali ke `x`.

Variabel ketiga itulah pointer.

> **DEFINISI.** *Pointer* adalah variabel yang nilainya berupa alamat suatu lokasi di memori. Kita katakan pointer itu *menunjuk* apa pun yang tinggal di alamat tersebut. Pointer yang tidak menunjuk apa-apa berisi nilai khusus *null*, yang dalam C++ ditulis `nullptr`.

Bacalah definisi itu dua kali, sebab setiap gagasan sesudahnya bergantung padanya. Pointer tidak berisi buku, bilangan, atau huruf. Pointer berisi *di mana* buku, bilangan, atau huruf itu dapat ditemukan. Nomor rumah seorang kawan yang ditulis di secarik kertas adalah pointer: kertas itu bukan rumahnya, melainkan cara untuk sampai ke rumahnya, dan Anda boleh memfotokopi kertas itu sesering apa pun tanpa membangun satu rumah baru pun.

6.2.2 Operator alamat `&` dan pointer pertama

C++ menyediakan dua operator untuk berpindah antara suatu benda dan alamatnya. Yang pertama adalah `&`, operator *address-of* (pengambil alamat). Ditulis di depan variabel, `&x` berarti "alamat `x`". Yang kedua adalah `*`, yang akan kita temui sebentar lagi. **Kode 6.1** memakai keduanya pada variabel-variabel di **Gambar 6.1**.

Kode 6.1 `01_first_pointer.cpp`: mengambil alamat dengan `&`, menyimpannya dalam pointer, dan mengikuti pointer itu dengan `*`.

```
// Pointer pertama: ambil alamat dengan &, simpan dalam variabel pointer, lalu
// ikuti dengan *.
#include <iostream>

int main() {
    char x = 'a';
    char* addressX = &x;                  // addressX menyimpan alamat x
    std::cout << "x has value " << x << ", stored at address "
              << static_cast<void*>(addressX) << '\n';

    float value = 13.4f;
    float* address = &value;              // pointer ke float
    std::cout << "value = " << value << ", stored at " << address << '\n';
    std::cout << "*address = " << *address << '\n';

    *address = 27.5f;                     // tulis melalui pointer
    std::cout << "after *address = 27.5f, value = " << value << '\n';

    std::cout << "sizeof(char*) = " << sizeof(char*)
              << ", sizeof(float*) = " << sizeof(float*)
              << ", sizeof(double*) = " << sizeof(double*) << '\n';
    return 0;
}
```

```
x has value a, stored at address 0x7ffc2d719d93
value = 13.4, stored at 0x7ffc2d719d94
*address = 13.4
after *address = 27.5f, value = 27.5
sizeof(char*) = 8, sizeof(float*) = 8, sizeof(double*) = 8
```

Mari kita baca pelan-pelan, baris demi baris.

- `char* addressX = &x;` mendeklarasikan variabel bernama `addressX` yang bertipe `char*`, dibaca "pointer ke `char`". Tanda `*` dalam *deklarasi* berarti "variabel ini pointer". Nilai yang kita simpan di dalamnya adalah `&x`, alamat `x`.
- Mencetak alamat memerlukan satu siasat kecil. `std::cout` memperlakukan `char*` sebagai awal sebuah string teks dan akan mencoba mencetak karakter sampai menemukan byte nol, jadi kita mengonversi pointer itu ke `void*`, "pointer ke sesuatu yang tidak ditentukan", yang dicetak `cout` sebagai bilangan heksadesimal. Untuk tipe pointer lain, seperti `float*`, konversi itu tidak diperlukan.
- `float* address = &value;` adalah pointer ke `float`. Mencetaknya menampilkan alamat; mencetak `*address` menampilkan 13.4.
- `*address = 27.5f;` menulis *melalui* pointer. Variabel `value` sendiri berubah, karena dalam gambar ini hanya ada satu float dan pointer hanyalah jalan kedua menuju ke sana.
- Baris terakhir menunjukkan bahwa setiap pointer, apa pun yang ditunjuknya, berukuran sama pada satu mesin: delapan byte pada komputer 64-bit, karena di sana alamat adalah bilangan 64-bit.

Alamat yang tercetak adalah alamat sungguhan dari satu kali eksekusi pada mesin Linux 64-bit. Di komputer Anda, bahkan pada eksekusi berikutnya dari program yang sama, alamat itu akan berbeda, karena sistem operasi sengaja menempatkan program di alamat yang sulit ditebak untuk mempersulit serangan (teknik ini disebut *address space layout randomization*). Yang stabil adalah *selisih* antaralamat, dan itulah yang diukur sepanjang sisa bab ini. Perhatikan bahwa `x` berada di alamat yang berakhiran `3` dan `value` tepat di alamat berikutnya yang berakhiran `4`: kompilator kebetulan meletakkan keduanya berdampingan, persis seperti di **Gambar 6.1**.

6.2.3 Operator dereferensi `*`

Operator kedua, `*` yang ditulis di depan pointer di dalam sebuah *ekspresi*, disebut *dereferensi* atau *indireksi*. `*address` berarti "pergilah ke alamat yang tersimpan di `address`, lalu pakailah benda yang tinggal di sana". Di sisi kanan penugasan ia membaca; di sisi kiri ia menulis.

> **DEFINISI.** Untuk pointer `p` bertipe `T*` yang menunjuk sebuah objek, ekspresi `*p` *adalah* objek itu: sebuah *lvalue* bertipe `T` yang dapat dibaca atau ditugasi. Kedua operator saling membalik: `*&x` adalah `x`, dan `&*p` adalah `p`.

Kehidupan ganda `*` ini membingungkan hampir setiap pemula, sehingga layak mendapat tabel tersendiri, **Tabel 6.1**. Hal yang sama terjadi pada `&`, yang di Subbab 6.9 akan kita lihat juga dipakai untuk mendeklarasikan referensi.

Tabel 6.1 Tiga makna `*` dan `&` dalam C++. Konteks selalu menentukan makna mana yang dimaksud.

Tempat muncul	* berarti	& berarti
Dalam deklarasi: `int* p;` / `int& r = a;`	`p` adalah pointer ke `int`	`r` adalah referensi ke `int` (Subbab 6.9)
Dalam ekspresi, di depan pointer atau variabel: `*p` / `&a`	ikuti pointer (dereferensi)	ambil alamat `a`
Di antara dua operand: `a * b` / `a & b`	perkalian	AND bitwise

Tabel 6.2 membandingkan pointer dengan variabel biasa, butir demi butir.

Tabel 6.2 Variabel biasa dan pointer, berdampingan.

	Variabel biasa `int a;`	Pointer `int* p;`
Berisi	nilai bertipe sesuai, misalnya 13	alamat, misalnya `0x7ffc1645e8c4`, atau `nullptr`
Operasi yang lazim	`+ - * /`, perbandingan	`&` untuk memperoleh alamat, `*` untuk mengikutinya, `+ -` untuk bergerak di dalam larik, perbandingan
Ukuran	bergantung pada tipe (1, 4, 8, … byte)	sama untuk setiap tipe pointer pada satu mesin (8 byte pada 64-bit)
Umur benda yang dirujuk	ia *adalah* objeknya	bisa hidup lebih lama, atau lebih singkat, daripada objek yang ditunjuknya

Baris terakhir **Tabel 6.2** adalah benih setiap bug pointer di Subbab 6.7: variabel biasa dan nilainya hidup dan mati bersama, sedangkan pointer dan benda yang ditunjuknya mempunyai kehidupan masing-masing, dan tidak ada yang memaksa keduanya sepakat.

6.2.4 Tipe pointer itu penting

Setiap pointer mempunyai tipe yang ditunjuk, dan kompilator menganggapnya serius. `float*` dipakai untuk menjangkau `float`, dan ketika Anda menulis `*address` kompilator membaca tepat empat byte dan menafsirkannya sebagai bilangan titik-mengambang. Seandainya pointer boleh menunjuk benda bertipe lain, program akan membaca byte yang benar dengan makna yang salah.

Misalkan sebuah program menyimpan alamat sebuah `long` di dalam `int*`. Kompilator C hanya memberi peringatan. Kompilator C++ menolak:

```
int main() {
    long value = 202320242;
    int* warningAddress = &value;
    return *warningAddress == 0;
}
```

```
wrong_type.cpp:3:27: error: cannot convert 'long int*' to 'int*' in initialization
    3 |     int* warningAddress = &value;
      |                           ^~~~~~
      |                           |
      |                           long int*
```

Penolakan itu justru fitur. Pada mesin Linux 64-bit sebuah `long` berukuran delapan byte dan `int` empat, sehingga `int*` yang diarahkan ke `long` diam-diam hanya melihat separuhnya. C++ mengizinkan Anda melangkahi pemeriksaan itu dengan *cast* eksplisit, dan program di **Kode 6.3** melakukannya sekali, dengan sengaja, untuk menghitung byte; dalam kode biasa Anda semestinya tidak pernah memerlukannya.

6.2.5 Menggambar memori: konvensi kotak dan panah

Alamat sungguhan itu panjang, berubah dari eksekusi ke eksekusi, dan menyembunyikan struktur yang kita pedulikan. Mulai sekarang kita menggambar memori seperti para pemrogram menggambarnya di papan tulis: variabel adalah kotak dengan namanya di samping; nilai ditulis di dalam kotak; dan pointer adalah kotak berisi sebuah titik dengan panah dari titik itu ke kotak yang ditunjuknya. Pointer null adalah kotak bersimbol ⌀ atau bertuliskan `nullptr`. Hanya itu. Dengan konvensi ini keadaan program menjadi gambar yang dapat diperiksa dengan mata, dan setiap bug pointer dalam bab ini menjadi panah yang menunjuk ke tempat yang salah.

Menggambar menuntut disiplin dalam tiga hal. Gambarlah *setiap* variabel, termasuk pointer, sebagai kotaknya sendiri: pointer bukan "panahnya", melainkan variabel yang *menyimpan* panah. Ketika sebuah pernyataan dijalankan, gambar ulang hanya yang diubahnya, dan jangan ubah hal lain. Dan ketika penugasan menyalin pointer, gambarlah panah *baru* dari kotak yang baru; jangan pernah memindahkan panah yang lama. Kebanyakan kesalahan dalam dua subbab berikut berasal dari melewatkan salah satu aturan ini.

6.3 Operasi pada Pointer

Pointer adalah variabel, sehingga ia dapat ditugasi, dibandingkan, dan, di dalam larik, digerakkan. Setiap operasi sederhana bila berdiri sendiri; kuncinya adalah jangan pernah mencampuradukkan operasi pada *pointer* dengan operasi pada benda yang ditunjuknya.

6.3.1 Penugasan: menyalin alamat versus menyalin nilai

Kode 6.2 membahas empat kasus kecil: dua pointer ke satu variabel; menyalin nilai yang ditunjuk ke variabel biasa; menyalin nilai antara dua variabel melalui pointer; dan mengubah variabel melalui pointer.

Kode 6.2 `02_pointer_assign.cpp`: `q = p` menyalin alamat, `*q = *p` menyalin nilai.

```cpp
// Menugasi pointer versus menugasi melalui pointer.
#include <iostream>

int main() {
    std::cout << std::boolalpha;

    // 1. Satu alamat, dua pointer.
    float y = 3.14f;
    float* x1 = &y;
    float* x2 = x1;                          // menyalin alamat, bukan nilai
    std::cout << "1. *x1 = " << *x1 << ", *x2 = " << *x2
              << ", x1 == x2: " << (x1 == x2) << '\n';

    // 2. Salin nilai yang ditunjuk ke variabel biasa.
    int a = 13;
    int* p = &a;
    int b = *p;                              // b mendapat 13; b kotak tersendiri
    std::cout << "2. a = " << a << ", b = " << b
              << ", p points at b: " << (p == &b) << '\n';

    // 3. Dua pointer, dua kotak: *q = *p menyalin nilai antarkotak.
    int c = 3;
    int* q = &c;
    std::cout << "3. before *q = *p: *p = " << *p << ", *q = " << *q
              << ", p == q: " << (p == q) << '\n';
    *q = *p;
    std::cout << "   after  *q = *p: *p = " << *p << ", *q = " << *q
              << ", p == q: " << (p == q) << ", c = " << c << '\n';

    // 4. q = p: kini q juga menunjuk a, dan c tak lagi terjangkau lewat q.
    c = 3;
    q = p;
    std::cout << "4. after  q = p:   *p = " << *p << ", *q = " << *q
              << ", p == q: " << (p == q) << ", c = " << c << '\n';

    // 5. Menulis lewat pointer mana pun mengubah satu-satunya variabel a.
    *q = 99;
    std::cout << "5. after *q = 99:  a = " << a << ", *p = " << *p
              << ", c = " << c << '\n';
    return 0;
}
```

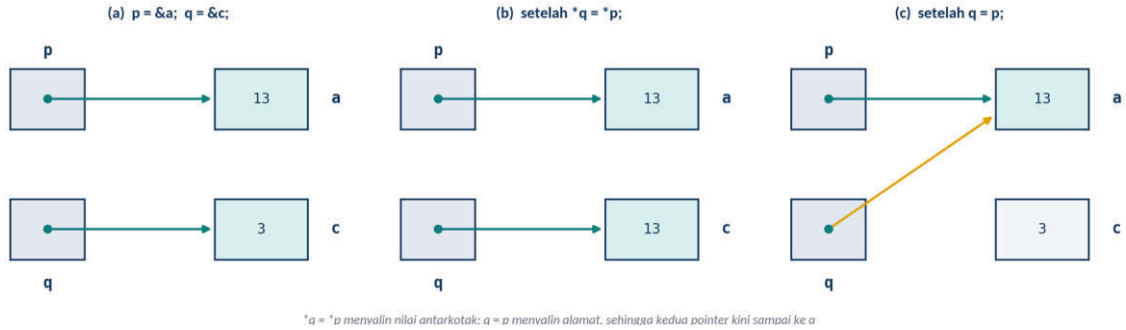
```
1. *x1 = 3.14, *x2 = 3.14, x1 == x2: true
2. a = 13, b = 13, p points at b: false
3. before *q = *p: *p = 13, *q = 3, p == q: false
   after  *q = *p: *p = 13, *q = 13, p == q: false, c = 13
4. after  q = p:   *p = 13, *q = 13, p == q: true, c = 3
5. after *q = 99:  a = 99, *p = 99, c = 3
```

Langkah 1 membuat dua pointer, `x1` dan `x2`, menyimpan alamat yang sama. Tidak ada yang disalin selain delapan byte alamat; tetap hanya ada satu `float`, dan kedua pointer menjangkaunya. Itulah sebabnya `x1 == x2` mencetak `true`: membandingkan dua pointer berarti membandingkan alamat yang disimpannya.

Langkah 2 mendeklarasikan `int b = *p;`. Sisi kanan mengikuti `p` ke `a` dan membaca 13; sisi kiri adalah variabel biasa yang baru dan menerima *salinan* 13. Sejak itu `a` dan `b` tidak berhubungan, dan `p` tidak menunjuk `b`.

Langkah 3 dan 4 adalah inti persoalannya, dan **Gambar 6.2** menggambarnya. Pada (a), `p` menunjuk `a` dan `q` menunjuk `c`. Pernyataan 3, `*q = *p`, membaca melalui `p` dan menulis melalui `q`: *nilai* 13 berpindah dari `a` ke `c`. Panah-panahnya tidak bergerak; `p == q` tetap salah. Pernyataan 4, `q = p`, menyalin *alamat* yang tersimpan di `p` ke `q`. Kini kedua panah sampai ke `a`, dan `c` sama sekali

tidak lagi terjangkau lewat `q`. Pernyataan 5 menulis 99 melalui `q`, dan karena `q` kini menunjuk `a`, yang berubah adalah `a`.



Gambar 6.2 Dua cara menugasi dengan pointer. `*q = *p` memindahkan nilai antarkotak; `q = p` memindahkan panah.

> **WAWASAN KUNCI.** Tanyakan pada setiap penugasan yang melibatkan pointer: *adakah `*` di sisi kiri?* Bila ada bintang, pernyataan itu mengubah objek di ujung panah dan membiarkan setiap panah di tempatnya. Bila tidak ada, pernyataan itu mengubah arah sebuah panah dan tidak menyentuh objek mana pun.

6.3.2 Perbandingan

Dua pointer bertipe sama dapat dibandingkan dengan `==` dan `!=`, yang menanyakan apakah keduanya menyimpan alamat yang sama. Ini adalah uji "apakah dua pointer ini menunjuk objek yang sama?", dan sama sekali berbeda dari menanyakan apakah dua objek mempunyai *nilai* yang sama: pada Langkah 2 di atas, `a` dan `b` sama, 13 = 13, tetapi `&a == &b` salah karena keduanya dua kotak yang berbeda.

Pointer juga dapat dibandingkan dengan `nullptr`, yang menanyakan apakah ia menunjuk ke mana pun, dan di dalam satu larik dua pointer dapat diurutkan dengan `<`, `<=`, `>`, dan `>=`: pointer ke elemen berindeks lebih kecil adalah pointer yang lebih kecil. Mengurutkan pointer ke larik yang *berbeda* atau ke variabel yang tidak berhubungan tidak mempunyai makna yang terdefinisi dalam C++, dan program tidak boleh bergantung padanya.

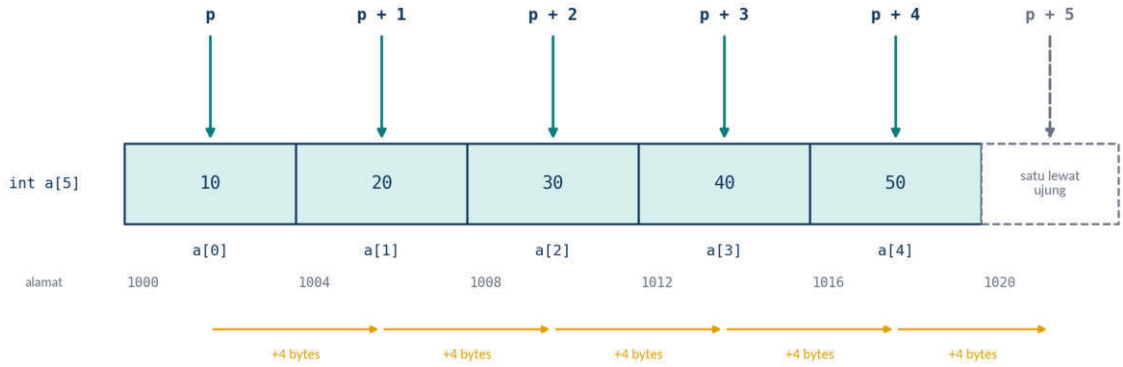
6.3.3 Aritmetika: bergerak di dalam larik

Karena elemen-elemen larik berdampingan, pointer ke satu elemen dapat digeser ke tetangganya dengan menambah atau mengurangkan bilangan bulat. Aturan yang membuatnya bekerja adalah rumus terpenting dalam bab ini. Jika `p` bertipe `T*`, maka

$$\text{alamat}(p + i) = \text{alamat}(p) + i \cdot \text{sizeof}(T) \tag{6.1}$$

dengan `sizeof(T)` adalah banyaknya byte yang ditempati satu `T`. Dengan kata lain: menambahkan 1 pada pointer tidak menggesernya satu byte, melainkan satu *elemen*. Persamaan (6.1) adalah rumus alamat yang sama dengan yang dipakai Bab 1 untuk menemukan slot ke-i sebuah larik, kini ditulis dengan pointer menggantikan alamat dasar larik.

Gambar 6.3 memperlihatkannya untuk larik `int` yang, demi gambar, dimulai di alamat 1000. Setiap langkah pointer bergeser empat byte, karena `sizeof(int)` bernilai 4.



Gambar 6.3 Aritmetika pointer pada `int a[5]`. Setiap `+1` bergeser satu elemen, empat byte; `p + 5` menunjuk satu lewat ujung, posisi yang boleh dihitung dan dibandingkan tetapi tidak pernah boleh didereferensi.

Gagasan yang sama berlaku terbalik. Mengurangkan dua pointer ke larik yang sama menghasilkan banyaknya *elemen* di antara keduanya, bukan banyaknya byte:

$$q - p = \frac{\text{alamat}(q) - \text{alamat}(p)}{\text{sizeof}(T)} \tag{6.2}$$

Hasilnya bertipe `std::ptrdiff_t`, bilangan bulat bertanda, karena $q - p$ negatif bila `q` berada sebelum `p`. **Kode 6.3** memeriksa kedua rumus pada empat tipe elemen. Untuk melihat jarak byte mentah, program itu mengonversi pointer ke `const char*`, karena `sizeof(char)` menurut definisi bernilai 1 sehingga pengurangan pointer `char` menghitung byte.

Kode 6.3 `03_pointer_arith.cpp`: setiap langkah `p + i` bergeser `sizeof(T)` byte, dan pengurangan pointer menghitung elemen.

```cpp
// Aritmetika pointer bergerak per elemen utuh, bukan per byte.
#include <cstddef>
#include <iostream>
#include <string>

struct Book {
    std::string id, title, author;
    int year;
};

template <typename T>
void showSteps(const char* name, T* p) {
    std::cout << name << " (sizeof " << sizeof(T) << ")";
    for (int i = 0; i < 4; ++i) {
        std::ptrdiff_t bytes = reinterpret_cast<const char*>(p + i)
                             - reinterpret_cast<const char*>(p);
        std::cout << "  p+" << i << ": +" << bytes;
    }
    std::cout << '\n';
}

int main() {
    char letters[4] = {'D', 'a', 't', 'a'};
    int counts[4] = {0, 0, 0, 0};
    double prices[4] = {0.0, 0.0, 0.0, 0.0};
    Book shelf[4];
    showSteps("char  ", letters);
    showSteps("int   ", counts);
    showSteps("double", prices);
    showSteps("Book  ", shelf);

    int a[5] = {10, 20, 30, 40, 50};
    int* first = a;                       // sama dengan &a[0]
    int* last = a + 4;                    // sama dengan &a[4]
    std::cout << std::boolalpha;
    std::cout << "*first = " << *first << ", *last = " << *last << '\n';
    std::cout << "*(first + 2) = " << *(first + 2) << ", first[2] = " << first[2] << '\n';
    std::cout << "last - first = " << (last - first) << " elements\n";
    std::cout << "first < last: " << (first < last) << '\n';
```

Kode 6.3 (Lanjutan) *03_pointer_arith.cpp: setiap langkah p + i bergeser sizeof(T) byte, dan pengurangan pointer menghitung elemen.*

```cpp
    int* p = first;
    ++p;                                  // kini di a[1]
    p += 2;                               // kini di a[3]
    --p;                                  // kini di a[2]
    std::cout << "after ++p, p += 2, --p: *p = " << *p
              << ", index = " << (p - a) << '\n';

    bool allSame = true;
    for (int i = 0; i < 5; ++i) {
        allSame = allSame && (&a[i] == a + i) && (a[i] == *(a + i));
    }
    std::cout << "&a[i] == a + i and a[i] == *(a + i) for every i: " << allSame << '\n';
    return 0;
}
```

```
char   (sizeof 1)  p+0: +0  p+1: +1  p+2: +2  p+3: +3
int    (sizeof 4)  p+0: +0  p+1: +4  p+2: +8  p+3: +12
double (sizeof 8)  p+0: +0  p+1: +8  p+2: +16  p+3: +24
Book   (sizeof 104)  p+0: +0  p+1: +104  p+2: +208  p+3: +312
*first = 10, *last = 50
*(first + 2) = 30, first[2] = 30
last - first = 4 elements
first < last: true
after ++p, p += 2, --p: *p = 30, index = 2
&a[i] == a + i and a[i] == *(a + i) for every i: true
```

Empat baris pertama membenarkan Persamaan (6.1): satu langkah adalah 1 byte untuk `char`, 4 untuk `int`, 8 untuk `double`, dan 104 untuk `Book` kita. Bilangan terakhir itu bukan hukum alam. Sebuah objek `std::string` di pustaka C++ GNU berukuran 32 byte (ia menyimpan pointer ke karakter-karakternya, panjangnya, dan sebuah penyangga internal kecil), tiga di antaranya menjadi 96, `int` menambah 4, dan kompilator menggenapkan totalnya menjadi 104 agar `Book` berikutnya dalam larik dimulai di kelipatan 8. Kompilator lain bisa memberi ukuran lain; intinya, *Anda tidak pernah perlu mengetahuinya*, karena kompilator mengalikan dengan `sizeof(Book)` untuk Anda. Baris-baris selebihnya memeriksa Persamaan (6.2) (`last - first` bernilai 4 elemen, bukan 16 byte) dan menggerakkan `p` maju dan mundur.

6.3.4 Yang boleh dan yang tidak

Tabel 6.3 menghimpun operasi-operasinya. Segala yang tidak ada di tabel, seperti menjumlahkan dua pointer, mengalikan pointer dengan bilangan, atau mengurangkan pointer ke larik yang berbeda, ditolak kompilator atau tidak mempunyai makna yang terdefinisi.

Tabel 6.3 Operasi yang didefinisikan C++ pada pointer `p` bertipe `T*`.

Operasi	Contoh	Hasil	Syarat
Mengambil alamat	`&a`	sebuah `T*`	`a` objek bertipe `T`
Dereferensi	`*p`, `p->m`	objek yang ditunjuk `p`	`p` menunjuk objek yang masih hidup
Penugasan	`q = p`	`q` menyimpan alamat yang sama	tipe pointer sama
Membandingkan identitas	`p == q`, `p != nullptr`	`bool`	selalu boleh
Mengurutkan	`p < q`	`bool`	keduanya ke larik yang sama
Menggeser	`p + i`, `p - i`, `++p`, `--p`, `p += i`	sebuah `T*`	hasilnya tetap di dalam larik atau satu lewat ujungnya
Jarak	`q - p`	`std::ptrdiff_t`, dalam elemen	keduanya ke larik yang sama
Subskrip	`p[i]`	sama dengan `*(p + i)`	seperti menggeser dan dereferensi

PERINGATAN. "Perilaku tak terdefinisi" (*undefined behavior*) adalah istilah teknis dalam C++, dan maknanya lebih keras daripada kedengarannya. Istilah itu tidak berarti "program mogok" atau "hasilnya sampah". Artinya standar bahasa *sama sekali tidak menetapkan syarat apa pun* tentang apa yang terjadi: program boleh mogok, boleh mencetak jawaban salah yang tampak masuk akal, boleh bekerja sempurna hari ini lalu gagal setelah kompilator diperbarui. Kompilator bahkan boleh menganggap perilaku tak terdefinisi tidak pernah terjadi dan mengoptimasi berdasarkan anggapan itu. Setiap isian kolom "syarat" di **Tabel 6.3** yang dilanggar menghasilkan perilaku tak terdefinisi.

6.4 Pointer dan Larik

6.4.1 Nama larik menjadi pointer

Hampir di setiap ekspresi, nama sebuah larik otomatis dikonversi menjadi pointer ke elemen pertamanya. Konversi ini disebut *array-to-pointer decay* (peluruhan larik menjadi pointer). Jadi sesudah `int a[5];`, menulis `int* p = a;` persis sama dengan menulis `int* p = &a[0];`. Dua pengecualian yang perlu diketahui adalah `sizeof(a)`, yang memberi ukuran seluruh larik, dan `&a`, yang memberi pointer ke seluruh larik alih-alih ke elemen pertamanya.

Peluruhan, bersama Persamaan (6.1), menjelaskan arti sesungguhnya kurung siku. Standar C++ *mendefinisikan* subskrip melalui aritmetika pointer [1]:

$$a[i] \equiv *(a + i) \tag{6.3}$$

Nama larik meluruh menjadi pointer ke `a[0]`, pointer digeser i elemen, dan hasilnya didereferensi. Satu akibat yang lucu muncul dari sifat komutatif penjumlahan: $*(a + i)$ sama dengan $*(i + a)$, sehingga `i[a]` sah dalam C++ dan berarti `a[i]`. Tidak ada yang menulis begitu dalam kode sungguhan, tetapi melihatnya sekali, di akhir **Kode 6.4**, menghapus sisa-sisa perasaan bahwa subskrip larik adalah sihir.

6.4.2 Menyusuri larik dengan pointer

Pointer dapat menyusuri larik lima elemen dengan dua cara, melompat langsung dengan `p = a + k` dan melangkah dengan `p++`, lalu dipakai untuk menimpa setiap elemen. **Kode 6.4** melakukan semuanya dan diakhiri dengan loop C++ yang idiomatis, yang berhenti ketika pointer mencapai satu lewat ujung.

Kode 6.4 `04_array_walk.cpp`: melompat, melangkah, dan menulis melalui pointer, lalu loop satu-lewat-ujung.

```cpp
// Menelusuri array dengan pointer: baca, perbarui, baca lagi.
#include <iostream>

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    int* p = a;                             // p = &a[0]
    std::cout << "The first p = " << *p << '\n';
    for (int k = 1; k < 5; ++k) {
        p = a + k;                          // lompat langsung ke elemen k
        std::cout << "p = a + " << k << ", *p = " << *p << '\n';
    }

    p = a;                                  // kembali ke awal
    std::cout << "The value of a =";
    for (int i = 0; i < 5; ++i) {
        std::cout << ' ' << *p;
        ++p;                                // melangkah ke elemen berikutnya
    }

    p = &a[0];                              // kembali ke awal, cara tulis lain
    std::cout << "\nUpdate the items...\n";
    for (int i = 0; i < 5; ++i) {
        *p = i * 10;                        // tulis melalui pointer
        ++p;
    }

    std::cout << "The value of a =";
    for (int* q = a; q != a + 5; ++q) {     // berhenti di satu-lewat-ujung
        std::cout << ' ' << *q;
    }
    std::cout << "\n2[a] = " << 2[a] << " (the same as a[2] = " << a[2] << ")\n";
    return 0;
}
```

```
The first p = 1
p = a + 1, *p = 2
p = a + 2, *p = 3
p = a + 3, *p = 4
p = a + 4, *p = 5
The value of a = 1 2 3 4 5
Update the items...
The value of a = 0 10 20 30 40
2[a] = 20 (the same as a[2] = 20)
```

Perhatikan loop terakhir, `for (int* q = a; q != a + 5; ++q)`. Pointer mulai di elemen pertama dan berhenti ketika ia *sama dengan* `a + 5`, posisi tepat sesudah elemen terakhir. C++ menjamin bahwa pointer satu-lewat-ujung ini boleh dihitung dan dibandingkan untuk setiap larik, walaupun tidak pernah boleh didereferensi. Pasangan (elemen pertama, satu lewat elemen terakhir) melukiskan *rentang setengah terbuka*, ditulis [first,last): kurung siku mencakup ujung kiri dan kurung biasa mengecualikan ujung kanan. Rentang setengah terbuka adalah konvensi seluruh pustaka standar C++ (`v.begin()` dan `v.end()` adalah pasangan semacam itu), dan ia mempunyai dua sifat yang menyenangkan: banyaknya elemen cukup last − first, dan rentang kosong cukup first = last.

6.4.3 String adalah larik `char`, dan terminatornya penting

Literal string seperti `"Danaya"` disimpan sebagai larik `char` yang diakhiri satu byte tambahan bernilai nol, ditulis `'\0'` dan disebut *null terminator*. Jadi `char s[] = "Danaya";` membuat larik berisi *tujuh* karakter: enam huruf dan terminatornya. Fungsi seperti `std::strlen` menemukan panjangnya dengan berjalan sampai bertemu nol.

Sebuah contoh klasik sangat mendidik. Contoh itu menyusuri `"Danaya"` dengan `p++` enam kali, mencetaknya, lalu pada “Test 1” menjalankan loop yang sama *sekali lagi tanpa mengembalikan pointer ke awal*. Loop kedua mulai di terminator dan terus berjalan ke memori apa pun yang kebetulan berada sesudah larik. Di salah satu mesin tempat contoh ini dijalankan, ia mencetak sebuah `H` dan beberapa byte tak kasatmata; “Test 2” lalu melangkah mundur dengan `p--` melewati memori asing yang sama. Keluarannya tampak tidak berbahaya, dan itulah hal paling berbahaya yang dapat dilakukan perilaku tak terdefinisi. **Kode 6.5** mempertahankan bagian yang benar, yaitu berjalan maju sampai terminator dan mundur sampai awal, dan hanya mengulang kesalahan itu bila Anda memintanya dengan kata `overrun`.

Kode 6.5 `05_danaya.cpp`: menyusuri string C maju dan mundur dengan pointer; `overrun` mengulang kesalahan klasik.

```
// String adalah array char yang diakhiri '\0'. Penelusuran dengan pointer
// harus tetap di dalam array. Jalankan dengan kata "overrun" untuk mengulang
// kesalahan klasik: lupa mengembalikan pointer ke awal.
#include <cstring>
#include <iostream>
#include <string>

int main(int argc, char* argv[]) {
    char s[] = "Danaya";                    // 7 char: D a n a y a '\0'
    std::cout << "sizeof(s) = " << sizeof(s) << ", strlen(s) = " << std::strlen(s) << '\n';

    char* p = s;                            // p = &s[0]
    std::cout << "Forward:  ";
    while (*p != '\0') {                    // berhenti di terminator
        std::cout << *p;
        ++p;
    }
    std::cout << "\np now points at s[" << (p - s) << "], the terminator\n";

    std::cout << "Backward: ";
    while (p != s) {                        // mundur sampai char pertama
        --p;
        std::cout << *p;
    }
    std::cout << '\n';

    if (argc > 1 && std::string(argv[1]) == "overrun") {
        p = s + 6;                          // terminator, seperti setelah loop
        std::cout << "Overrun:  ";
        for (int i = 0; i < 6; ++i) {       // lupa mengembalikan p = s
            ++p;                            // s + 7, s + 8, ... bukan milik kita
            std::cout << *p;
        }
        std::cout << '\n';
    }
    return 0;
}
```

```
sizeof(s) = 7, strlen(s) = 6
Forward:  Danaya
p now points at s[6], the terminator
Backward: ayanaD
```

Bila dijalankan biasa, program ini benar. Sekarang bangunlah dengan AddressSanitizer, alat pemeriksa yang tertanam dalam GCC dan Clang dan akan kita pelajari sungguh-sungguh di Subbab 6.7, lalu mintalah *overrun*. Laporan di bawah ini asli tetapi dipersingkat: kita simpan baris-baris pertamanya dan kita potong baris yang panjang di sebelah kanan.

```
g++ -std=c++17 -g -fsanitize=address -o danaya 05_danaya.cpp
./danaya overrun
```

```
sizeof(s) = 7, strlen(s) = 6
Forward:  Danaya
p now points at s[6], the terminator
Backward: ayanaD
=================================================================
==23802==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7f67b7d00087 ...
READ of size 1 at 0x7f67b7d00087 thread T0
    #0 0x55e0eaca1a26 in main 05_danaya.cpp:32
```

Laporan itu menyebutkan jenis kesalahannya (*stack-buffer-overflow*: membaca melewati ujung larik yang tinggal di *stack*), apakah itu pembacaan atau penulisan, berapa byte, dan baris sumber yang tepat, baris 32, yaitu `std::cout << *p;` di dalam loop *overrun*. Pembacaan pertama di `s + 6` sebenarnya tidak berbahaya (itulah terminatornya); `++p` pertama dalam loop bergeser ke `s + 7`, satu lewat ujung, dan dereferensi pertama di sana langsung tertangkap. Tanpa *sanitizer*, program yang sama biasanya mencetak beberapa karakter aneh lalu selesai dengan normal, dan tepat begitulah bug semacam itu lolos tanpa disadari.

6.4.4 Larik sebagai parameter fungsi

Ketika larik dikirim ke fungsi, peluruhan terjadi saat pemanggilan: fungsi hanya menerima alamat elemen pertama, tidak pernah salinan elemen-elemennya. Itulah sebabnya fungsi seperti `fill` dapat mengisi larik milik pemanggil, dan itulah sebabnya setiap fungsi yang menerima larik memerlukan informasi kedua, panjangnya, dari suatu tempat. Artinya pula, parameter yang ditulis `int data[]` sama sekali bukan larik. Kompilator menuliskannya ulang menjadi `int* data`, dan GCC bahkan memperingatkan Anda bila meminta `sizeof`-nya:

```
sizeof_param.cpp: In function 'void fill(int*)':
sizeof_param.cpp:3:25: warning: 'sizeof' on array function parameter 'data' will return
size of 'int*' [-Wsizeof-array-argument]
    3 |     std::cout << sizeof(data) << '\n';
      |                         ~^~~~~
```

Kode 6.6 menunjukkan hal itu tanpa peringatan dengan menulis parameter secara jujur sebagai `int*`. Program itu juga menunjukkan tiga cara modern mengirim deretan data: pointer dan cacah, rentang setengah terbuka berupa dua pointer, dan kontainer standar yang dikirim melalui referensi `const`, yang membawa ukurannya sendiri.

Kode 6.6 `06_array_param.cpp`: argumen larik tiba sebagai pointer; cacah, rentang pointer, atau kontainer melengkapi panjang yang hilang.

```cpp
// Array yang dikirim ke fungsi tiba sebagai pointer ke elemen pertamanya.
#include <array>
#include <iostream>
#include <vector>

void fill(int* data, int n) {            // int data[] berarti persis int* data
    for (int i = 0; i < n; ++i) {
        data[i] = i + 1;                  // menulis ke array milik pemanggil
    }
    std::cout << "  inside fill: sizeof(data) = " << sizeof(data)
              << " (the size of a pointer)\n";
}

void show(const int* data, int n) {      // const: show berjanji tidak menulis
    for (int i = 0; i < n; ++i) {
        std::cout << data[i] << ' ';
    }
    std::cout << '\n';
}

int sum(const int* first, const int* last) {   // rentang setengah terbuka [first, last)
    int total = 0;
    for (const int* p = first; p != last; ++p) {
        total += *p;
    }
    return total;
}

int sum(const std::vector<int>& v) {     // vector tahu ukurannya sendiri
    int total = 0;
    for (int x : v) total += x;
    return total;
}

int main() {
    int data[10];
    std::cout << "Fill the data...\n";
    std::cout << "  in main: sizeof(data) = " << sizeof(data) << " (10 ints)\n";
    fill(data, 10);
    std::cout << "The data has been filled.\nShow the data...\n";
    show(data, 10);
    std::cout << "sum of all ten = " << sum(data, data + 10) << '\n';
    std::cout << "sum of data[2..4] = " << sum(data + 2, data + 5) << '\n';
```

Kode 6.6 (Lanjutan) *06_array_param.cpp: argumen larik tiba sebagai pointer; cacah, rentang pointer, atau kontainer melengkapi panjang yang hilang.*

```cpp
    std::array<int, 4> fixed = {5, 6, 7, 8};      // ukuran bagian dari tipe
    std::vector<int> grown = {1, 2, 3};
    grown.push_back(4);
    std::cout << "std::array sum = " << sum(fixed.data(), fixed.data() + fixed.size())
              << ", std::vector sum = " << sum(grown) << '\n';
    return 0;
}
```

```
Fill the data...
  in main: sizeof(data) = 40 (10 ints)
  inside fill: sizeof(data) = 8 (the size of a pointer)
The data has been filled.
Show the data...
1 2 3 4 5 6 7 8 9 10
sum of all ten = 55
sum of data[2..4] = 12
std::array sum = 26, std::vector sum = 10
```

Di `main`, `sizeof(data)` bernilai 40: sepuluh bilangan bulat empat byte. Di dalam `fill` nilainya 8, ukuran pointer, karena di dalam `fill` tidak ada larik, hanya alamat. Kata `const` pada `show(const int* data, int n)` adalah janji, yang diperiksa kompilator, bahwa `show` hanya membaca melalui pointer; Subbab 6.11 membahas janji semacam itu. Pemanggilan dua pointer `sum(data + 2, data + 5)` menjumlahkan tepat `data[2]`, `data[3]`, dan `data[4]`, yaitu $3 + 4 + 5 = 12$, yang menunjukkan betapa wajarnya rentang setengah terbuka melukiskan sebagian larik. C++20 menambahkan `std::span`, objek kecil yang membungkus pointer dan panjang menjadi satu; buku ini tetap memakai C++17, dengan pasangan pointer-dan-cacah atau pasangan dua pointer sebagai idiom bakunya.

6.4.5 Larik dua dimensi

Larik dua dimensi seperti `int grid[R][C]` disimpan sebagai R baris yang diletakkan berurutan, masing-masing berisi C elemen berurutan. Tata letak ini disebut *row-major order* (urutan utama-baris). Elemen `grid[i][j]` karenanya berada i baris utuh ditambah j elemen dari awal:

$$\text{alamat}(\text{grid}[i][j]) = \text{alamat}(\text{grid}[0][0]) + (i \cdot C + j) \cdot \text{sizeof}(T) \tag{6.4}$$

Persamaan (6.4) adalah Persamaan (6.1) yang diterapkan dua kali, sekali untuk baris dan sekali untuk kolom, dan rumus itu menjelaskan mengapa fungsi yang menerima larik dua dimensi harus diberi tahu C dalam tipe parameternya (`int g[][4]`): tanpa panjang baris, ia tidak dapat menghitung di mana baris ke-i dimulai. Untuk grid `int` berukuran 3×4 yang dimulai di alamat 1000, `grid[2][1]` tinggal di $1000 + (2 \cdot 4 + 1) \cdot 4 = 1036$.

6.5 Pointer ke Struktur dan Operator Panah

Pointer dapat menunjuk seluruh struktur semudah menunjuk sebuah `int`. Dengan `Book* bp = &catalog[2];`, ekspresi `*bp` adalah `Book` itu sendiri, sehingga judulnya adalah `(*bp).title`. Tanda kurung itu wajib: titik mengikat lebih kuat daripada bintang, sehingga `*bp.title` berarti `*(bp.title)`, yang mencoba memperlakukan `bp` sebagai struktur dan gagal dikompilasi. Karena menjangkau anggota melalui pointer sangat sering dilakukan, C++ menyediakan singkatan, yaitu *operator panah*:

> **DEFINISI.** Untuk pointer `p` ke sebuah struktur atau kelas, `p->m` berarti persis `(*p).m`: ikuti pointer, lalu pilih anggota `m`.

Kode 6.7 memakai panah dengan tiga cara: untuk membaca field, untuk mengubah field elemen larik milik pemanggil, dan untuk menyusuri seluruh katalog dengan `const Book*`. Program itu lalu membangun sesuatu yang baru: tiga objek `Node`, masing-masing berisi sebuah `Book` dan pointer ke simpul berikutnya, lalu mengikuti rantainya dari `head` sampai bertemu `nullptr`.

Kode 6.7 `07_book_pointer.cpp`: operator panah pada larik `Book`, dan rantai simpul yang pertama.

```cpp
// Pointer ke struktur: (*bp).field dan bp->field, pointer yang menelusuri
// array Book, dan rantai simpul pertama yang ditautkan pointer.
#include <iostream>
#include <string>

struct Book {
    std::string id, title, author;
    int year;
};

struct Node {
    Book data;
    Node* next;                          // alamat simpul berikutnya, atau nullptr
};

int main() {
    Book catalog[5] = {
        {"B4001", "Laskar Pelangi", "Andrea Hirata", 2005},
        {"B4002", "Filosofi Kopi", "Dee Lestari", 2006},
        {"B4003", "Bumi Manusia", "Pramoedya Ananta Toer", 1980},
        {"B4004", "Negeri 5 Menara", "Ahmad Fuadi", 2009},
        {"B4005", "Cantik Itu Luka", "Eka Kurniawan", 2002},
    };

    Book* bp = &catalog[2];
    std::cout << "catalog[2].title = " << catalog[2].title << '\n';
    std::cout << "(*bp).title      = " << (*bp).title << '\n';
    std::cout << "bp->title        = " << bp->title << '\n';

    bp->year = 1981;                     // menulis ke catalog[2] itu sendiri
    std::cout << "after bp->year = 1981, catalog[2].year = " << catalog[2].year << '\n';
    bp->year = 1980;

    std::cout << "\nWalking the catalog with a Book*:\n";
    for (const Book* p = catalog; p != catalog + 5; ++p) {
        std::cout << "  [" << (p - catalog) << "] " << p->id << "  "
                  << p->title << " (" << p->year << ")\n";
    }

    // Tiga simpul, masing-masing menunjuk simpul berikutnya: rantai, bukan
    // array.
    Node third  {catalog[4], nullptr};
    Node second {catalog[0], &third};
    Node first  {catalog[2], &second};
    Node* head = &first;
```

Kode 6.7 (Lanjutan) *07_book_pointer.cpp: operator panah pada larik Book, dan rantai simpul yang pertama.*

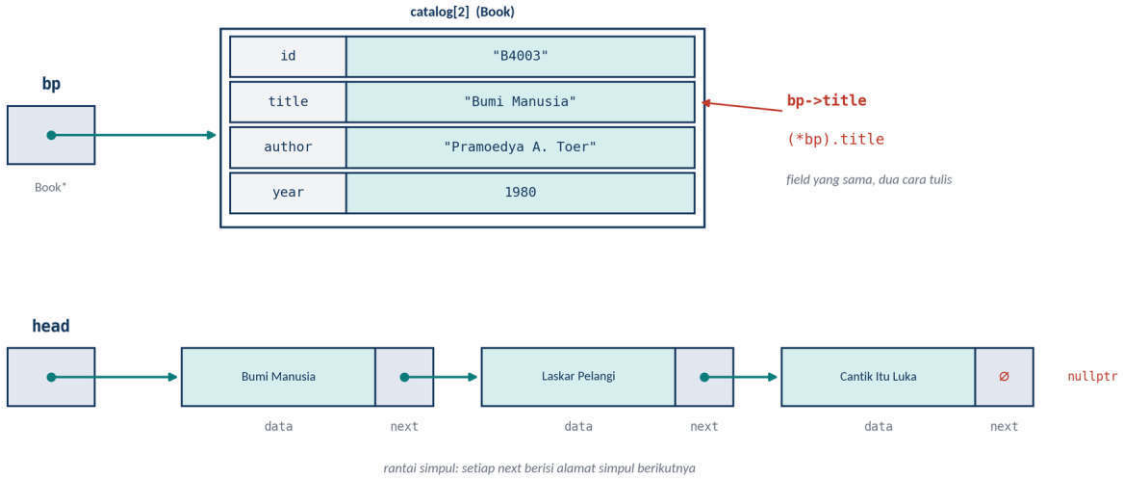
```cpp
    std::cout << "\nFollowing the chain from head:\n";
    for (Node* cur = head; cur != nullptr; cur = cur->next) {
        std::cout << "  " << cur->data.title << '\n';
    }
    std::cout << "head->next->next->data.title = "
              << head->next->next->data.title << '\n';
    return 0;
}
```

```
catalog[2].title = Bumi Manusia
(*bp).title      = Bumi Manusia
bp->title        = Bumi Manusia
after bp->year = 1981, catalog[2].year = 1981

Walking the catalog with a Book*:
  [0] B4001  Laskar Pelangi (2005)
  [1] B4002  Filosofi Kopi (2006)
  [2] B4003  Bumi Manusia (1980)
  [3] B4004  Negeri 5 Menara (2009)
  [4] B4005  Cantik Itu Luka (2002)

Following the chain from head:
  Bumi Manusia
  Laskar Pelangi
  Cantik Itu Luka
head->next->next->data.title = Cantik Itu Luka
```

Gambar 6.4 menggambar kedua bagiannya. Di atas, `bp` menyimpan alamat `catalog[2]`; `bp->title` dan `(*bp).title` adalah dua cara menulis satu field. Di bawah, `head` menunjuk sebuah simpul, yang `next`-nya menunjuk sebuah simpul, yang `next`-nya menunjuk sebuah simpul, yang `next`-nya `nullptr`. Bacalah `head->next->next->data.title` dari kiri ke kanan, satu panah demi satu panah: mulai di `head`, ikuti ke simpul pertama (*Bumi Manusia*), ambil `next` simpul itu dan ikuti ke simpul kedua (*Laskar Pelangi*), ambil `next`-*nya* dan ikuti ke simpul ketiga (*Cantik Itu Luka*), lalu baca judul pada `data` simpul tersebut.



Gambar 6.4 Atas: sebuah `Book*` dan dua cara menulis akses field. Bawah: tiga simpul yang ditautkan oleh pointer `next`-nya, bentuk *linked list* di Bab 7.

Perhatikan apa yang *tidak* dibutuhkan rantai itu. Ketiga simpul tidak berada dalam larik dan tidak berurutan dengan cara tertentu di memori; dalam program, ketiganya adalah tiga variabel lokal terpisah yang dideklarasikan dalam urutan terbalik. Urutan daftar sepenuhnya tinggal di pointer `next`. Ubahlah satu pointer dan urutannya berubah, tanpa memindahkan satu `Book` pun. Pengamatan itu adalah seluruh gagasan *linked list*, dan Bab 7 membangun di atasnya. Yang ditambahkan Bab 7 adalah kemampuan membuat simpul ketika program berjalan, sebanyak yang diperlukan, dan itulah pokok subbab berikutnya.

Loop `for (Node* cur = head; cur != nullptr; cur = cur->next)` layak dihafalkan, karena Anda akan menulisnya ratusan kali: mulai di *head*, berhenti di null, dan maju dengan mengikuti `next`. Bandingkan dengan loop larik `for (int* q = a; q != a + 5; ++q)`: strukturnya identik, hanya dua gagasan “ujung” dan “berikutnya” yang berbeda.

6.6 Memori Dinamis: `new` dan `delete`

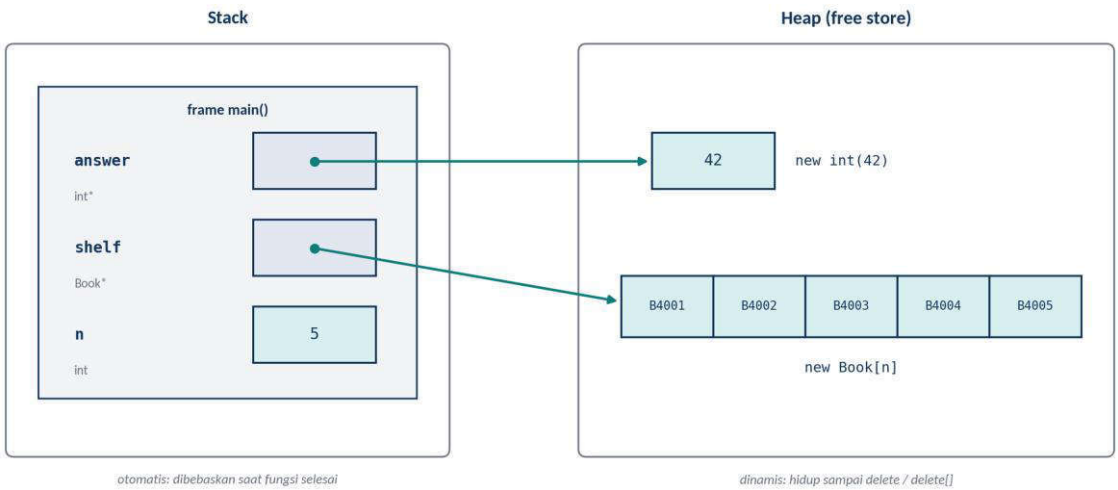
6.6.1 Dua tempat tinggal: *stack* dan *heap*

Setiap variabel yang kita deklarasikan sejauh ini bersifat *otomatis*: ia mulai ada ketika eksekusi mencapai deklarasinya dan lenyap ketika eksekusi meninggalkan blok yang memuatnya. Mesin menyimpan variabel otomatis di wilayah yang disebut *stack*, dan nama itu tepat. Setiap pemanggilan fungsi menumpuk sebuah *frame* berisi parameter dan variabel lokal pemanggilan itu di puncak; setiap *return* mengambilnya kembali. Tumpukan piring di Bab 5 benar-benar cara pemanggilan fungsi dikelola. Penyimpanan otomatis cepat dan tidak menuntut pembukuan apa pun dari pemrogram, tetapi mempunyai dua batas yang keras. Ukuran setiap variabel otomatis harus diketahui ketika program dikompilasi, dan tidak ada yang tinggal di *stack* dapat hidup lebih lama daripada fungsi yang membuatnya.

Struktur data melanggar kedua batas itu setiap saat. Sebuah perpustakaan tidak tahu pada hari Senin berapa buku yang akan dimilikinya pada hari Jumat. Fungsi yang membuat simpul *linked list* baru harus menyerahkan simpul itu kepada pemanggilnya, dan simpul itu harus tetap ada setelah fungsi selesai. Untuk objek semacam itu C++ menyediakan *penyimpanan dinamis*, yang diambil dari wilayah kedua bernama *free store* atau, lebih lazim, *heap*. (“Heap” di sini tidak ada hubungannya dengan struktur data *heap* di Bab 14; kedua kata itu hanya kebetulan bertabrakan puluhan tahun lalu.)

> **DEFINISI.** Objek yang *dialokasikan secara dinamis* dibuat oleh ekspresi `new T` (atau `new T[n]` untuk larik) ketika program berjalan. Objek itu hidup sampai program menghancurkannya secara eksplisit dengan `delete` (atau `delete[]`), tidak peduli fungsi mana yang membuatnya. Ekspresi `new` mengembalikan pointer ke objek itu; pointer tersebut satu-satunya jalan untuk menjangkaunya.

Gambar 6.5 memperlihatkan keadaan `main` dalam program yang akan kita jalankan, kira-kira di tengah jalan. *Frame stack* berisi tiga variabel: dua pointer dan sebuah bilangan bulat. *Heap* berisi objek-objek yang ditunjuk keduanya: sebuah `int` bernilai 42 dan larik lima `Book`. Pointer-pointer itu adalah variabel otomatis biasa dan akan lenyap ketika `main` selesai; objek di *heap* tidak akan lenyap kecuali ada yang memanggil `delete`.



Gambar 6.5 Variabel otomatis tinggal di *frame stack* fungsinya; objek yang dibuat dengan `new` tinggal di *heap* dan hanya dapat dijangkau melalui pointer.

6.6.2 `new`, `delete`, `new[]`, dan `delete[]`

Aturan memakai memori dinamis muat dalam satu baris dan dilanggar di program yang tak terhitung banyaknya: *setiap `new` berpasangan dengan tepat satu `delete`, dan setiap `new[]` dengan tepat satu `delete[]`*. **Kode 6.8** menaatinya untuk sebuah `int`, sebuah `Book`, dan larik `Book` yang ukurannya berupa variabel. Lalu program itu membangun sesuatu yang lebih menarik.

Kode 6.8 `08_dynamic.cpp`: objek tunggal dan larik di *heap*, serta larik tumbuh yang menggandakan kapasitasnya.

```
// Memori dinamis: new/delete untuk satu objek, new[]/delete[] untuk array,
// dan array tumbuh yang menggandakan kapasitasnya.
#include <iostream>
#include <string>

struct Book {
    std::string id, title;
    int year;
};

// Array int tumbuh yang minimal, dikelola secara manual.
struct IntArray {
    int* data = nullptr;
    int size = 0;
    int capacity = 0;
    long copies = 0;                    // elemen yang disalin selama tumbuh

    void push(int value) {
        if (size == capacity) {
            int newCapacity = (capacity == 0) ? 1 : 2 * capacity;
            int* bigger = new int[newCapacity];
            for (int i = 0; i < size; ++i) {
                bigger[i] = data[i];    // salin elemen lama ke blok baru
            }
            copies += size;
            delete[] data;              // kembalikan blok lama
            data = bigger;
            capacity = newCapacity;
            std::cout << "  grow: capacity " << capacity << ", copied " << size << '\n';
        }
        data[size++] = value;
    }

    ~IntArray() { delete[] data; }      // blok terakhir juga dikembalikan
};

int main() {
    int* answer = new int(42);          // satu int di heap
    std::cout << "*answer = " << *answer << '\n';
    delete answer;                      // satu new, satu delete
    answer = nullptr;

    Book* book = new Book{"B4003", "Bumi Manusia", 1980};
    std::cout << "book->title = " << book->title << '\n';
    delete book;
```

Kode 6.8 (Lanjutan) *08_dynamic.cpp: objek tunggal dan larik di heap, serta larik tumbuh yang menggandakan kapasitasnya.*

```
    int n = 5;                              // baru diketahui saat program berjalan
    Book* shelf = new Book[n];              // n Book yang dikonstruksi bawaan
    for (int i = 0; i < n; ++i) {
        shelf[i].id = "B400" + std::to_string(i + 1);
        shelf[i].year = 2000 + i;
    }
    std::cout << "shelf[4].id = " << shelf[4].id << ", year " << shelf[4].year << '\n';
    delete[] shelf;                         // new[] berpasangan dengan delete[]

    IntArray arr;
    const int count = 17;
    for (int v = 1; v <= count; ++v) {
        arr.push(v * v);
    }
    std::cout << "size " << arr.size << ", capacity " << arr.capacity
              << ", last " << arr.data[arr.size - 1]
              << ", total copies " << arr.copies << " (< 2 * " << count << " = "
              << 2 * count << ")\n";
    return 0;
}
```

```
*answer = 42
book->title = Bumi Manusia
shelf[4].id = B4005, year 2004
  grow: capacity 1, copied 0
  grow: capacity 2, copied 1
  grow: capacity 4, copied 2
  grow: capacity 8, copied 4
  grow: capacity 16, copied 8
  grow: capacity 32, copied 16
size 17, capacity 32, last 289, total copies 31 (< 2 * 17 = 34)
```

Empat rincian layak diperhatikan.

- `new int(42)` membuat sebuah `int` yang diinisialisasi dengan 42; `new Book{"B4003", "Bumi Manusia", 1980}` membuat sebuah `Book` dengan field-field yang diinisialisasi berurutan. Keduanya mengembalikan pointer.
- `new Book[n]` menerima `n` yang baru diketahui saat program berjalan, sesuatu yang tidak diizinkan untuk deklarasi larik biasa dalam C++ standar. Masing-masing dari n `Book` dikonstruksi secara bawaan: string kosong dan `year` yang nilainya tak tentu, dan itulah sebabnya loop mengisinya sebelum dibaca.
- Sesudah `delete answer;` program menjalankan `answer = nullptr;`. `delete` menghancurkan `int` di *heap* tetapi tidak berbuat apa-apa terhadap variabel pointernya, yang masih menyimpan alamat lama. Mengisinya dengan null sesudahnya adalah kebiasaan murah yang mengubah pemakaian kedua yang tak disengaja menjadi kegagalan yang jelas alih-alih kerusakan yang diam-diam; Subbab 6.7 menunjukkan alasannya.
- `IntArray` memiliki sebuah larik di *heap*. *Destruktor*-nya, fungsi anggota `~IntArray()` yang dipanggil C++ secara otomatis ketika sebuah `IntArray` keluar dari cakupan, menjalankan `delete[]` yang berpasangan. Pola itu, sumber daya di *heap* yang dilepaskan oleh destruktor objek pemiliknya, adalah benih *smart pointer* di Subbab 6.13.

6.6.3 Larik tumbuh dan aturan penggandaan

`IntArray` di **Kode 6.8** memecahkan masalah ukuran tetap dari Bab 5. Ia menyimpan blok *heap* berkapasitas tertentu, mengisinya, dan ketika penuh meminta blok baru berkapasitas dua kali

lipat, menyalin elemen-elemen lama ke sana, lalu mengembalikan blok lama. **Pseudokode 6.1** menyatakan prosedurnya dalam pseudokode.

Pseudokode 6.1 `DYNAMIC-ARRAY-PUSH`: menambahkan `x`, setelah lebih dahulu menggandakan kapasitas bila larik penuh.

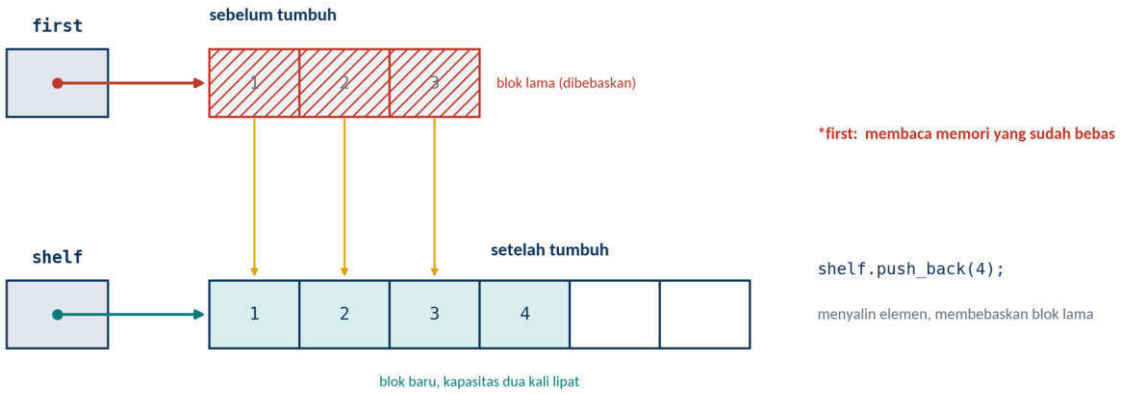
```
DYNAMIC-ARRAY-PUSH(A, x)
1  if A.size == A.capacity
2      if A.capacity == 0
3          newCap = 1
4      else newCap = 2 · A.capacity
5      B = allocate an array of newCap elements
6      for i = 0 to A.size − 1
7          B[i] = A.data[i]
8      free A.data
9      A.data = B
10     A.capacity = newCap
11 A.data[A.size] = x
12 A.size = A.size + 1
```

Penggandaan tampak boros, karena larik penuh berisi sejuta elemen tiba-tiba meminta ruang untuk dua juta. Justru penggandaan itulah alasan struktur ini cepat. Hitunglah salinan elemen yang terjadi selama mendorong n nilai. Pertumbuhan terjadi ketika ukurannya mencapai 1,2,4,8, …, sampai pangkat dua terbesar di bawah n, katakanlah 2^k dengan $2^k < n$. Setiap pertumbuhan menyalin semua elemen yang ada, sehingga total salinannya adalah

$$C(n) = 1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 < 2n \tag{6.5}$$

Langkah tengah adalah jumlah deret geometri: setiap suku berlipat dua, sehingga jumlahnya satu kurang dari pangkat dua berikutnya. Langkah terakhir memakai $2^k < n$, sehingga $2^{k+1} < 2n$. Persamaan (6.5) mengatakan bahwa mendorong n nilai memerlukan total kurang dari $2n$ salinan, kurang dari dua salinan *per dorongan secara rata-rata*, walaupun satu dorongan bisa saja menyalin segalanya. Keluaran **Kode 6.8** memeriksanya untuk $n = 17$: pertumbuhan pada ukuran 1,2,4,8,16, total 31 salinan, di bawah $2 \cdot 17 = 34$. Biaya rata-rata per operasi yang berlaku untuk sembarang deret operasi disebut biaya *teramortisasi*, dan dorongan pada larik yang menggandakan diri berbiaya $O(1)$ teramortisasi; teorinya yang lengkap ada di Cormen dkk. [3]. Tumbuh dengan jumlah *tetap*, misalnya $+10$ setiap kali, akan membuat totalnya $\Theta(n^2)$, dan karena itulah setiap pustaka yang serius, termasuk `std::vector`, tumbuh dengan *faktor* konstan.

Gambar 6.6 memperlihatkan satu langkah pertumbuhan dan menambahkan peringatan yang akan muncul lagi di Subbab 6.7: pointer mana pun yang menunjuk ke dalam blok *lama* masih menyimpan alamat lama sesudah pertumbuhan, padahal blok lama sudah dibebaskan.



Gambar 6.6 Satu langkah pertumbuhan larik yang menggandakan diri. Elemen-elemen disalin ke blok baru dan blok lama dibebaskan, sehingga pointer yang diambil sebelum pertumbuhan kini menunjuk ke memori yang sudah bebas.

6.6.4 Ketika `new` gagal

Memori itu terbatas. Bila `new` tidak menemukan memori bebas yang cukup, ia tidak mengembalikan pointer null; ia *melempar* eksepsi bertipe `std::bad_alloc`, yang mengakhiri program kecuali ada yang menangkapnya. Bila Anda lebih suka hasil null, bentuk `new (std::nothrow) T` mengembalikan `nullptr` saat gagal, dan Anda wajib memeriksanya. Kebanyakan program membiarkan `bad_alloc` merambat, karena mesin yang benar-benar kehabisan memori jarang mempunyai cara yang masuk akal untuk melanjutkan.

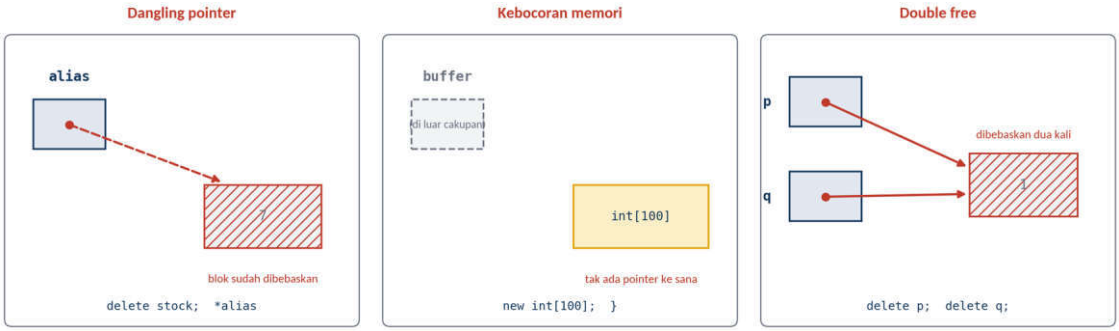
6.7 Ketika Pointer Salah Jalan

6.7.1 Lima bug, satu akar masalah

Ingat baris terakhir **Tabel 6.2**: pointer dan objek yang ditunjuknya mempunyai umur masing-masing. Setiap bug pointer klasik adalah ketidaksepakatan di antara keduanya.

- **Dangling pointer** (pointer menggantung) masih menyimpan alamat objek yang sudah tidak ada: objek itu telah di-`delete`, atau berupa variabel lokal fungsi yang sudah selesai, atau berupa elemen kontainer yang sejak itu memindahkan penyimpanannya. Memakainya disebut *use after free*.
- **Kebocoran memori** (*memory leak*) adalah ketidaksepakatan sebaliknya: objeknya masih ada tetapi tidak ada lagi pointer yang menunjuknya, sehingga program tidak akan pernah dapat menghapusnya. Setiap blok yang bocor hilang sampai program berakhir.
- **Double free** menghapus blok yang sama dua kali, biasanya melalui dua pointer yang merupakan salinan satu sama lain.
- **`delete` yang tidak berpasangan** membebaskan larik yang dibuat dengan `new[]` memakai `delete` biasa, atau objek tunggal dengan `delete[]`. Kedua bentuk itu memakai pembukuan yang berbeda, dan mencampurnya adalah perilaku tak terdefinisi.
- **Dereferensi null** mengikuti pointer yang berisi `nullptr`, biasanya karena fungsi yang "mungkin tidak menemukan apa-apa" dipercaya selalu menemukan sesuatu.

Ke dalam lima bug ini tambahkan *akses di luar batas* yang sudah kita temui pada `"Danaya"`: pointer yang berjalan melampaui lariknya. **Gambar 6.7** menggambar tiga yang pertama dengan konvensi kotak dan panah. Dalam setiap kasus gambar itu membuat bugnya langsung terlihat: panah ke dalam blok yang diarsir (sudah bebas), blok tanpa satu panah pun yang masuk, dan dua panah ke dalam blok yang dibebaskan dua kali.



Gambar 6.7 Tiga bug umur objek yang digambar sebagai kotak dan panah. Blok yang diarsir sudah dibebaskan.

Semua ini bukan keanehan akademis. Daftar MITRE tahun 2025 tentang 25 kelemahan perangkat lunak paling berbahaya, yang disusun dari catatan kerentanan yang baru dipublikasikan, memuat enam entri yang persis merupakan bug dalam subbab ini [4]. **Tabel 6.4** mendaftarnya beserta peringkat 2025-nya.

Tabel 6.4 Kelemahan keamanan memori dalam 2025 CWE Top 25 Most Dangerous Software Weaknesses, yang diterbitkan MITRE pada Desember 2025 [4].

Peringkat 2025	Kelemahan (pengenal CWE)	Dalam bab ini
5	Out-of-bounds Write (CWE-787)	`overflow` di **Kode 6.9**
7	Use After Free (CWE-416)	`dangling`, `stale` di **Kode 6.9**
8	Out-of-bounds Read (CWE-125)	`overrun` di **Kode 6.5**
11	Buffer Copy without Checking Size of Input (CWE-120)	alasan perlunya pemeriksaan panjang ala `strlen`
13	NULL Pointer Dereference (CWE-476)	`null` di **Kode 6.9**
16	Heap-based Buffer Overflow (CWE-122)	`overflow` di **Kode 6.9**

6.7.2 Bug-bug itu, dengan sengaja

Kode 6.9 melakukan setiap bug dalam fungsi kecilnya sendiri, yang dipilih dengan sebuah kata di baris perintah. Dijalankan tanpa kata, program itu mencetak menu dan menjalankan versi-versi yang benar, dan itulah keluaran yang ditampilkan. Dua di antara fungsinya menyimpan pointer di dalam `std::vector<int*>`. Itu bukan cara yang wajar untuk menulisnya, dan alasannya justru menarik: bila `new`, `delete`, dan pemakaian yang salah berdiri gamblang dalam tiga baris berurutan, GCC 13 menyadarinya dan memberi peringatan (`-Wuse-after-free`, `-Wmismatched-new-delete`), sedangkan kode buku ini harus dikompilasi tanpa peringatan. Bug sungguhan jarang sebaik itu. Kedua pointer biasanya tinggal di fungsi yang berbeda atau di struktur data yang berbeda, dan kompilator, yang memeriksa satu fungsi pada satu waktu, tidak dapat melihatnya. Di sini vector memainkan peran itu.

Kode 6.9 `09_mem_bugs.cpp`: tujuh bug pointer klasik, masing-masing dipilih dengan kata di baris perintah, beserta bentuk yang benar.

```cpp
// Tujuh bug pointer klasik, satu untuk tiap kata di baris perintah. Jalankan
// tanpa kata untuk melihat menu dan versi yang benar; jalankan di bawah
// AddressSanitizer atau valgrind untuk melihat setiap bug tertangkap.
#include <cstring>
#include <iostream>
#include <string>
#include <vector>

void useAfterDelete() {
    std::vector<int*> owners;           // dua catatan untuk blok yang sama
    owners.push_back(new int(7));
    owners.push_back(owners[0]);        // pointer kedua, alamat sama
    delete owners[0];                   // blok dikembalikan ...
    std::cout << "stock is " << *owners[1] << '\n';  // ... tetapi masih dipakai
}

void staleAfterGrowth() {
    std::vector<int> shelf = {1, 2, 3};
    int* first = &shelf[0];             // menunjuk ke dalam blok milik vector
    shelf.push_back(4);                 // bisa memindahkan semua elemen
    std::cout << "first is " << *first << '\n';
}

void leak(int times) {
    for (int i = 0; i < times; ++i) {
        int* buffer = new int[100];     // 400 byte setiap kali
        buffer[0] = i;
    }                                   // buffer keluar cakupan: blok hilang
    std::cout << "leaked " << times << " blocks\n";
}

void doubleFree() {
    int* p = new int(1);
    int* q = p;
    delete p;
    delete q;                           // blok yang sama, untuk kedua kalinya
}

void mismatch() {
    std::vector<int*> rows;
    rows.push_back(new int[4]);         // dialokasikan dengan new[] ...
    rows[0][0] = 1;
    delete rows[0];                     // ... tetapi dibebaskan dengan delete, bukan
delete[]
}
```

Kode 6.9 (Lanjutan) *09_mem_bugs.cpp: tujuh bug pointer klasik, masing-masing dipilih dengan kata di baris perintah, beserta bentuk yang benar.*

```
void overflow(int n) {
    int* marks = new int[n];
    for (int i = 0; i <= n; ++i) {      // <= : kelebihan satu langkah
        marks[i] = i;
    }
    std::cout << "last mark " << marks[n - 1] << '\n';
    delete[] marks;
}

void nullDeref(const char* id) {
    int* found = nullptr;               // "tidak ditemukan"
    if (std::strcmp(id, "B4001") == 0) {
        found = new int(2005);
    }
    std::cout << "year " << *found << '\n';  // tanpa pemeriksaan sebelum *
    delete found;
}

void correct() {
    int* stock = new int(7);
    std::cout << "stock is " << *stock << '\n';
    delete stock;
    stock = nullptr;                    // tidak meninggalkan dangling pointer

    std::vector<int> shelf = {1, 2, 3};
    shelf.push_back(4);
    int* first = &shelf[0];             // diambil SETELAH pertumbuhan terakhir
    std::cout << "first is " << *first << '\n';

    int n = 5;
    int* marks = new int[n];
    for (int i = 0; i < n; ++i) marks[i] = i;
    std::cout << "last mark " << marks[n - 1] << '\n';
    delete[] marks;                     // new[] berpasangan dengan delete[]
}
```

Kode 6.9 (Lanjutan) *09_mem_bugs.cpp: tujuh bug pointer klasik, masing-masing dipilih dengan kata di baris perintah, beserta bentuk yang benar.*

```
int main(int argc, char* argv[]) {
    std::string bug = argc > 1 ? argv[1] : "";
    if (bug == "dangling")      useAfterDelete();
    else if (bug == "stale")    staleAfterGrowth();
    else if (bug == "leak")     leak(3);
    else if (bug == "double")   doubleFree();
    else if (bug == "mismatch") mismatch();
    else if (bug == "overflow") overflow(5);
    else if (bug == "null")     nullDeref("B9999");
    else {
        std::cout << "usage: mem_bugs dangling|stale|leak|double|mismatch|overflow|null\n";
        std::cout << "no bug chosen; the correct versions:\n";
        correct();
    }
    return 0;
}
```

```
usage: mem_bugs dangling|stale|leak|double|mismatch|overflow|null
no bug chosen; the correct versions:
stock is 7
first is 1
last mark 4
```

Bila dikompilasi biasa lalu dijalankan dengan kata bug, sebagian besar fungsi ini *tampak bekerja*. Di mesin uji kami `./mem_bugs dangling` mencetak `stock is 7`, karena blok yang sudah bebas belum dipakai ulang; `./mem_bugs leak` mencetak barisnya dan selesai dengan normal; `./mem_bugs overflow` menulis melewati larik lalu mencetak `last mark 4`. Itulah bagian yang licik: programnya salah, keluarannya benar, dan kerusakannya tak terlihat sampai perubahan lain yang tak berkaitan menggeser susunan memori. Kita memerlukan alat yang mengawasi setiap akses.

6.7.3 AddressSanitizer

AddressSanitizer (ASan), yang diperkenalkan Serebryany dan rekan-rekannya di Google [5], tertanam dalam GCC dan Clang. Ketika program dikompilasi dengan `-fsanitize=address`, kompilator mengelilingi setiap blok *heap*, setiap larik di *stack*, dan setiap larik global dengan *zona merah* kecil berisi byte terlarang, mencatat dalam *shadow memory* yang ringkas byte mana yang saat ini sah, dan menyisipkan pemeriksaan sebelum setiap pembacaan dan penulisan. Blok yang dibebaskan tidak segera dipakai ulang melainkan disimpan dalam *karantina*, sehingga *use after free* menyentuh memori yang diketahui sudah mati. Program berjalan kira-kira dua kali lebih lambat, harga yang murah selama pengembangan. Tambahkan `-g` agar laporan menyebutkan baris sumber:

```
g++ -std=c++17 -g -fsanitize=address -o mem_bugs 09_mem_bugs.cpp
./mem_bugs dangling
```

Berikut baris-baris pertama laporan untuk empat bug, dari eksekusi sungguhan; setiap laporan berlanjut dengan lebih banyak *stack frame* yang kami hilangkan, dan alamat heksadesimal yang panjang berbeda pada setiap eksekusi.

```
==23665==ERROR: AddressSanitizer: heap-use-after-free on address 0x502000000010 ...
READ of size 4 at 0x502000000010 thread T0
    #0 0x55bc1c95b8eb in useAfterDelete() 09_mem_bugs.cpp:14
    #1 0x55bc1c95cb8a in main 09_mem_bugs.cpp:84
0x502000000010 is located 0 bytes inside of 4-byte region [0x502000000010,0x502000000014)
freed by thread T0 here:
    #0 0x7f30524ff5e8 in operator delete(void*, unsigned long)
    #1 0x55bc1c95b864 in useAfterDelete() 09_mem_bugs.cpp:13
```

Bacalah laporan itu seperti laporan seorang detektif. *Apa*: *use after free* di *heap*. *Akses yang mana*: pembacaan empat byte, sebuah `int`. *Di mana*: baris 14, `std::cout` yang mendereferensi `owners[1]`. *Apa yang ada di sana*: blok 4 byte, dan alamatnya berada 0 byte di dalamnya. *Siapa yang membebaskannya*: `delete` di baris 13. Setiap fakta yang diperlukan untuk memperbaiki bug itu tersedia.

```
==23668==ERROR: AddressSanitizer: heap-use-after-free on address 0x502000000010 ...
READ of size 4 at 0x502000000010 thread T0
    #0 0x56342fa1bcb3 in staleAfterGrowth() 09_mem_bugs.cpp:21
0x502000000010 is located 0 bytes inside of 12-byte region [0x502000000010,0x50200000001c)
freed by thread T0 here:
    #0 0x7f128daff5e8 in operator delete(void*, unsigned long)
    #1 0x56342fa21460 in std::__new_allocator<int>::deallocate(int*, unsigned long)
    ...
    #4 0x56342fa2016e in void std::vector<int, std::allocator<int>
>::_M_realloc_insert<int>(...)
```

Laporan `stale` inilah yang patut diingat dari bab ini. Tidak ada yang memanggil `delete`. Wilayah 12 byte itu adalah blok asli vector berisi tiga `int`, dan blok itu dibebaskan *di dalam* `push_back`, oleh realokasi yang digambar di **Gambar 6.6**. Pointer, referensi, atau iterator ke dalam `std::vector` menjadi tidak sah setiap kali vector itu tumbuh. Aturan ini akan penting setiap kali

bab-bab berikutnya menyimpan alamat elemen yang tinggal di dalam kontainer yang sedang tumbuh.

```
==23675==ERROR: AddressSanitizer: attempting double-free on 0x502000000010 in thread T0:
    #0 0x7f920a2ff5e8 in operator delete(void*, unsigned long)
    #1 0x55c2a4b5df2a in doubleFree() 09_mem_bugs.cpp:36
==23678==ERROR: AddressSanitizer: alloc-dealloc-mismatch (operator new [] vs operator
delete) on 0x502000000010
    #1 0x55b14b3220ee in mismatch() 09_mem_bugs.cpp:43
==23681==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x503000000054 ...
WRITE of size 4 at 0x503000000054 thread T0
    #0 0x5617ff2ce230 in overflow(int) 09_mem_bugs.cpp:49
0x503000000054 is located 0 bytes after 20-byte region [0x503000000040,0x503000000054)
```

Double free, ketidakcocokan `delete`, dan *overflow* masing-masing tertangkap tepat di barisnya. Laporan *overflow* sangat presisi: penulisan terjadi *0 byte sesudah* wilayah 20 byte, yaitu lima `int`, sehingga loop menulis `marks[5]`, satu elemen lewat ujung, persis kelebihan satu langkah akibat `i <= n`. Untuk `null`, ASan melaporkan `SEGV on unknown address 0x000000000000` dengan petunjuk `address points to the zero page`, dan untuk `leak`, *LeakSanitizer* yang tertanam dalam ASan melapor ketika program berakhir:

```
leaked 3 blocks
==23797==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 1200 byte(s) in 3 object(s) allocated from:
    #0 0x7f2e41afe6c8 in operator new[](unsigned long)
    #1 0x5600bf86ee0e in leak(int) 09_mem_bugs.cpp:26
SUMMARY: AddressSanitizer: 1200 byte(s) leaked in 3 allocation(s).
```

Tiga blok berisi 100 bilangan bulat empat byte adalah 1200 byte, dialokasikan di baris 26, dan tidak pernah dibebaskan. Menambahkan `-fsanitize=undefined` di sampingnya (`-fsanitize=address,undefined`) sekaligus menyalakan *UndefinedBehaviorSanitizer*, yang melaporkan kesalahan lain seperti *overflow* bilangan bulat bertanda dan akses anggota yang tidak sejajar atau null.

6.7.4 Valgrind

Valgrind menempuh pendekatan sebaliknya: alih-alih mengompilasi ulang, ia menjalankan program yang tidak diubah pada prosesor simulasi dan memeriksa setiap akses memori saat terjadi [6]. Alat bawaannya, *Memcheck*, tidak memerlukan flag kompilator khusus (walaupun `-g` tetap memberi nomor baris), menjalankan program dua puluh sampai lima puluh kali lebih lambat, dan sangat baik dalam melaporkan kebocoran. Rilis terbarunya adalah Valgrind 3.27.1 bulan Mei 2026 [7]; eksekusi di bawah ini memakai paket 3.22 dari Ubuntu 24.04, yang laporannya berbentuk sama.

```
g++ -std=c++17 -g -O0 -o mem_bugs_vg 09_mem_bugs.cpp
valgrind --leak-check=full ./mem_bugs_vg leak
```

```
leaked 3 blocks
==23744== HEAP SUMMARY:
==23744==     in use at exit: 1,200 bytes in 3 blocks
==23744==   total heap usage: 5 allocs, 2 frees, 79,024 bytes allocated
==23744==
==23744== 1,200 bytes in 3 blocks are definitely lost in loss record 1 of 1
==23744==    at 0x48485C3: operator new[](unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==23744==    by 0x10A7D4: leak(int) (09_mem_bugs.cpp:26)
==23744==    by 0x10AE48: main (09_mem_bugs.cpp:86)
==23744==
==23744== LEAK SUMMARY:
==23744==    definitely lost: 1,200 bytes in 3 blocks
```

"Definitely lost" adalah sebutan Valgrind untuk blok yang tidak lagi dijangkau pointer mana pun dalam program: kebocoran sejati. Untuk *dangling pointer*, Valgrind melaporkan `Invalid read of size 4`, lalu `Address 0x4e26080 is 0 bytes inside a block of size 4 free'd`, beserta baris `delete` dan baris `new` aslinya. Yang terpenting, alat-alat ini juga meneguhkan kode yang *baik*. Setiap program dalam bab ini selain bug yang disengaja telah dijalankan di bawah kedua alat; misalnya, vonis Valgrind atas **Kode 6.8** adalah

```
==23823==   total heap usage: 10 allocs, 10 frees, 78,516 bytes allocated
==23823== All heap blocks were freed -- no leaks are possible
==23823== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

dan AddressSanitizer diam untuk semuanya. (Alokasi di luar milik kita berasal dari pustaka *runtime* C++, yang juga membebaskannya.)

6.7.5 Alat yang mana, di komputer yang mana

Tabel 6.5 merangkum ketersediaannya. Aturan paling sederhana: bangunlah dengan `-g -fsanitize=address,undefined` setiap kali Anda mengembangkan kode pointer, dan gunakan Valgrind bila Anda tidak dapat mengompilasi ulang atau menginginkan laporan kebocoran yang paling rinci.

Tabel 6.5 Perbandingan AddressSanitizer dan Valgrind, per September 2026 [5], [6], [7], [8].

	AddressSanitizer	Valgrind (Memcheck)
Cara kerja	kompilator menyisipkan pemeriksaan; bangun ulang dengan `-fsanitize=address`	menjalankan biner yang tidak diubah pada CPU simulasi
Perlambatan	sekitar 2×	sekitar 20–50×
Linux	GCC dan Clang	ya
macOS	Apple Clang (`clang++ -fsanitize=address`)	tidak pada Apple silicon
Windows	Microsoft C++ (`cl /fsanitize=address`) atau Clang; tidak ada pada g++ MinGW dari MSYS2	tidak (gunakan WSL, lingkungan Linux di dalam Windows)
Paling unggul untuk	akses di luar batas dan *use after free*, dengan baris yang tepat	laporan kebocoran; memeriksa program yang tidak dapat dibangun ulang

Bahasanya sendiri pun bergerak ke arah ini. Standar C++26, yang isi teknisnya rampung pada Maret 2026, menghapus perilaku tak terdefinisi dari pembacaan variabel lokal yang belum diinisialisasi (pembacaan itu kini berstatus *erroneous behavior*, dengan nilai yang terdefinisi walaupun tidak berguna) dan menambahkan pustaka standar yang *diperkeras* (*hardened*), yang kontainernya memeriksa batas [9], [10]; GCC 16.1, yang dirilis April 2026, mengimplementasikan sebagian besar standar baru itu, termasuk pengerasan pustaka, dan merupakan rilis GCC pertama yang mengompilasi C++20 secara bawaan [11]. Semua itu tidak membuat disiplin bab ini tidak

diperlukan: kontainer yang diperkeras tidak memeriksa *raw pointer*, dan tidak ada standar yang dapat mengetahui kapan sebuah pointer hidup lebih lama daripada objeknya.

> **WAWASAN KUNCI.** Alat-alat itu menemukan bug yang *sudah terjadi* dalam suatu eksekusi. Mereka tidak dapat membuktikan program benar. Pertahanan sesungguhnya bersifat struktural: berikan setiap objek di *heap* tepat satu pemilik yang jelas, utamakan kontainer dan *smart pointer* daripada `new` dan `delete` mentah, dan biarkan destruktor yang membebaskan. Subbab 6.12 dan 6.13 membahas cara membangun struktur itu.

6.8 `nullptr`: Pointer ke Ketiadaan

Pointer yang tidak menunjuk objek apa pun tetap harus menyimpan *suatu* nilai, dan pilihan yang aman adalah pointer null. Kode C dan C++ lama menuliskannya sebagai `0` atau sebagai makro `NULL`, yang biasanya didefinisikan sebagai `0`. Ini menimbulkan masalah nyata: `0` pertama-tama adalah *bilangan bulat*, sehingga fungsi yang di-*overload* untuk `int` dan untuk `int*` memilih versi `int` bila dipanggil dengan `0` atau, pada sebagian kompilator, dengan `NULL`. C++11 memperkenalkan kata kunci `nullptr`, yang mempunyai tipe sendiri, `std::nullptr_t`, dan terkonversi ke setiap tipe pointer tetapi tidak ke tipe bilangan bulat mana pun.

Kode 6.10 memperlihatkan masalah *overload* itu terpecahkan, tiga uji null yang setara, dan pemakaian null yang terpenting dalam struktur data: fungsi pencarian yang mengembalikan pointer ke apa yang ditemukannya, atau `nullptr` bila tidak menemukan apa-apa.

Kode 6.10 `10_nullptr.cpp`: `nullptr` memilih *overload* pointer, dan pencarian mengembalikan `nullptr` untuk “tidak ditemukan”.

```cpp
// nullptr: pointer null yang punya tipe sendiri.
#include <iostream>
#include <string>

void describe(int value)  { std::cout << "  describe(int " << value << ")\n"; }
void describe(int* where) { std::cout << "  describe(int* " << (where ? "non-null" :
"null") << ")\n"; }

const std::string* findTitle(const std::string* titles, int n, char firstLetter) {
    for (int i = 0; i < n; ++i) {
        if (!titles[i].empty() && titles[i][0] == firstLetter) {
            return &titles[i];            // ketemu: alamat elemennya
        }
    }
    return nullptr;                       // tidak ketemu: pointer null
}

int main() {
    std::cout << "Calling describe with 0 and with nullptr:\n";
    describe(0);                          // 0 adalah int
    describe(nullptr);                    // nullptr hanya terkonversi ke pointer

    int* p{};                             // diinisialisasi nilai: null
    int* q = nullptr;
    std::cout << std::boolalpha << "p == nullptr: " << (p == nullptr)
              << ", p == q: " << (p == q) << ", !p: " << !p << '\n';

    std::string titles[3] = {"Laskar Pelangi", "Filosofi Kopi", "Bumi Manusia"};
    for (char letter : {'B', 'Z'}) {
        const std::string* hit = findTitle(titles, 3, letter);
        if (hit != nullptr) {             // periksa SEBELUM dereferensi
            std::cout << letter << ": found \"" << *hit << "\" at index "
                      << (hit - titles) << '\n';
        } else {
            std::cout << letter << ": no title starts with this letter\n";
        }
    }
    return 0;
}
```

```
Calling describe with 0 and with nullptr:
  describe(int 0)
  describe(int* null)
p == nullptr: true, p == q: true, !p: true
B: found "Bumi Manusia" at index 2
Z: no title starts with this letter
```

Aturan yang membuat null berguna sama pendeknya: *periksa sebelum mendereferensi*. `if (hit != nullptr)`, `if (hit)`, dan `if (!hit)` adalah tiga cara penulisan yang lazim. Mendereferensi pointer null adalah perilaku tak terdefinisi (pada sistem operasi biasa program mogok dengan *segmentation fault*, seperti yang diperlihatkan ASan), sehingga setiap pointer yang *bisa* null harus diperiksa di setiap jalur yang memakainya. Dari sini lahir sebuah konvensi yang praktis: parameter fungsi atau nilai kembalian bertipe pointer mengumumkan "ini bisa null", sedangkan referensi, pokok subbab berikutnya, mengumumkan "ini tidak pernah null".

Pointer yang belum diinisialisasi kadang secara longgar disebut "dangling", tetapi kosakata baku membedakan dua kasus. Pointer yang *belum diinisialisasi* (disebut juga *wild pointer*) tidak pernah diberi nilai; isinya bit apa pun yang tersisa di memorinya. Pointer yang *menggantung* (*dangling*) pernah menunjuk objek sungguhan yang sejak itu sudah mati. Keduanya berbahaya, dan obat untuk yang pertama sangat mudah: jangan pernah mendeklarasikan pointer tanpa menginisialisasinya, setidaknya dengan `nullptr`, seperti `int* p{};` atau `int* q = nullptr;` di **Kode 6.10**.

6.9 Referensi

6.9.1 Nama kedua untuk objek yang sudah ada

Pointer adalah variabel yang *menyimpan* alamat. *Referensi* adalah sesuatu yang lebih sederhana dan lebih terbatas: nama lain untuk objek yang sudah ada.

> **DEFINISI.** Deklarasi `T& r = x;` menjadikan `r` sebuah *referensi ke* `x`. Sejak itu `r` adalah alias `x`: setiap pemakaian `r`, baik membaca, menulis, maupun mengambil alamatnya, adalah pemakaian `x`. Referensi harus diinisialisasi ketika dideklarasikan, tidak pernah dapat dibuat merujuk objek lain, dan tidak pernah dapat bernilai null.

Kode 6.11 menjelajahi ketiga batasan itu dan dua tempat referensi paling sering dipakai, yaitu dalam loop dan sebagai nama bagi elemen.

Kode 6.11 `11_references.cpp`: referensi adalah alias; menugasinya berarti menugasi objeknya, dan loop `for` atas referensi mengubah kontainernya.

```
// Referensi: nama kedua untuk objek yang sudah ada.
#include <iostream>
#include <string>
#include <vector>

struct Book {
    std::string id, title;
    int year;
};

int main() {
    int a = 13;
    int& r = a;                          // r adalah nama lain untuk a
    r = 20;
    std::cout << std::boolalpha;
    std::cout << "after r = 20: a = " << a << ", &r == &a: " << (&r == &a) << '\n';

    int b = 99;
    r = b;                               // menugasi a dengan 99; r TIDAK dipindahkan
    b = 7;
    std::cout << "after r = b; b = 7: a = " << a << ", b = " << b
              << ", r = " << r << '\n';

    const int& sum = a + b;              // referensi const boleh terikat ke nilai
sementara
    std::cout << "const int& sum = a + b gives " << sum << '\n';

    std::vector<int> stock = {3, 5, 8};
    for (int x : stock) x *= 10;         // x salinan: stock tidak berubah
    std::cout << "after copy loop:      ";
    for (int x : stock) std::cout << x << ' ';
    for (int& x : stock) x *= 10;        // x referensi: stock berubah
    std::cout << "\nafter reference loop: ";
    for (int x : stock) std::cout << x << ' ';
    std::cout << '\n';

    Book catalog[2] = {{"B4001", "Laskar Pelangi", 2005},
                       {"B4002", "Filosofi Kopi", 2006}};
    Book& favorite = catalog[1];         // tanpa salinan, tanpa panah
    favorite.title += " (2nd ed.)";
    std::cout << "catalog[1].title = " << catalog[1].title << '\n';
    return 0;
}
```

```
after r = 20: a = 20, &r == &a: true
after r = b; b = 7: a = 99, b = 7, r = 99
const int& sum = a + b gives 106
after copy loop:      3 5 8
after reference loop: 30 50 80
catalog[1].title = Filosofi Kopi (2nd ed.)
```

Baris kedua keluaran adalah jebakan yang menangkap setiap pendatang baru. `r = b;` *tidak* membuat `r` merujuk `b`. Referensi tidak dapat dipindahkan, sehingga pernyataan itu berarti persis sama dengan `a = b;`: salin 99 ke `a`. Ketika `b` kemudian menjadi 7, `a` dan `r` tetap 99. Dengan pointer, `p = &b` akan memindahkan panahnya; dengan referensi tidak ada panah yang dapat dipindahkan, hanya label kedua yang direkatkan selamanya pada kotak yang sama.

Baris berikutnya memperlihatkan aturan khusus: referensi *const* boleh terikat pada nilai sementara seperti `a + b`, dan nilai sementara itu dihidupkan selama referensinya hidup. Aturan ini ada agar fungsi yang menerima `const T&` dapat dipanggil dengan ekspresi, seperti `printBook(makeBook())`.

Kedua loop memperlihatkan pemakaian sehari-hari yang paling umum. `for (int x : stock)` memberikan setiap elemen kepada salinan baru `x`, sehingga mengalikan `x` tidak mengubah apa pun di `stock`. `for (int& x : stock)` menjadikan `x` referensi ke setiap elemen secara bergiliran, sehingga perkaliannya terjadi di tempat. Akhirnya, `Book& favorite = catalog[1];` memberi nama pendek yang praktis bagi sebuah elemen tanpa menyalinnya dan tanpa sintaks panah milik pointer.

6.9.2 Referensi versus pointer

Di balik layar, kompilator biasanya mengimplementasikan referensi sebagai pointer yang didereferensi secara otomatis. Perbedaannya terletak pada apa yang diizinkan *bahasa*, dan **Tabel 6.6** menjajarkan keduanya.

Tabel 6.6 Perbandingan pointer dan referensi.

Sifat	Pointer `T* p`	Referensi `T& r`
Wajib diinisialisasi	tidak (tetapi sebaiknya)	ya
Dapat bernilai null	ya, `nullptr`	tidak
Dapat dipindahkan ke objek lain	ya, `p = &other;`	tidak; `r = other;` menugasi nilai
Sintaks untuk menjangkau objek	`*p`, `p->m`	`r`, `r.m`
Aritmetika	ya, di dalam larik	tidak
Mempunyai alamat sendiri	ya, `&p` bertipe `T**`	`&r` adalah alamat objeknya
Pemakaian khas	objek opsional, tautan yang dapat diarahkan ulang (simpul list), larik, kepemilikan	parameter, variabel loop, alias

Aturan praktis yang lahir dari **Tabel 6.6** juga merupakan anjuran C++ Core Guidelines, pedoman gaya yang dipelihara Stroustrup dan Sutter [12]: *pakailah referensi bila objeknya selalu harus ada dan tidak akan berganti; pakailah pointer bila "tidak ada objek" merupakan kemungkinan yang sah atau bila pointer itu kelak harus menunjuk ke tempat lain.* Struktur bertaut memerlukan pointer tepat karena kedua alasan itu: `next` milik simpul terakhir tidak mempunyai objek untuk dirujuk, dan penyisipan serta penghapusan harus mengarahkan ulang tautan.

6.10 Fungsi dan Tiga Cara Mengirim Argumen

6.10.1 Fungsi: tinjauan singkat

Fungsi adalah bagian program bernama yang mengerjakan satu tugas dan ditulis terpisah dari kode yang memakainya. Memecah program menjadi fungsi-fungsi adalah bentuk *divide and conquer* (bagi dan kuasai) yang paling tua: program besar menjadi program-program kecil yang dapat ditulis, diuji, diperbaiki, dan dipakai ulang satu per satu, oleh orang yang berbeda bila perlu. Fungsi yang tidak mengembalikan apa-apa dideklarasikan `void` (teks-teks lama menyebutnya *prosedur*); fungsi yang mengembalikan nilai menyebutkan tipe nilai itu di depan namanya dan diakhiri dengan `return`.

Beberapa gagasan lain termasuk dalam tinjauan ini, dan **Kode 6.12** menghimpunnya dalam satu program yang dapat dijalankan.

- **Deklarasi dan definisi.** Baris `int add(int x, int y);` di dekat awal adalah *deklarasi*, disebut juga *prototipe*: janji bahwa ada fungsi dengan nama ini, tipe parameter ini, dan tipe kembalian ini. *Definisinya*, lengkap dengan badannya, muncul sesudah `main`. Kompilator membaca berkas dari atas ke bawah, sehingga tanpa deklarasi itu pemanggilan `add(13, 3)` di dalam `main` akan menjadi galat. Dalam program berberkas banyak, deklarasi ditaruh di berkas *header* dan definisi di berkas `.cpp`.
- **Parameter formal dan aktual.** Dalam definisi, `x` dan `y` adalah parameter *formal*, variabel lokal milik `add`. Dalam pemanggilan `add(13, 3)`, nilai 13 dan 3 adalah parameter *aktual*, disebut juga *argumen*. Pemanggilan menginisialisasi setiap parameter formal dari argumen yang bersesuaian.
- **Cakupan dan umur.** Variabel *global*, yang dideklarasikan di luar semua fungsi, terlihat di seluruh kode sesudahnya dan hidup sepanjang eksekusi. Variabel *lokal* hanya terlihat di dalam bloknya dan hanya hidup selama blok itu berjalan. Variabel lokal yang namanya sama dengan variabel global *menyembunyikan* variabel global itu, dan `::a` tetap dapat menjangkau yang global. Variabel *lokal statis* hanya terlihat di dalam fungsinya tetapi, seperti variabel global, dibuat sekali dan mempertahankan nilainya antarpanggilan: `countCalls` menghitung dengan menjaga `calls` tetap hidup.
- **Nilai kembalian `main`.** `int main()` mengembalikan status kepada sistem operasi: 0 berarti sukses, selain itu gagal. Bentuk `void main()`, yang muncul dalam sebagian materi C lama, tidak sah dalam C++.

Kode 6.12 `12_functions_scope.cpp`: prototipe, parameter formal dan aktual, variabel global, lokal, dan statis, serta perbandingan pertama antara pengiriman berdasarkan nilai dan melalui pointer.

```cpp
// Fungsi: deklarasi sebelum dipakai, parameter, dan tiga jenis umur variabel
// (global, lokal, lokal statis).
#include <iostream>

int add(int x, int y);                    // deklarasi (prototipe)

int a = 3;                                // global: terlihat di seluruh kode di bawahnya

void aGlobal() {
    std::cout << "a in aGlobal() is " << a << " (the global)\n";
}

void passByValue(int a) {                 // a ini salinan lokal yang baru
    a = a * 2;
    std::cout << "a in passByValue() is " << a << '\n';
}

void passByPointer(int* a) {              // a ini menyimpan alamat milik main
    *a = *a * 2;
    std::cout << "*a in passByPointer() is " << *a << '\n';
}

int countCalls() {
    static int calls = 0;                 // dibuat sekali, bertahan antarpanggilan
    return ++calls;
}

int main() {
    int a = 13;                           // lokal: menyembunyikan a global
    aGlobal();
    std::cout << "a in main() is " << a << ", global ::a is " << ::a << '\n';
    passByValue(a);
    std::cout << "a in main() after passByValue() is " << a << '\n';
    passByPointer(&a);
    std::cout << "a in main() after passByPointer() is " << a << '\n';

    int t = add(13, 3);                   // parameter aktual 13 dan 3
    std::cout << "add(13, 3) = " << t << '\n';

    countCalls();
    countCalls();
    std::cout << "countCalls() has now been called " << countCalls() << " times\n";
    return 0;                             // 0 memberi tahu sistem operasi "sukses"
}
```

Kode 6.12 (Lanjutan) *12_functions_scope.cpp: prototipe, parameter formal dan aktual, variabel global, lokal, dan statis, serta perbandingan pertama antara pengiriman berdasarkan nilai dan melalui pointer.*

```cpp
int add(int x, int y) {                   // definisi: parameter formal x, y
    int r = x + y;
    return r;
}
```

```
a in aGlobal() is 3 (the global)
a in main() is 13, global ::a is 3
a in passByValue() is 26
a in main() after passByValue() is 13
*a in passByPointer() is 26
a in main() after passByPointer() is 26
add(13, 3) = 16
countCalls() has now been called 3 times
```

Baris-baris tengah keluaran sudah memuat sisa subbab ini dalam bentuk mini. `passByValue` menggandakan `a` *miliknya sendiri*, sebuah salinan, dan `a` milik `main` tetap 13 sesudahnya.

`passByPointer` menerima *alamat* milik `main` dan menggandakan apa yang tinggal di sana, sehingga `a` milik `main` menjadi 26.

6.10.2 Apa yang terjadi selama pemanggilan

Ketika `main` memanggil sebuah fungsi, mesin menumpuk *frame* baru di *stack* untuk fungsi yang dipanggil. *Frame* itu mempunyai slot untuk setiap parameter dan setiap variabel lokal fungsi tersebut. Sebelum badan fungsi dijalankan, setiap slot parameter *diinisialisasi dari* argumen yang bersesuaian. Segala sesuatu dalam subbab ini mengalir dari satu pertanyaan: *apa* yang dipakai untuk menginisialisasi parameter itu?

- Jika parameternya bertipe biasa `T`, ia diinisialisasi dengan **salinan nilai argumen**. Inilah *pass by value* (pengiriman berdasarkan nilai).
- Jika parameternya pointer `T*`, ia diinisialisasi dengan salinan sebuah **alamat**, yang diberikan pemanggil dengan `&`. Inilah *pass by pointer* (tepatnya, mengirim pointer berdasarkan nilai).
- Jika parameternya referensi `T&`, tidak ada objek baru yang dibuat sama sekali: parameter itu menjadi **nama lain bagi argumen itu sendiri**. Inilah *pass by reference* (pengiriman melalui referensi).

6.10.3 Kisah *swap*

Uji klasik untuk ketiga cara ini adalah fungsi yang menukar dua bilangan bulat. Kedengarannya remeh, padahal inilah eksperimen terbaik dalam seluruh topik ini. **Kode 6.13** menuliskannya dengan tiga cara dan memanggil ketiganya berturut-turut pada dua variabel yang sama.

Kode 6.13 `13_swap.cpp`: menukar berdasarkan nilai (gagal), melalui pointer dan melalui referensi (keduanya berhasil), serta `std::swap`.

```cpp
// Kisah swap: berdasarkan nilai, melalui pointer, melalui referensi.
#include <iostream>
#include <utility>

void swapByValue(int x, int y) {          // x, y adalah salinan
    int t = x;
    x = y;
    y = t;
    std::cout << "  inside swapByValue:     x = " << x << ", y = " << y << '\n';
}

void swapByPointer(int* x, int* y) {      // x, y menyimpan alamat milik pemanggil
    if (x == nullptr || y == nullptr) return;
    int t = *x;
    *x = *y;
    *y = t;
}

void swapByReference(int& x, int& y) {    // x, y ADALAH variabel pemanggil
    int t = x;
    x = y;
    y = t;
}

int main() {
    int a = 13, b = 3;
    std::cout << "start:                  a = " << a << ", b = " << b << '\n';

    swapByValue(a, b);
    std::cout << "after swapByValue:      a = " << a << ", b = " << b << '\n';

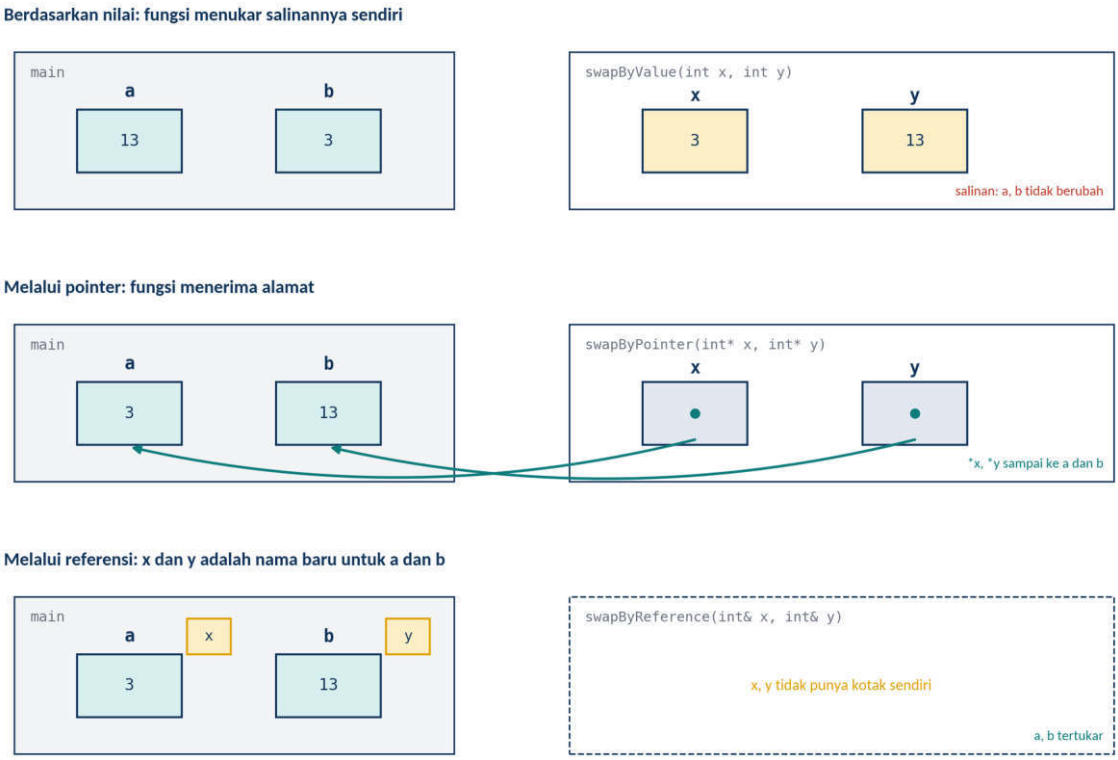
    swapByPointer(&a, &b);
    std::cout << "after swapByPointer:    a = " << a << ", b = " << b << '\n';

    swapByReference(a, b);
    std::cout << "after swapByReference:  a = " << a << ", b = " << b << '\n';

    std::swap(a, b);                      // swap pustaka standar berbasis referensi
    std::cout << "after std::swap:        a = " << a << ", b = " << b << '\n';
    return 0;
}
```

```
start:                  a = 13, b = 3
  inside swapByValue:     x = 3, y = 13
after swapByValue:      a = 13, b = 3
after swapByPointer:    a = 3, b = 13
after swapByReference:  a = 13, b = 3
after std::swap:        a = 3, b = 13
```

Ikuti variabel-variabelnya melalui **Gambar 6.8**.



Gambar 6.8 Satu *swap* yang ditulis dengan tiga cara. Berdasarkan nilai, fungsi menukar salinannya sendiri; melalui pointer, fungsi menukar variabel pemanggil lewat alamatnya; melalui referensi, parameternya adalah variabel pemanggil dengan nama baru.

Berdasarkan nilai. `swapByValue(a, b)` membuat `x` $=13$ dan `y` $=3$ di *frame*-nya sendiri. Badannya menukar keduanya, dan baris yang dicetak dari dalam membuktikannya: `x = 3, y = 13`. Lalu fungsi selesai, *frame*-nya diambil dari *stack*, dan salinan yang sudah ditukar itu lenyap. `a` dan `b` milik `main` tidak pernah ikut serta. Ini bukan bug dalam C++; inilah persis makna *pass by value*, dan itulah sebabnya cara ini aman: *fungsi yang dipanggil berdasarkan nilai tidak dapat merusak variabel pemanggilnya.*

Melalui pointer. `swapByPointer(&a, &b)` mengirim alamat `a` dan `b`. Parameter `x` dan `y` adalah pointer, dan badan fungsi bekerja melaluinya: `t = *x` membaca `a` milik `main`, `*x = *y` menulis `a` milik `main` dari `b` milik `main`, dan `*y = t` menulis `b` milik `main`. Variabel pemanggil tertukar: `a = 3, b = 13`. Ada dua hal yang terlihat dalam kode sumber dan dapat diperiksa pembaca: titik pemanggilan menulis `&a`, yang mengingatkan pembaca bahwa `a` mungkin berubah, dan fungsinya memeriksa kedua pointer terhadap `nullptr` sebelum memakainya, karena parameter pointer bisa null.

Melalui referensi. `swapByReference(a, b)` di titik pemanggilan tampak persis seperti versi berdasarkan nilai, tetapi `x` di dalam badan fungsi *adalah* `a` dan `y` *adalah* `b`. Tidak ada salinan, tidak ada alamat untuk diikuti, tidak ada null untuk diperiksa. Variabel-variabel itu, yang oleh pemanggilan pointer baru saja ditukar, kini ditukar sekali lagi. `std::swap` dari pustaka standar ditulis dengan cara yang sama.

> **CONTOH.** Telusuri ketiga pemanggilan mulai dari `a = 13, b = 3`. Sesudah `swapByValue`: $a = 13, b = 3$, tidak berubah. Sesudah `swapByPointer`: $a = 3, b = 13$. Sesudah `swapByReference`: nilainya ditukar lagi, sehingga $a = 13, b = 3$. Sesudah `std::swap`: $a = 3, b = 13$. Setiap baris keluaran program cocok dengan penelusuran ini.

6.10.4 Biaya pengiriman berdasarkan nilai

Swap menyangkut *apakah pemanggil berubah*. Ada alasan kedua untuk menghindari *pass by value*, dan alasan itu tetap berlaku walaupun fungsinya hanya membaca: sebuah salinan memakan waktu dan memori sebanding dengan ukuran objek yang disalin. Jika sebuah objek menempati s byte dan fungsi yang menerimanya berdasarkan nilai dipanggil k kali, banyaknya byte yang disalin program adalah

$$\text{byte yang disalin} = \begin{cases} k \cdot s & \text{berdasarkan nilai,} \\ 0 & \text{melalui referensi atau pointer} \end{cases} \tag{6.6}$$

dan untuk objek yang memiliki memori *heap*, seperti `std::string` dan `std::vector`, setiap salinan juga mengalokasikan lalu kelak membebaskan memorinya sendiri. Untuk `int` ($s = 4$) salinannya lebih murah daripada mengikuti referensi, sehingga tipe bawaan yang kecil biasanya dikirim berdasarkan nilai. Untuk `Book` yang berisi tiga string, dan terutama untuk kontainer berisi ribuan elemen, Persamaan (6.6) yang memutuskan: kirimlah melalui referensi `const`. Subbab 6.11 mengukurnya.

6.10.5 Memilih cara

Tabel 6.7 merangkum pilihannya. Tabel itu menyatakan ulang anjuran Core Guidelines tentang pengiriman parameter [12] dalam bentuk pertanyaan yang sungguh-sungguh diajukan pemrogram.

Tabel 6.7 Memilih cara fungsi C++ menerima argumennya.

Fungsi perlu…	Tulis parameternya sebagai	Pemanggilannya	Contoh
membaca nilai kecil (`int`, `double`, `char`, pointer)	`T x`	`f(a)`	`add(int x, int y)`
membaca objek besar	`const T& x`	`f(book)`	`print(const Book& b)`
mengubah objek pemanggil yang selalu ada	`T& x`	`f(book)`	`swapByReference(int& x, int& y)`
mengubah objek yang mungkin tidak ada, atau menerima larik	`T* x` (periksa `nullptr`)	`f(&book)`, `f(array)`	`swapByPointer`, `fill(int* data, int n)`
tetap menyimpan salinannya sendiri	`T x`	`f(book)`	`makeReprint` (Subbab 6.12)
mengambil alih objek di *heap*	`std::unique_ptr<T> x`	`f(std::move(p))`	`withdraw` (Subbab 6.13)

Mengembalikan *beberapa* hasil adalah pertanyaan yang berkaitan. Pendekatan klasik, dan satu-satunya pendekatan C, adalah *parameter keluaran*: kirimkan pointer atau referensi dan biarkan fungsi menulis ke dalamnya, seperti yang dilakukan `minMaxByPointer` dan `minMaxByReference` pada penyelesaian Soal 6.12. C++ modern biasanya lebih menyukai pengembalian struktur kecil atau `std::pair` yang dibongkar dengan *structured binding*, `auto [lo, hi] = minMax(v);`, karena nilai kembalian lebih mudah dibaca di titik pemanggilan daripada argumen yang diam-diam ditulisi.

6.11 Ketepatan `const`

Objek *const* adalah objek yang dijanjikan program untuk tidak diubah, dan kompilator menagih janji itu. Pada pointer ada *dua* hal yang dapat bersifat konstan, pointernya dan objek yang ditunjuknya, sehingga ada empat kombinasi. **Tabel 6.8** mendaftarnya. Cara yang andal untuk membacanya adalah *dari kanan ke kiri*, mulai dari nama variabel:

Tabel 6.8 Empat kombinasi `const` dengan pointer.

Deklarasi	Dibaca dari kanan ke kiri	Mengubah *p?	Mengubah p?
`int* p`	`p` adalah pointer ke int	ya	ya
`const int* p`	`p` adalah pointer ke int yang const	tidak	ya
`int* const p`	`p` adalah pointer const ke int	ya	tidak
`const int* const p`	`p` adalah pointer const ke int const	tidak	tidak

(`int const* p` adalah ejaan lain yang sah untuk `const int* p`; `const` berlaku pada apa yang ada di kirinya, atau di kanannya bila di kirinya tidak ada apa-apa.) **Kode 6.14** mencoba setiap kombinasi dan juga memperagakan *fungsi anggota const*: `int age(int now) const` berjanji tidak mengubah `Book` tempat ia dipanggil, dan justru itulah yang memungkinkannya dipanggil pada `const Book novel`.

Kode 6.14 `14_const_correct.cpp`: empat bentuk `const`, fungsi anggota const, dan biaya penyalinan yang diukur dengan penghitung salinan.

```cpp
// Ketepatan const, dan biaya sebenarnya dari parameter berdasarkan nilai.
#include <iostream>
#include <string>
#include <utility>

struct Book {
    std::string id, title;
    int year;
    static long copies;                   // berapa Book yang sudah disalin
    Book(std::string i, std::string t, int y) : id(std::move(i)), title(std::move(t)),
year(y) {}
    Book(const Book& other) : id(other.id), title(other.title), year(other.year) {
++copies; }
    int age(int now) const { return now - year; }   // const: tidak mengubah *this
};
long Book::copies = 0;

int ageByValue(Book b, int now) { return b.age(now); }          // menyalin Book
int ageByConstRef(const Book& b, int now) { return b.age(now); } // tanpa salinan, hanya
baca

int main() {
    int stock = 5, spare = 9;

    const int* readOnly = &stock;         // pointer ke const int
    readOnly = &spare;                    // boleh: pointernya sendiri boleh berpindah
    std::cout << "*readOnly = " << *readOnly << '\n';

    int* const fixedPlace = &stock;       // pointer const ke int
    *fixedPlace = 6;                      // boleh: int-nya boleh berubah
    std::cout << "stock = " << stock << '\n';

    const int* const frozen = &stock;     // keduanya tidak boleh berubah
    std::cout << "*frozen = " << *frozen << '\n';

    const Book novel("B4003", "Bumi Manusia", 1980);
    std::cout << "age in 2026 = " << novel.age(2026) << '\n';   // diizinkan: age bersifat
const

    long total = 0;
    for (int i = 0; i < 1000; ++i) total += ageByValue(novel, 2026);
    std::cout << "1000 calls by value:     " << Book::copies << " copies\n";
    Book::copies = 0;
    for (int i = 0; i < 1000; ++i) total += ageByConstRef(novel, 2026);
    std::cout << "1000 calls by const ref: " << Book::copies << " copies\n";
    std::cout << "(sum of ages " << total << ")\n";
    return 0;
}
```

```
*readOnly = 9
stock = 6
*frozen = 6
age in 2026 = 46
1000 calls by value:     1000 copies
1000 calls by const ref: 0 copies
(sum of ages 92000)
```

Semua yang ada dalam program itu lolos kompilasi karena menepati janjinya. Langgarlah janji itu dan kompilator menghentikan Anda, dan memang itulah gunanya menulis `const`:

```
  *readOnly = 1;          // error: assignment of read-only location '* readOnly'
  fixedPlace = &spare;    // error: assignment of read-only variable 'fixedPlace'
  novel.reprint();        // error: passing 'const Book' as 'this' argument discards
  qualifiers
```

(Itulah pesan persis GCC 13 untuk ketiga pernyataan itu, yang dikompilasi dalam berkas uji kecil.) Penghitung salinan mengukur Persamaan (6.6): seribu pemanggilan `ageByValue` membuat seribu salinan `Book`, masing-masing menyalin dua string, sedangkan seribu pemanggilan `ageByConstRef` tidak membuat satu pun, dan keduanya menghitung total yang sama. Itulah sebabnya parameter besar yang hanya dibaca ditulis `const T&` di hampir seluruh kode C++, dan itulah sebabnya Bab 2 sampai 5 menulis `const Book&` di mana pun sebuah buku hanya dibaca.

Ketepatan const (*const correctness*) adalah kebiasaan menandai `const` segala sesuatu yang tidak perlu berubah: parameter, fungsi anggota, variabel lokal yang dihitung sekali. Kebiasaan ini tidak berbiaya apa pun saat program berjalan, mendokumentasikan maksud, dan mengubah sekeluarga bug akibat perubahan tak disengaja menjadi galat kompilasi.

6.12 Mengembalikan Pointer dan Referensi dengan Aman

Fungsi dapat mengembalikan pointer atau referensi, dan di sinilah *dangling pointer* paling sering lahir. Aturannya pendek: *jangan pernah mengembalikan alamat, atau referensi, ke objek yang mati ketika fungsi selesai.* Contoh buku teksnya adalah variabel lokal:

```
int* makeCounter() {
    int counter = 0;
    return &counter;
}
```

```
local_addr.cpp:3:12: warning: address of local variable 'counter' returned [-Wreturn-
local-addr]
```

GCC menangkap kasus sederhana ini dengan peringatan, dan bahkan melangkah lebih jauh: karena tahu alamat itu tidak berguna, GCC 13 membuat fungsi mengembalikan pointer null (kami memeriksanya pada `-O0` dan pada `-O2`), sehingga pemanggil langsung mogok. Versi yang kurang kentara, seperti mengembalikan pointer ke elemen `std::vector` lokal atau referensi ke anggota objek sementara, tidak selalu tertangkap.

Lalu apa yang *aman*? **Kode 6.15** memperlihatkan empat pola yang aman.

Kode 6.15 `15_find_book.cpp`: mengembalikan pointer ke dalam larik milik pemanggil atau `nullptr`, referensi ke dalam vector milik pemanggil, dan nilai baru.

```cpp
// Mengembalikan pointer dan referensi dengan aman.
#include <iostream>
#include <string>
#include <vector>

struct Book {
    std::string id, title;
    int year;
};

// Mengembalikan alamat elemen yang cocok, atau nullptr. Array milik
// pemanggil, sehingga alamatnya tetap sah setelah fungsi selesai.
const Book* findById(const Book* first, const Book* last, const std::string& id) {
    for (const Book* p = first; p != last; ++p) {
        if (p->id == id) return p;
    }
    return nullptr;
}

// Kembaran non-const agar pemanggil dapat mengubah yang ditemukan.
Book* findById(Book* first, Book* last, const std::string& id) {
    for (Book* p = first; p != last; ++p) {
        if (p->id == id) return p;
    }
    return nullptr;
}

// Mengembalikan referensi ke buku tertua: vector hidup lebih lama daripada
// panggilan.
const Book& oldest(const std::vector<Book>& books) {
    std::size_t best = 0;
    for (std::size_t i = 1; i < books.size(); ++i) {
        if (books[i].year < books[best].year) best = i;
    }
    return books[best];
}

// Mengembalikan nilai baru: hasilnya objek baru milik pemanggil.
Book makeReprint(const Book& original, int year) {
    Book copy = original;
    copy.title += " (reprint)";
    copy.year = year;
    return copy;
}
```

Kode 6.15 (Lanjutan) *15_find_book.cpp: mengembalikan pointer ke dalam larik milik pemanggil atau nullptr, referensi ke dalam vector milik pemanggil, dan nilai baru.*

```
int main() {
    Book shelf[3] = {{"B4001", "Laskar Pelangi", 2005},
                     {"B4003", "Bumi Manusia", 1980},
                     {"B4005", "Cantik Itu Luka", 2002}};

    const Book* view = shelf;            // pandangan hanya-baca atas array yang sama
    for (const char* id : {"B4003", "B4999"}) {
        const Book* hit = findById(view, view + 3, id);
        if (hit) std::cout << id << " -> " << hit->title << '\n';
        else     std::cout << id << " -> not in the catalog\n";
    }

    if (Book* b = findById(shelf, shelf + 3, "B4005")) {
        b->year = 2004;                  // ubah melalui pointer yang dikembalikan
    }
    std::cout << "shelf[2].year is now " << shelf[2].year << '\n';

    std::vector<Book> books(shelf, shelf + 3);
    const Book& first = oldest(books);
    std::cout << "oldest: " << first.title << " (" << first.year << ")\n";

    Book fresh = makeReprint(first, 2025);
    std::cout << "reprint: " << fresh.title << " (" << fresh.year << ")\n";
    return 0;
}
```

```
B4003 -> Bumi Manusia
B4999 -> not in the catalog
shelf[2].year is now 2004
oldest: Bumi Manusia (1980)
reprint: Bumi Manusia (reprint) (2025)
```

1. **Pointer ke dalam sesuatu milik pemanggil.** `findById` mencari di larik milik pemanggil dan mengembalikan alamat elemen yang cocok, atau `nullptr`. Larik itu hidup lebih lama daripada pemanggilannya, sehingga pointer tetap sah selama pemanggil tidak mengubah lariknya. Versi `const` melayani pembaca; kembarannya yang non-`const` memungkinkan pemanggil mengubah elemen yang ditemukan, seperti yang dilakukan `b->year = 2004`. Inilah pola setiap pencarian dalam struktur bertaut, yang hasilnya berupa pointer ke sebuah simpul.
2. **Referensi ke dalam sesuatu milik pemanggil.** `oldest` mengembalikan `const Book&` ke elemen vector milik pemanggil: tanpa salinan, dan tidak pernah null, cocok untuk fungsi yang selalu mempunyai jawaban (untuk vector yang tidak kosong).
3. **Nilai baru.** `makeReprint` mengembalikan `Book` berdasarkan nilai. Dulu pengembalian berdasarkan nilai ditakuti karena mahal; C++ modern dalam kebanyakan kasus membangun hasilnya langsung di variabel pemanggil (*return value optimization*), atau memindahkannya, sehingga mengembalikan objek besar sekalipun berdasarkan nilai adalah pilihan bawaan yang bersih untuk "membuat sesuatu yang baru".
4. **Objek baru di *heap* beserta pemiliknya.** Bila objek baru harus tinggal di *heap*, misalnya karena akan ditautkan ke dalam sebuah struktur, fungsi mengembalikan `std::unique_ptr`, pokok subbab berikutnya, sehingga kepemilikannya tertulis dalam tipe.

Frasa "selama pemanggil tidak mengubah lariknya" pada pola 1 bukan sekadar formalitas. Seandainya larik itu sebuah `std::vector` dan pemanggil menambahkan elemen ke dalamnya, pointer itu akan menggantung persis seperti bug `stale` di Subbab 6.7.

6.13 *Smart Pointer*: Kepemilikan yang Tertulis dalam Tipe

6.13.1 Masalah dengan `new` dan `delete` mentah

Setiap bug di Subbab 6.7 berasal dari satu pertanyaan yang tidak dapat dijawab *raw pointer*: *siapa yang bertanggung jawab menghapus objek ini?* Sebuah `Book*` mentah bisa menunjuk ke dalam larik, ke variabel lokal, ke objek *heap* yang akan dihapus pihak lain, atau ke objek *heap* yang *Anda* wajib hapus. Tidak ada apa pun dalam tipenya yang menyatakan yang mana. Lebih buruk lagi, `delete` yang benar pun dapat terlewati: bila sebuah eksepsi dilempar atau fungsi kembali lebih awal di antara `new` dan `delete`, objek itu bocor.

C++ menjawabnya dengan teknik bernama RAII, *Resource Acquisition Is Initialization* (perolehan sumber daya adalah inisialisasi): ikatkan setiap sumber daya pada sebuah objek yang destruktornya melepaskan sumber daya itu. Karena C++ menjamin destruktor dijalankan ketika objek keluar dari cakupan, entah lewat jalan keluar yang normal, `return` yang lebih awal, maupun eksepsi, pelepasan itu tidak mungkin terlupa. Kita sudah memakai RAII pada `IntArray` di **Kode 6.8**. Pustaka standar mengemasnya untuk objek *heap* sebagai *smart pointer* (pointer cerdas), yang didefinisikan di *header* `<memory>` [1], [2].

6.13.2 `std::unique_ptr`: tepat satu pemilik

`std::unique_ptr<T>` menyimpan pointer ke objek *heap* dan *memiliki* objek itu: ketika `unique_ptr` dihancurkan, ia menghapus objeknya. Ia tidak dapat disalin, karena dua pemilik akan menghapus dua kali; ia hanya dapat *dipindahkan* (*move*), yang mengalihkan kepemilikan dan membuat sumbernya kosong. Ia sama sekali tidak lebih mahal daripada *raw pointer*, baik dalam ukuran maupun kecepatan. **Kode 6.16** memberi `Book` destruktor yang mengumumkan dirinya, sehingga kita dapat menyaksikan setiap objek mati tepat pada saatnya.

Kode 6.16 `16_unique_ptr.cpp`: membuat, memindahkan, menyerahkan, dan mengumpulkan pemilik `std::unique_ptr`; tidak ada `delete` di mana pun.

```cpp
// std::unique_ptr: tepat satu pemilik, dan delete terjadi dengan sendirinya.
#include <iostream>
#include <memory>
#include <string>
#include <utility>
#include <vector>

struct Book {
    std::string id, title;
    Book(std::string i, std::string t) : id(std::move(i)), title(std::move(t)) {
        std::cout << "  [create " << id << "]\n";
    }
    ~Book() { std::cout << "  [destroy " << id << "]\n"; }
};

std::unique_ptr<Book> acquire(const std::string& id, const std::string& title) {
    return std::make_unique<Book>(id, title);    // kepemilikan berpindah ke pemanggil
}

void withdraw(std::unique_ptr<Book> b) {         // mengambil kepemilikan (sebuah "sink")
    std::cout << "  withdrawing " << b->title << '\n';
}                                                // b dihancurkan di sini

int main() {
    std::cout << "1. make_unique\n";
    std::unique_ptr<Book> a = acquire("B4001", "Laskar Pelangi");
    std::cout << "  a->title = " << a->title << '\n';

    std::cout << "2. move ownership from a to b\n";
    std::unique_ptr<Book> b = std::move(a);      // salinan tidak akan lolos kompilasi
    std::cout << std::boolalpha << "  a is empty: " << (a == nullptr)
              << ", b->id = " << b->id << '\n';

    std::cout << "3. hand b to a function that keeps it\n";
    withdraw(std::move(b));
    std::cout << "  back in main, b is empty: " << (b == nullptr) << '\n';
```

Kode 6.16 (Lanjutan) *16_unique_ptr.cpp: membuat, memindahkan, menyerahkan, dan mengumpulkan pemilik std::unique_ptr; tidak ada delete di mana pun.*

```cpp
    std::cout << "4. a vector of owners\n";
    {
        std::vector<std::unique_ptr<Book>> shelf;
        shelf.push_back(acquire("B4002", "Filosofi Kopi"));
        shelf.push_back(acquire("B4003", "Bumi Manusia"));
        Book* observer = shelf[1].get();         // bukan pemilik, hanya melihat
        std::cout << "  observer sees " << observer->title << '\n';
        std::cout << "  leaving the block...\n";
    }                                            // vector menghancurkan keduanya

    std::cout << "5. an owned array\n";
    std::unique_ptr<int[]> counts = std::make_unique<int[]>(4);   // diisi nol
    counts[2] = 7;
    std::cout << "  counts = " << counts[0] << ' ' << counts[1] << ' '
              << counts[2] << ' ' << counts[3] << '\n';
    std::cout << "end of main\n";
    return 0;
}
```

```
1. make_unique
   [create B4001]
   a->title = Laskar Pelangi
2. move ownership from a to b
   a is empty: true, b->id = B4001
3. hand b to a function that keeps it
   withdrawing Laskar Pelangi
   [destroy B4001]
   back in main, b is empty: true
4. a vector of owners
   [create B4002]
   [create B4003]
   observer sees Bumi Manusia
   leaving the block...
   [destroy B4002]
   [destroy B4003]
5. an owned array
   counts = 0 0 7 0
end of main
```

Telusuri keluarannya terhadap kodenya.

1. `std::make_unique<Book>(id, title)` mengalokasikan sebuah `Book` di *heap* dan mengembalikan pointer pemiliknya. Fungsi pabrik `acquire` mengembalikannya, dan kepemilikan berpindah ke `a` di `main`.
2. `std::unique_ptr<Book> b = std::move(a);` mengalihkan kepemilikan. `std::move` sendiri tidak memindahkan apa pun; ia menandai `a` sebagai sesuatu yang boleh diambil isinya. Sesudahnya `a` kosong (sama dengan `nullptr`) dan `b` memiliki *B4001*. Menulis `b = a;` tanpa `std::move` tidak lolos kompilasi, dan pesan GCC menjelaskan alasannya: `use of deleted function 'std::unique_ptr<...>::unique_ptr(const std::unique_ptr<...>&)'`. Operasi salin itu sengaja dihapus.
3. `withdraw(std::move(b))` menyerahkan kepemilikan kepada fungsi yang menerima `unique_ptr` berdasarkan nilai. Ketika `withdraw` selesai, parameternya dihancurkan, dan bersamanya `Book` itu: `[destroy B4001]` tercetak *di dalam* langkah 3, sebelum `main` berlanjut.
4. Vector berisi `unique_ptr` memiliki satu rak utuh. `shelf[1].get()` mengembalikan *raw pointer* yang *bukan pemilik* bagi pihak yang hanya perlu melihat. Ketika bloknya berakhir, vector dihancurkan, ia menghancurkan `unique_ptr`-nya satu per satu, dan `unique_ptr` itu menghancurkan buku-bukunya.
5. `std::make_unique<int[]>(4)` memiliki sebuah larik; destruktornya memakai `delete[]`, sehingga bug ketidakcocokan di Subbab 6.7 tidak mungkin terjadi.

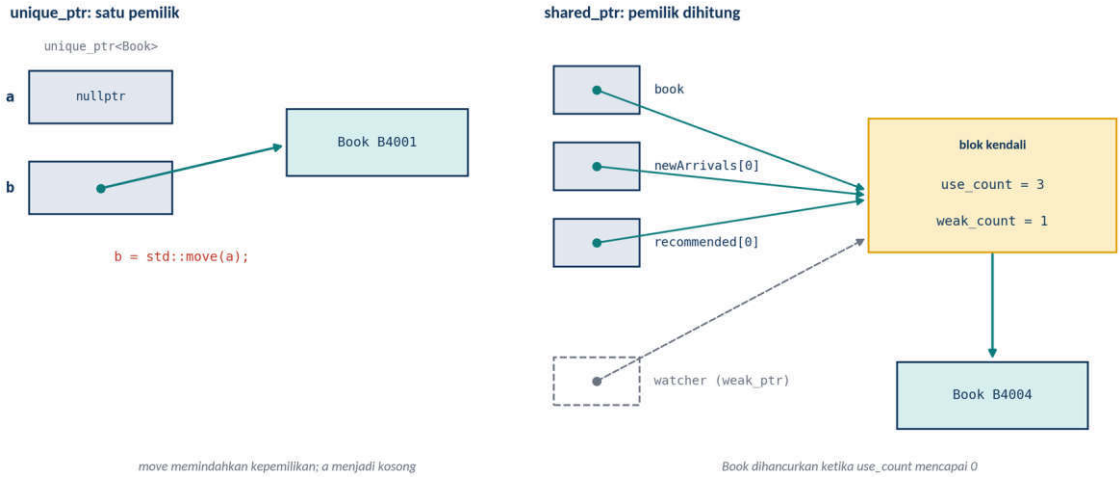
Dijalankan di bawah Valgrind atau AddressSanitizer, **Kode 6.16** bersih: setiap `new` yang tersembunyi di dalam `make_unique` dipasangkan dengan `delete` yang tersembunyi di dalam destruktor, dan pemrogram tidak menulis keduanya.

6.13.3 `std::shared_ptr` dan `std::weak_ptr`: pemilik yang dihitung

Kadang sebuah objek sungguh-sungguh mempunyai beberapa pemilik, dan tidak satu pun tahu siapa yang akan selesai paling akhir. Di Pustaka Ceria, satu buku dapat berada di daftar *New Arrivals* sekaligus di daftar *Recommended*; buku itu semestinya lenyap ketika sudah keluar dari kedua daftar. `std::shared_ptr<T>` menanganinya dengan *reference count* (hitungan referensi). Di samping objeknya, dalam sebuah *blok kendali* kecil, ia menyimpan banyaknya `shared_ptr` yang saat ini memilikinya. Menyalin `shared_ptr` menaikkan hitungan; menghancurkan atau mengosongkan satu `shared_ptr` menurunkannya; dan objek dihancurkan tepat ketika

$$\text{use\_count} = 0.$$

`std::weak_ptr` mengamati objek bersama *tanpa* memilikinya: ia tidak menahan hitungan, dan sebelum memakai objeknya Anda memanggil `lock()`, yang mengembalikan `shared_ptr` bila objek itu masih ada atau `shared_ptr` kosong bila sudah hilang. **Gambar 6.9** mempertentangkan kedua jenis kepemilikan itu.



Gambar 6.9 Kiri: `unique_ptr` mempunyai satu pemilik, dan *move* mengalihkannya. Kanan: tiga `shared_ptr` berbagi satu `Book` melalui blok kendali yang menghitungnya; `weak_ptr` mengamati tanpa ikut dihitung.

Kode 6.17 `17_shared_ptr.cpp`: buku yang dibagi dua daftar dan dihancurkan ketika pemilik terakhir melepaskannya, serta sepasang anggota yang ditautkan dengan `weak_ptr` sehingga keduanya dapat dihancurkan.

```
// std::shared_ptr dan std::weak_ptr: kepemilikan bersama yang dihitung.
#include <iostream>
#include <memory>
#include <string>
#include <utility>
#include <vector>

struct Book {
    std::string id;
    explicit Book(std::string i) : id(std::move(i)) { std::cout << "  [create " << id <<
"]\n"; }
    ~Book() { std::cout << "  [destroy " << id << "]\n"; }
};

struct Member {                          // dua anggota yang saling merekomendasikan
    std::string name;
    std::shared_ptr<Member> friendStrong; // tautan pemilik: dapat membentuk siklus
    std::weak_ptr<Member> friendWeak;     // tautan pengamat: tidak dapat
    explicit Member(std::string n) : name(std::move(n)) {}
    ~Member() { std::cout << "  [destroy member " << name << "]\n"; }
};

int main(int argc, char* argv[]) {
    std::cout << "1. one book on two lists\n";
    std::vector<std::shared_ptr<Book>> newArrivals, recommended;
    {
        auto book = std::make_shared<Book>("B4004");
        std::cout << "  use_count = " << book.use_count() << '\n';
        newArrivals.push_back(book);
        recommended.push_back(book);
        std::cout << "  use_count = " << book.use_count() << " (book + 2 lists)\n";
    }
    std::cout << "  local gone, use_count = " << newArrivals[0].use_count() << '\n';
    std::weak_ptr<Book> watcher = newArrivals[0];
    newArrivals.clear();
    std::cout << "  newArrivals cleared, use_count = " << recommended[0].use_count() <<
'\n';
    recommended.clear();                 // pemilik terakhir melepaskan
    std::cout << "  watcher expired: " << std::boolalpha << watcher.expired() << '\n';
```

Kode 6.17 (Lanjutan) *17_shared_ptr.cpp: buku yang dibagi dua daftar dan dihancurkan ketika pemilik terakhir melepaskannya, serta sepasang anggota yang ditautkan dengan weak_ptr sehingga keduanya dapat dihancurkan.*

```
    bool cycle = argc > 1 && std::string(argv[1]) == "cycle";
    std::cout << "2. two members who point at each other ("
              << (cycle ? "shared_ptr both ways" : "weak_ptr links") << ")\n";
    {
        auto ani = std::make_shared<Member>("Ani");
        auto budi = std::make_shared<Member>("Budi");
        if (cycle) {
            ani->friendStrong = budi;
            budi->friendStrong = ani;     // siklus pemilik
        } else {
            ani->friendWeak = budi;
            budi->friendWeak = ani;
        }
        if (auto f = ani->friendWeak.lock()) {
            std::cout << "  Ani's friend is " << f->name << '\n';
        }
        std::cout << "  leaving the block...\n";
    }
    std::cout << "end of main\n";
    return 0;
}
```

```
1. one book on two lists
  [create B4004]
  use_count = 1
  use_count = 3 (book + 2 lists)
  local gone, use_count = 2
  newArrivals cleared, use_count = 1
  [destroy B4004]
  watcher expired: true
2. two members who point at each other (weak_ptr links)
  Ani's friend is Budi
  leaving the block...
  [destroy member Budi]
  [destroy member Ani]
end of main
```

Kode 6.17 menjalankan kedua situasi. Bagian pertama mengikuti hitungannya: 1 sesudah `make_shared`, 3 begitu kedua daftar memegang salinan, 2 ketika `book` lokal keluar dari cakupan, 1 sesudah `newArrivals` dikosongkan, dan 0, disertai `[destroy B4004]`, ketika `recommended` melepaskannya. `watcher` lalu melaporkan bahwa objeknya sudah kedaluwarsa.

Bagian kedua memperlihatkan satu hal yang tidak dapat dilakukan penghitungan referensi sendirian. Ani dan Budi saling merekomendasikan. Seandainya masing-masing memegang yang lain dengan `shared_ptr`, setiap hitungan akan tetap 1 selamanya setelah nama-nama lokalnya lenyap, karena setiap objek dihidupkan oleh objek lainnya: sebuah *siklus referensi*, dan sebuah kebocoran. Jalankan program dengan kata `cycle` dan kedua baris `[destroy member ...]` tidak muncul; pemeriksa kebocoran AddressSanitizer lalu melaporkan dua blok kendali yang bocor, masing-masing 80 byte. Menautkan keduanya dengan `weak_ptr`, seperti pada eksekusi bawaan di atas, memutus siklus itu dan keduanya dihancurkan. Aturan umumnya: pemilik menunjuk *ke bawah* dalam struktur dengan `unique_ptr` atau `shared_ptr`, dan tautan balik menunjuk *ke atas* dengan `weak_ptr` atau *raw pointer* yang bukan pemilik. *Doubly linked list* di Bab 9 dan pointer ke induk pada pohon di Bab 11 adalah tautan balik semacam itu.

Tabel 6.9 merangkum pilihannya. Dalam C++ masa kini *raw pointer* tetap sangat layak, bahkan diutamakan, sebagai *pengamat yang bukan pemilik*, seperti `observer` di **Kode 6.16** atau hasil `findById` di **Kode 6.15**; yang dihindari kode modern adalah *raw pointer* yang *memiliki*.

Tabel 6.9 *Raw pointer*, referensi, dan *smart pointer*, serta kapan memakai masing-masing.

Alat	Memiliki?	Dapat disalin?	Biaya	Pakailah untuk
`T*`	tidak (menurut konvensi)	ya	nol	mengamati, argumen opsional, tautan ke objek milik pihak lain
`T&`	tidak	terikat sekali	nol	argumen dan alias yang selalu ada
`std::unique_ptr<T>`	ya, sendirian	hanya *move*	nol	pemilik bawaan objek *heap*
`std::shared_ptr<T>`	ya, bersama	ya (dihitung)	blok kendali, pembaruan hitungan atomik	objek dengan beberapa pemilik yang umurnya tak diketahui
`std::weak_ptr<T>`	tidak	ya	memakai blok kendali	tautan balik, *cache*, memutus siklus

C++ Core Guidelines memadatkan hal ini menjadi aturan R.20 sampai R.24 [12]: pakailah `unique_ptr` atau `shared_ptr` untuk menyatakan kepemilikan, utamakan `unique_ptr` daripada `shared_ptr`, buatlah dengan `make_unique` dan `make_shared`, dan pakailah `weak_ptr` untuk memutus siklus.

> **WAWASAN KUNCI.** Lalu mengapa Bab 7 sampai 13 masih menulis `new` dan `delete` mentah dalam *linked list* dan pohonnya? Karena studi pertama tentang struktur data adalah tempat Anda belajar apa yang dikerjakan *smart pointer* atas nama Anda. Menulis sendiri `delete`-nya, lalu memeriksanya dengan alat-alat di Subbab 6.7, adalah cara gagasan kepemilikan menjadi konkret. Dalam kode produksi, simpul yang memiliki sisa sebuah *list* akan memegang penerusnya dalam `unique_ptr`.

6.14 Referensi Java dan Nama Python

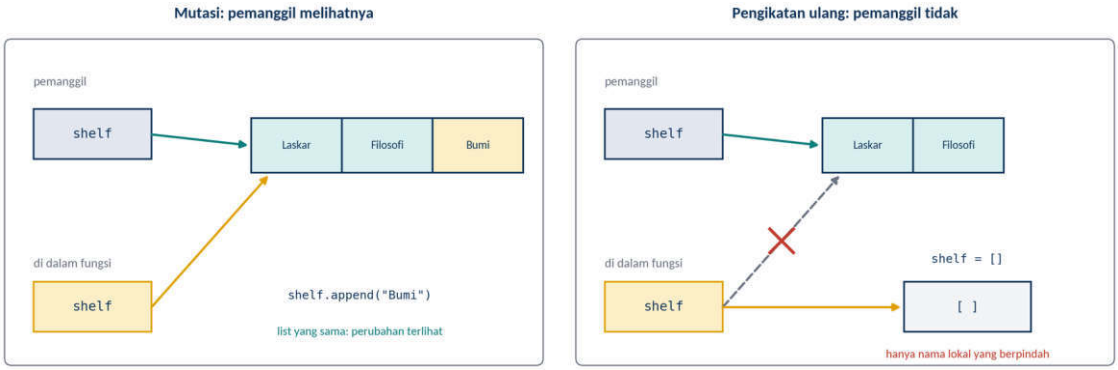
6.14.1 Satu aturan untuk masing-masing

Java dan Python tidak mempunyai aritmetika pointer, tidak ada `&`, tidak ada `*`, dan tidak ada `delete`. Namun setiap program Java atau Python yang tidak remeh penuh dengan gagasan bab ini, tersembunyi di balik satu aturan untuk tiap bahasa.

Java. Variabel bertipe *primitif* (`int`, `double`, `char`, `boolean`, dan empat lainnya) menyimpan nilainya secara langsung, seperti `int` dalam C++. Variabel bertipe *kelas atau larik* apa pun menyimpan *referensi*: dalam istilah C++, pointer yang tidak dapat digeser dengan aritmetika dan diikuti secara otomatis oleh tanda titik. Objek hanya dibuat dengan `new` dan selalu di *heap*; tidak ada yang menghapusnya; *garbage collector* (pengumpul sampah) mengambil kembali objek begitu tidak ada lagi referensi kepadanya. Dan setiap argumen, tanpa kecuali, **dikirim berdasarkan nilai** [13]: metode menerima salinan argumen. Untuk tipe primitif, salinan itu adalah bilangannya; untuk objek, salinannya adalah *referensi*, sehingga metode dan pemanggil merujuk objek yang sama.

Python. Setiap nilai adalah objek, dan setiap variabel adalah *nama yang terikat pada sebuah objek*. Penugasan `x = y` tidak pernah menyalin objek; ia mengikat nama `x` pada objek yang terikat pada `y`. Pemanggilan fungsi mengikat setiap nama parameter pada objek yang dikirim, sehingga fungsi dan pemanggil berbagi objek itu [14]. Aturan ini mempunyai beberapa nama dalam literatur. Dokumentasi Python menyatakan argumen dikirim *by assignment* (melalui penugasan); para pemrogram sering menyebutnya *pass by object reference*; dan istilah *call by sharing*, yang diciptakan untuk bahasa CLU oleh Liskov dan rekan-rekannya pada tahun 1970-an [15], melukiskan perilaku yang sama di Java maupun Python.

Dalam bahasa C++: variabel objek Java dan nama Python berperilaku seperti *pointer C++ yang dikirim berdasarkan nilai*. Fungsi yang dipanggil dapat mengikuti salinan pointernya dan mengubah objek di ujung sana, dan pemanggil akan melihat perubahan itu. Fungsi itu juga dapat membuat salinan pointernya menunjuk ke tempat lain, dan pemanggil *tidak* akan menyadarinya. Yang pertama adalah **mutasi**; yang kedua adalah **pengikatan ulang** (*rebinding*). **Gambar 6.10** menggambar keduanya, dan perbedaan di antara keduanya adalah gagasan terpenting dalam subbab ini.



Gambar 6.10 Kiri: fungsi memutasi *list* yang dirujuk parameternya, dan pemanggil melihat perubahannya. Kanan: fungsi mengikat ulang parameternya pada *list* baru, dan nama milik pemanggil masih merujuk *list* yang lama.

6.14.2 Mutasi versus pengikatan ulang, dalam Java

Kode 6.18 menjalankan eksperimen itu dalam Java pada sebuah rak judul buku.

Kode 6.18 `RefsDemo.java`: aliasing, mutasi melalui parameter, pengikatan ulang parameter, argumen primitif, salinan sungguhan, `==` versus `equals`, dan referensi null.

```
import java.util.ArrayList;
import java.util.List;

// Mutasi versus pengikatan ulang: apa yang bisa dan tidak bisa dilakukan
// metode terhadap pemanggilnya.
public class RefsDemo {
    static List<String> nothing = null;

    static void addBook(List<String> shelf) {
        shelf.add("Bumi Manusia");                    // memutasi list milik pemanggil
    }

    static void replaceShelf(List<String> shelf) {
        shelf = new ArrayList<>();                    // hanya mengikat ulang salinan lokal
        shelf.add("Negeri 5 Menara");
    }

    static void bumpYear(int year) {
        year++;                                       // salinan int
    }

    public static void main(String[] args) {
        List<String> shelf = new ArrayList<>(List.of("Laskar Pelangi"));
        List<String> alias = shelf;                   // dua variabel, satu list
        alias.add("Filosofi Kopi");
        System.out.println("shelf after alias.add:   " + shelf);
        System.out.println("shelf == alias: " + (shelf == alias));

        addBook(shelf);
        System.out.println("shelf after addBook:     " + shelf);
        replaceShelf(shelf);
        System.out.println("shelf after replaceShelf: " + shelf);

        int year = 2005;
        bumpYear(year);
        System.out.println("year after bumpYear:     " + year);

        List<String> copy = new ArrayList<>(shelf);  // salinan sungguhan
        copy.clear();
        System.out.println("copy cleared, shelf has " + shelf.size() + " books");

        String a = new String("Pustaka"), b = new String("Pustaka");
        System.out.println("a == b: " + (a == b) + ", a.equals(b): " + a.equals(b));
```

Kode 6.18 (Lanjutan) *RefsDemo.java: aliasing, mutasi melalui parameter, pengikatan ulang parameter, argumen primitif, salinan sungguhan, == versus equals, dan referensi null.*

```
        try {
            nothing.add("x");                         // nothing adalah field bernilai null
        } catch (NullPointerException e) {
            System.out.println("NullPointerException: " + e.getMessage());
        }
    }
}
```

```
shelf after alias.add:   [Laskar Pelangi, Filosofi Kopi]
shelf == alias: true
shelf after addBook:     [Laskar Pelangi, Filosofi Kopi, Bumi Manusia]
shelf after replaceShelf: [Laskar Pelangi, Filosofi Kopi, Bumi Manusia]
year after bumpYear:     2005
copy cleared, shelf has 3 books
a == b: false, a.equals(b): true
NullPointerException: Cannot invoke "java.util.List.add(Object)" because "RefsDemo.nothing"
is null
```

Baris demi baris:

- `alias = shelf` menyalin referensi. Ada satu *list* dengan dua nama, sehingga `alias.add` mengubah apa yang ditampilkan `shelf`, dan `shelf == alias` bernilai `true`. Untuk objek, `==` dalam Java membandingkan *referensi*, persis seperti membandingkan dua pointer C++.
- `addBook(shelf)` menerima salinan referensi dan mengikutinya: mutasi, terlihat oleh pemanggil.
- `replaceShelf(shelf)` menerima salinan referensi lalu *menugasi ulang salinan itu* dengan *list* yang benar-benar baru. `shelf` milik pemanggil masih merujuk *list* lama, yang tidak berubah. Ini persis `swapByValue` dari **Kode 6.13**, dengan referensi menggantikan `int`.
- `bumpYear(year)` menerima salinan `int` primitif; tidak ada yang dilakukan metode itu yang dapat menjangkau `year` milik `main`.
- `new ArrayList<>(shelf)` membuat salinan sungguhan, *list* baru dengan elemen yang sama; mengosongkan salinan itu tidak menyentuh `shelf`.
- Dua objek `String` dengan karakter yang sama adalah dua objek yang berbeda: `a == b` bernilai `false`, sedangkan `a.equals(b)`, yang membandingkan isi, bernilai `true`. Memakai `==` pada string adalah salah satu kesalahan Java yang paling umum, dan itu tidak lain perbedaan pointer-versus-nilai di Subbab 6.3.2 yang sedang menyamar.
- Mengikuti referensi `null` melempar `NullPointerException`. Sejak JDK 15 pesannya menyebutkan ekspresi persis yang bernilai null [16], seperti diperlihatkan baris terakhir; padanannya dalam C++ adalah *segmentation fault* di Subbab 6.7.

6.14.3 Mutasi versus pengikatan ulang, dalam Python

Kode 6.19 adalah eksperimen yang sama dalam Python, ditambah satu jebakan yang hanya ada di sana.

Kode 6.19 `refs_demo.py`: eksperimen yang sama dengan nama-nama Python, ditambah jebakan argumen bawaan yang dapat diubah.

```
# Mutasi versus pengikatan ulang: apa yang bisa dan tidak bisa dilakukan
# fungsi terhadap pemanggilnya.
def add_book(shelf):
    shelf.append("Bumi Manusia")       # memutasi list milik pemanggil


def replace_shelf(shelf):
    shelf = []                         # hanya mengikat ulang nama lokal
    shelf.append("Negeri 5 Menara")


def bump_year(year):
    year += 1                          # int tak dapat diubah: int baru, nama lokal


def add_tag(tag, tags=[]):             # list bawaan dibuat SEKALI
    tags.append(tag)
    return tags


shelf = ["Laskar Pelangi"]
alias = shelf                          # dua nama, satu list
alias.append("Filosofi Kopi")
print(f"shelf after alias.append:  {shelf}")
print(f"shelf is alias: {shelf is alias}")

add_book(shelf)
print(f"shelf after add_book:      {shelf}")
replace_shelf(shelf)
print(f"shelf after replace_shelf: {shelf}")

year = 2005
bump_year(year)
print(f"year after bump_year:      {year}")

copy = list(shelf)                     # salinan sungguhan (dangkal)
copy.clear()
print(f"copy cleared, shelf has {len(shelf)} books")

a, b = "".join(["Pus", "taka"]), "".join(["Pust", "aka"])
print(f"a is b: {a is b}, a == b: {a == b}")

print(f"add_tag('novel')  -> {add_tag('novel')}")
print(f"add_tag('sejarah') -> {add_tag('sejarah')}   (the same default list!)")
```

Kode 6.19 (Lanjutan) *refs_demo.py: eksperimen yang sama dengan nama-nama Python, ditambah jebakan argumen bawaan yang dapat diubah.*

```
nothing = None
try:
    nothing.append("x")
except AttributeError as e:
    print(f"AttributeError: {e}")
```

```
shelf after alias.append:  ['Laskar Pelangi', 'Filosofi Kopi']
shelf is alias: True
shelf after add_book:      ['Laskar Pelangi', 'Filosofi Kopi', 'Bumi Manusia']
shelf after replace_shelf: ['Laskar Pelangi', 'Filosofi Kopi', 'Bumi Manusia']
year after bump_year:      2005
copy cleared, shelf has 3 books
a is b: False, a == b: True
add_tag('novel')  -> ['novel']
add_tag('sejarah') -> ['novel', 'sejarah']   (the same default list!)
AttributeError: 'NoneType' object has no attribute 'append'
```

Hasilnya cocok dengan program Java baris demi baris, dengan kosakata Python sendiri: `is` menanyakan apakah dua nama terikat pada objek yang sama (setara `==` pada referensi Java), sedangkan `==` membandingkan nilai (setara `equals` Java). Mengikat ulang parameter di `replace_shelf` tidak berpengaruh pada pemanggil, demikian pula `year += 1` di `bump_year`. Alasan yang kedua layak diuraikan: `int` Python *tak dapat diubah* (*immutable*), sehingga `year += 1` tidak dapat mengubah objek 2005; ia membuat objek 2006 lalu mengikat ulang nama *lokal* padanya. String dan tuple juga tak dapat diubah. List, dictionary, set, dan instans kelas biasa dapat diubah (*mutable*), dan objek-objek itulah yang dapat diubah sebuah fungsi atas nama pemanggilnya.

Dua baris tentang `add_tag` memperlihatkan jebakannya. Nilai bawaan seperti `tags=[]` dievaluasi *sekali*, ketika pernyataan `def` dijalankan, dan *list* hasilnya disimpan bersama fungsi itu. Setiap pemanggilan yang mengandalkan nilai bawaan mendapat *list* yang *sama*, sehingga *list* itu mengingat pemanggilan-pemanggilan sebelumnya: pemanggilan kedua mengembalikan `['novel', 'sejarah']`. Perbaikan bakunya adalah `def add_tag(tag, tags=None):` diikuti `if tags is None: tags = []`, yang membuat *list* baru pada setiap pemanggilan.

6.14.4 Kisah *swap* dalam Java dan Python

Kedua bahasa itu tidak dapat menulis fungsi yang menukar dua variabel milik pemanggil, dan kini kita dapat menyebut alasannya dengan tepat: fungsi menerima salinan nilai atau salinan referensi, dan mengikat ulang salinan tidak pernah sampai ke pemanggil. **Kode 6.20** dan **Kode 6.21** memperlihatkan apa yang *bisa* dilakukan.

Kode 6.20 `SwapDemo.java`: Java dapat menukar isi dua objek, atau dua slot sebuah larik, tetapi tidak pernah dua variabel milik pemanggil.

```java
// Kisah swap di Java: argumen selalu dikirim berdasarkan nilai, dan untuk
// objek nilai itu adalah referensi.
public class SwapDemo {
    static void swapInts(int x, int y) {             // salinan int
        int t = x; x = y; y = t;
    }

    static void swapBoxes(int[] x, int[] y) {        // salinan referensi
        int[] t = x; x = y; y = t;                   // hanya mengikat ulang salinan
    }

    static void swapContents(int[] x, int[] y) {     // memutasi objek bersama
        int t = x[0]; x[0] = y[0]; y[0] = t;
    }

    static void swapInArray(int[] a, int i, int j) { // idiom Java yang lazim
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }

    public static void main(String[] args) {
        int a = 13, b = 3;
        swapInts(a, b);
        System.out.println("after swapInts:     a = " + a + ", b = " + b);

        int[] boxA = {13}, boxB = {3};
        swapBoxes(boxA, boxB);
        System.out.println("after swapBoxes:    boxA[0] = " + boxA[0] + ", boxB[0] = " +
boxB[0]);
        swapContents(boxA, boxB);
        System.out.println("after swapContents: boxA[0] = " + boxA[0] + ", boxB[0] = " +
boxB[0]);

        int[] pair = {13, 3};
        swapInArray(pair, 0, 1);
        System.out.println("after swapInArray:  pair = " + pair[0] + ", " + pair[1]);
    }
}
```

```
after swapInts:     a = 13, b = 3
after swapBoxes:    boxA[0] = 13, boxB[0] = 3
after swapContents: boxA[0] = 3, boxB[0] = 13
after swapInArray:  pair = 3, 13
```

Kode 6.21 `swap_demo.py`: percobaan yang sama dalam Python, diakhiri dengan penugasan tuple yang menukar di pihak pemanggil.

```
# Kisah swap di Python: fungsi menerima referensi ke objek dan hanya dapat
# mengikat ulang nama lokalnya sendiri.
def swap_ints(x, y):
    x, y = y, x                         # hanya mengikat ulang nama-nama lokal


def swap_boxes(x, y):
    x, y = y, x                         # tetap hanya nama lokal


def swap_contents(x, y):
    x[0], y[0] = y[0], x[0]             # memutasi kedua list bersama


a, b = 13, 3
swap_ints(a, b)
print(f"after swap_ints:     a = {a}, b = {b}")

box_a, box_b = [13], [3]
swap_boxes(box_a, box_b)
print(f"after swap_boxes:    box_a[0] = {box_a[0]}, box_b[0] = {box_b[0]}")
swap_contents(box_a, box_b)
print(f"after swap_contents: box_a[0] = {box_a[0]}, box_b[0] = {box_b[0]}")

a, b = b, a                             # idiom Python: ikat ulang di pemanggil
print(f"after a, b = b, a:   a = {a}, b = {b}")
```

```
after swap_ints:     a = 13, b = 3
after swap_boxes:    box_a[0] = 13, box_b[0] = 3
after swap_contents: box_a[0] = 3, box_b[0] = 13
after a, b = b, a:   a = 3, b = 13
```

`swapInts` dan `swap_ints` gagal karena alasan yang sama dengan `swapByValue`. `swapBoxes` dan `swap_boxes` juga gagal, dan dengan cara yang lebih mendidik: keduanya menukar *referensi*, tetapi hanya salinan milik fungsi itu sendiri. `swapContents` dan `swap_contents` berhasil, karena keduanya memutasi dua objek bersama alih-alih mengikat ulang nama; inilah padanan `swapByPointer` di Java dan Python. Dalam kode Java sungguhan bentuk yang lazim adalah `swapInArray(a, i, j)`, dan dengan cara itulah setiap rutin pengurutan di Bab 3 menukar elemen. Dalam Python, *swap* yang idiomatis sama sekali bukan fungsi: `a, b = b, a` membangun tuple dari kedua objek lalu membongkarnya ke dalam kedua nama, di dalam cakupan pemanggil sendiri.

6.14.5 Siapa yang membebaskan memori?

C++ membebaskan objek *heap* ketika program memintanya, secara langsung dengan `delete` atau melalui destruktor *smart pointer*. Java dan Python membebaskannya secara otomatis, tetapi dengan cara yang berbeda, dan perbedaan itu tersambung rapi dengan Subbab 6.13.

CPython, implementasi Python yang baku, menyimpan hitungan referensi di setiap objek, persis seperti blok kendali `shared_ptr`, dan membebaskan objek pada saat hitungannya turun ke nol. `sys.getrefcount` menampilkan hitungan itu. Karena penghitungan saja tidak dapat membebaskan siklus, CPython menambahkan *cycle collector* (pengumpul siklus), yaitu modul `gc` [17], yang secara berkala mencari kelompok objek yang hanya terjangkau dari satu sama lain. Program `shared_demo.py` di folder kode bab ini memperlihatkan keduanya: buku yang dihancurkan begitu *list* terakhir melepaskannya, dan dua anggota dalam siklus yang bertahan sampai `gc.collect()` menemukannya. Java sama sekali tidak memakai hitungan. *Garbage collector*-nya yang bersifat *tracing* (menelusur) secara berkala mulai dari variabel yang pasti masih hidup (variabel lokal metode yang sedang berjalan, field statis) dan menandai segala yang

terjangkau dari sana; yang tidak bertanda adalah sampah, siklus pun termasuk. Harganya, saat pengambilan kembali tidak dapat diramalkan, dan karena itulah Java dan Python sama-sama menyediakan konstruksi eksplisit, `try`-with-resources dan `with`, untuk sumber daya seperti berkas yang harus dilepaskan pada saat yang diketahui: gagasan RAII lagi, dengan ejaan berbeda.

6.14.6 Tiga bahasa berdampingan

Tabel 6.10 menghimpun perbandingannya.

Tabel 6.10 Pointer, referensi, dan pengiriman parameter dalam C++, Java, dan Python.

Pertanyaan	C++	Java	Python
Apa isi sebuah variabel?	objeknya sendiri, atau pointer, atau ia sebuah referensi	nilai primitif, atau referensi ke objek	ikatan sebuah nama pada objek
Dapatkah alamat dilihat atau dihitung?	ya: `&`, `*`, aritmetika pointer	tidak	tidak (`id` memberi nomor identitas)
Bagaimana argumen dikirim?	berdasarkan nilai, melalui pointer, atau melalui referensi: pilihan Anda per parameter	selalu berdasarkan nilai; untuk objek nilainya adalah referensi	parameter diikat pada objek argumen (“call by sharing”)
Dapatkah fungsi menukar dua variabel pemanggil?	ya, melalui pointer atau referensi	tidak	tidak (pakai `a, b = b, a`)
Dapatkah fungsi mengubah objek pemanggil?	ya, melalui pointer atau referensi non-const	ya, bila objeknya dapat diubah	ya, bila objeknya dapat diubah
Objek yang sama? / Nilai yang sama?	`p == q` / `*p == *q`	`a == b` / `a.equals(b)`	`a is b` / `a == b`
“Tidak ada objek”	`nullptr`	`null`	`None`
Memakai “tidak ada objek”	perilaku tak terdefinisi (biasanya mogok)	`NullPointerException`	`AttributeError` atau `TypeError`
Membebaskan objek *heap*	`delete`, atau destruktor *smart pointer*	*garbage collector* yang menelusur	hitungan referensi ditambah pengumpul siklus
Mungkinkah *dangling pointer* atau *double free*?	ya	tidak	tidak
Mungkinkah kebocoran?	ya	ya, dengan menyimpan referensi yang tak diperlukan	ya, dengan menyimpan referensi yang tak diperlukan

Dua baris terakhir perlu dikomentari. *Garbage collection* menghapus *dangling pointer* dan *double free* sepenuhnya, dan itulah sebagian besar alasan Java dan Python disebut bahasa yang aman-memori. Namun ia tidak menghapus kebocoran. Objek yang masih terjangkau, karena sebuah *list* atau *cache* yang berumur panjang masih merujuknya, tidak pernah dikumpulkan walaupun program tidak akan memakainya lagi. Versi Java dan Python dari `mem_bugs` di folder kode memperlihatkannya: kasus `leak` menambahkan larik ke sebuah *list* statis, dan *garbage collector* dengan tepat menolak membebaskannya.

6.15 Biaya Setiap Operasi

Buku struktur data berutang kepada pembacanya biaya setiap operasi. **Tabel 6.11** memberikannya untuk perkakas bab ini, dengan n banyaknya elemen yang terlibat dan s ukuran objek dalam byte.

Tabel 6.11 Waktu eksekusi operasi-operasi dalam bab ini.

Operasi	Biaya	Alasan
`&x`, `*p`, `p->m`, `p + i`, `q - p`, `p == q`	$\Theta(1)$	beberapa instruksi mesin (Persamaan (6.1), Persamaan (6.2))
`a[i]` untuk larik atau melalui pointer	$\Theta(1)$	satu perhitungan alamat dan satu akses (Persamaan (6.3))
mengikuti rantai k pointer `next`	$\Theta(k)$	satu dereferensi per tautan; tidak ada jalan pintas
mengirim atau mengembalikan melalui referensi atau pointer	$\Theta(1)$	satu alamat
mengirim berdasarkan nilai	$\Theta(s)$	salinan setiap byte, ditambah memori *heap* yang dimiliki objek (Persamaan (6.6))
`new T`, `delete p`	$O(1)$ secara khas, tidak dijamin	bergantung pada alokator; jauh lebih lambat daripada variabel *stack*
`new T[n]`, `delete[] p`	$\Theta(n)$ untuk tipe yang mempunyai konstruktor atau destruktor	setiap elemen dikonstruksi atau dihancurkan
mendorong ke larik yang menggandakan diri	$O(1)$ teramortisasi, $\Theta(n)$ untuk satu dorongan terburuk	Persamaan (6.5)
operasi `unique_ptr`	sama dengan *raw pointer*	tidak ada data tambahan
menyalin atau menghancurkan `shared_ptr`	$O(1)$, lebih lambat daripada *raw pointer*	memperbarui penghitung atomik

Dua baris **Tabel 6.11** akan membentuk segala yang menyusul. Menjangkau elemen ke-i sebuah larik berbiaya $\Theta(1)$; menjangkau simpul ke-i sebuah rantai berbiaya $\Theta(i)$. Dan menyisipkan di tengah larik berarti menggeser segala yang ada sesudahnya, $\Theta(n)$, sedangkan menyisipkan ke dalam rantai berarti mengubah dua pointer, $\Theta(1)$ begitu Anda sudah sampai di sana. Bab 7 adalah pengembangan penuh dari tawar-menawar itu.

6.16 Profitina dalam Praktik

Profitina, platform visibilitas AI yang sudah diperkenalkan di bab-bab sebelumnya, mengukur seberapa siap sebuah situs web dikutip oleh mesin penjawab AI dan membantu pemiliknya memperbaiki temuan-temuannya [18]. Aplikasi webnya ditulis dalam Python dengan kerangka kerja FastAPI, dan model bahasanya (Qwen2.5, dilayani secara lokal melalui Ollama) berjalan di server yang sama. Dua gagasan dari bab ini muncul dalam kerja hariannya dalam bentuk yang sangat konkret.

Mengalirkan hasil melalui *pipeline* tanpa menyalinnya. Pemindaian sebuah situs adalah rangkaian pengukuran: ambil halaman, ekstrak teksnya, periksa data terstrukturnya, periksa sinyal teknisnya, hitung skornya, lalu jelaskan setiap sinyal. Dalam Python setiap tahap adalah fungsi yang menerima objek hasil pemindaian, dan menurut aturan Subbab 6.14 setiap tahap menerima *objek yang sama*, bukan salinannya. Itulah tepatnya yang dibutuhkan sebuah *pipeline*. Teks hasil ekstraksi sebuah halaman bisa mencapai ribuan kata; seandainya setiap tahap menyalinnya, seperti yang dilakukan *pass by value* dalam C++, salinan itu akan memakan waktu

dan memori sebanding dengan ukuran halaman di setiap langkah (Persamaan (6.6)). Berbagi itu gratis. Aturan yang sama membawa kewajiban yang selalu menyertai berbagi: tahap yang seharusnya hanya membaca tidak boleh memutasi, karena setiap tahap sesudahnya akan melihat perubahan itu. Gerbang rilis Profitina, rangkaian uji otomatis berisi jauh lebih dari seribu uji yang harus lulus sebelum setiap rilis, mencakup uji jenis ini, yang memberi sebuah tahap suatu hasil lalu memeriksa bahwa apa yang semestinya tak tersentuh memang tak tersentuh. Jebakan argumen bawaan yang dapat diubah di **Kode 6.19** adalah sumber klasik bug semacam itu dalam layanan web Python, karena *list* bawaan yang dibuat sekali dipakai bersama oleh setiap permintaan yang pernah dilayani server.

Satu objek mahal, banyak pemakai. Model bahasa lokal menempati memori berukuran gigabyte, dan tidak masuk akal bila setiap permintaan memuat salinannya sendiri. Model dimuat sekali dan setiap permintaan yang memerlukannya bekerja dengan referensi ke satu instans bersama itu, pola `shared_ptr` dan hitungan referensi Python di Subbab 6.13 dan 6.14. Ketika tidak ada model yang tersedia, aplikasi mengatakannya terus terang alih-alih diam-diam menyajikan templat: hasil “null” ditangani secara eksplisit, tidak pernah didereferensi begitu saja, yaitu disiplin `nullptr` Subbab 6.8 dalam skala seluruh layanan.

Studi kasus yang dapat dijalankan untuk bab ini adalah **Profitina Laundry**, usaha binatu kecil yang dipakai sebagai contoh praktik di semua buku Profitina. Pesanannya menunggu dalam dua antrean, *regular* dan *express*, dan pelanggan dapat membayar untuk memindahkan pesanan dari satu antrean ke antrean lain. **Kode 6.22** mengimplementasikannya dalam C++ dengan setiap pesanan dimiliki tepat satu antrean melalui `unique_ptr`. Memindahkan pesanan berarti memindahkan *pointer*-nya, bukan pesanannya, sehingga pengamat yang bukan pemilik tetap berfungsi. Program itu juga mempertentangkan penawaran harga yang dihitung pada salinan dengan diskon yang diterapkan melalui referensi.

Kode 6.22 `18_laundry.cpp`: antrean Profitina Laundry memiliki pesanannya melalui `unique_ptr`; pemindahan antrean memindahkan kepemilikan, bukan pesanannya.

```cpp
// Profitina Laundry: pesanan dimiliki antrean, dipindahkan antarantrean tanpa
// disalin, diamati lewat raw pointer, dan diperbarui melalui referensi.
#include <deque>
#include <iostream>
#include <memory>
#include <string>
#include <utility>

struct Order {
    int number;
    std::string customer;
    double kilograms;
    long price;                             // dalam rupiah
    Order(int n, std::string c, double kg, long p)
        : number(n), customer(std::move(c)), kilograms(kg), price(p) {}
    ~Order() { std::cout << "  [order " << number << " released]\n"; }
};

using Queue = std::deque<std::unique_ptr<Order>>;

void applyDiscount(Order& o, int percent) {        // melalui referensi: mengubah pesanan
    o.price -= o.price * percent / 100;
}

long quoteWithDiscount(Order o, int percent) {     // berdasarkan nilai: bekerja pada
salinan
    o.price -= o.price * percent / 100;
    return o.price;
}

// Pindahkan pesanan bernomor ini dari satu antrean ke antrean lain. Hanya
// pointernya yang berpindah; objek Order tetap di tempatnya di heap.
bool promote(Queue& from, Queue& to, int number) {
    for (auto it = from.begin(); it != from.end(); ++it) {
        if ((*it)->number == number) {
            to.push_back(std::move(*it));  // kepemilikan berpindah tangan
            from.erase(it);                // hapus slot yang kini kosong
            return true;
        }
    }
    return false;
}
```

Kode 6.22 (Lanjutan) *18_laundry.cpp: antrean Profitina Laundry memiliki pesanannya melalui unique_ptr; pemindahan antrean memindahkan kepemilikan, bukan pesanannya.*

```
void print(const char* name, const Queue& q) {
    std::cout << name << ":";
    for (const auto& o : q) {
        std::cout << "  #" << o->number << ' ' << o->customer << " Rp" << o->price;
    }
    std::cout << '\n';
}

int main() {
    Queue regular, express;
    regular.push_back(std::make_unique<Order>(101, "Sari", 3.0, 21000));
    regular.push_back(std::make_unique<Order>(102, "Joko", 5.5, 38500));
    regular.push_back(std::make_unique<Order>(103, "Rina", 2.0, 14000));
    print("regular", regular);

    const Order* watched = regular[1].get();       // amati pesanan 102
    promote(regular, express, 102);
    print("regular", regular);
    print("express", express);
    std::cout << "watched still sees #" << watched->number
              << " (same object, new owner)\n";

    std::cout << "quote with 10% off: Rp" << quoteWithDiscount(*express[0], 10)
              << ", stored price still Rp" << express[0]->price << '\n';
    applyDiscount(*express[0], 10);
    std::cout << "after applyDiscount: Rp" << express[0]->price << '\n';
    std::cout << "(quoteWithDiscount copied the order, and the copy was released)\n";

    express.pop_front();                           // pesanan 102 diambil
    std::cout << "end of main\n";
    return 0;
}
```

```
regular:  #101 Sari Rp21000  #102 Joko Rp38500  #103 Rina Rp14000
regular:  #101 Sari Rp21000  #103 Rina Rp14000
express:  #102 Joko Rp38500
watched still sees #102 (same object, new owner)
quote with 10% off: Rp34650, stored price still Rp38500
  [order 102 released]
after applyDiscount: Rp34650
(quoteWithDiscount copied the order, and the copy was released)
  [order 102 released]
end of main
  [order 101 released]
  [order 103 released]
```

Bacalah kepemilikannya pada keluaran. Pesanan 102 dibuat di `regular`, dipindahkan ke `express` dengan `std::move`, dan tidak pernah disalin; *raw pointer* `watched`, yang diambil sebelum pemindahan, sesudahnya masih melihat `#102` karena objek `Order` itu sendiri tidak pernah berpindah di memori, hanya `unique_ptr` pemiliknya yang berganti antrean. `quoteWithDiscount` menerima pesanan *berdasarkan nilai*, menghitung $38500 - 3850 = 34650$ pada salinannya, dan membiarkan harga tersimpan tetap 38500; baris `[order 102 released]` yang pertama adalah salinan itu yang dihancurkan di akhir pernyataan. `applyDiscount` menerima pesanan *melalui referensi* dan mengubah harga yang sebenarnya. Akhirnya `pop_front` mengambil pesanan 102, dan pesanan-pesanan yang tersisa dilepaskan secara otomatis ketika antrean-antreannya dihancurkan di akhir `main`. Tidak ada `delete` di mana pun, dan Valgrind melaporkan semua blok *heap* telah dibebaskan.

Kode 6.23 adalah program yang sama dalam Python. Pemindahan cukup berupa pengambilan sebuah referensi dari satu `deque` dan penambahannya ke `deque` lain; `watched is express[0]` mencetak `True` karena keduanya objek yang sama persis; dan penawaran harga harus meminta salinan secara eksplisit dengan `copy.copy`, karena Python tidak pernah menyalin dengan sendirinya.

Kode 6.23 `laundry.py`: antrean yang sama dalam Python, tempat memindahkan pesanan antarantrean berarti memindahkan referensi.

```
# Profitina Laundry di Python: memindahkan pesanan antarantrean berarti
# memindahkan referensi; salinan harus diminta secara eksplisit.
from collections import deque
from copy import copy
from dataclasses import dataclass


@dataclass
class Order:
    number: int
    customer: str
    kilograms: float
    price: int                          # dalam rupiah


def apply_discount(order, percent):     # memutasi pesanan milik pemanggil
    order.price -= order.price * percent // 100


def quote_with_discount(order, percent):
    c = copy(order)                     # bekerja pada salinan eksplisit
    c.price -= c.price * percent // 100
    return c.price


def promote(source, target, number):
    for order in source:
        if order.number == number:
            source.remove(order)        # objeknya tidak dihancurkan ...
            target.append(order)        # ... hanya pindah antrean
            return True
    return False


def show(name, queue):
    print(f"{name}:" + "".join(f"  #{o.number} {o.customer} Rp{o.price}" for o in queue))
```

Kode 6.23 (Lanjutan) *laundry.py: antrean yang sama dalam Python, tempat memindahkan pesanan antarantrean berarti memindahkan referensi.*

```
regular, express = deque(), deque()
regular.append(Order(101, "Sari", 3.0, 21000))
regular.append(Order(102, "Joko", 5.5, 38500))
regular.append(Order(103, "Rina", 2.0, 14000))
show("regular", regular)
watched = regular[1]
promote(regular, express, 102)
show("regular", regular)
show("express", express)
print(f"watched still sees #{watched.number} (same object, new owner): {watched is
express[0]}")
print(f"quote with 10% off: Rp{quote_with_discount(express[0], 10)}, "
      f"stored price still Rp{express[0].price}")
apply_discount(express[0], 10)
print(f"after apply_discount: Rp{express[0].price}")
express.popleft()
print("end of main")
```

```
regular:  #101 Sari Rp21000  #102 Joko Rp38500  #103 Rina Rp14000
regular:  #101 Sari Rp21000  #103 Rina Rp14000
express:  #102 Joko Rp38500
watched still sees #102 (same object, new owner): True
quote with 10% off: Rp34650, stored price still Rp38500
after apply_discount: Rp34650
end of main
```

6.17 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. **Mencampuradukkan `q = p` dengan `*q = *p`.** Yang pertama memindahkan panah, yang kedua menyalin nilai. Tanyakan apakah ada bintang di sisi kiri (Subbab 6.3.1).
2. **Mendeklarasikan beberapa pointer dalam satu baris.** `int* p, q;` mendeklarasikan satu pointer `p` dan satu `int` biasa `q`; bintangnya milik setiap nama, bukan milik tipe. Deklarasikan satu pointer per baris.
3. **Memakai pointer yang belum diinisialisasi.** Selalu inisialisasi, setidaknya dengan `nullptr`.
4. **Mendereferensi tanpa memeriksa null.** Setiap pointer yang bisa bernilai `nullptr`, terutama hasil pencarian, harus diperiksa di setiap jalur yang mendereferensinya.
5. **Berjalan melewati ujung larik atau string.** Pakailah loop setengah terbuka `for (p = first; p != last; ++p)`, jangan pernah `<=` dengan panjangnya, dan kembalikan pointer ke awal sebelum memakainya lagi, sebagaimana diajarkan contoh `"Danaya"`.
6. **Meminta `sizeof` untuk panjang parameter larik.** Di dalam fungsi, hasilnya ukuran pointer. Kirimkan panjangnya, pasangan pointer, atau kontainer.
7. **Mengembalikan alamat variabel lokal,** atau referensi kepadanya. Kembalikan nilai, pointer ke penyimpanan milik pemanggil, atau *smart pointer*.
8. **Lupa `delete`, menghapus dua kali, atau mencampur `delete` dengan `new[]`.** Pasangkan setiap `new` dengan satu `delete` dan setiap `new[]` dengan satu `delete[]`; lebih baik lagi, biarkan `std::vector`, `std::unique_ptr`, dan `std::make_unique<T[]>` yang mengerjakannya.
9. **Memakai pointer, referensi, atau iterator ke dalam `std::vector` sesudah vector itu tumbuh.** Pertumbuhan memindahkan elemen-elemennya. Ambillah alamat sesudah penyisipan terakhir, simpan indeks sebagai gantinya, atau panggil `reserve` lebih dahulu.
10. **Mengira `r = b` memindahkan referensi.** Pernyataan itu menugasi objek yang dirujuk `r` dengan nilai `b`.

11. **Mengirim objek besar berdasarkan nilai "karena lebih sederhana".** `const T&` tidak berbiaya apa pun dan di titik pemanggilan terbaca sama saja.
12. **Menulis siklus `shared_ptr`.** Dua objek yang saling memiliki tidak pernah dihancurkan; jadikan salah satu tautannya `weak_ptr` atau *raw pointer* yang bukan pemilik.
13. **(Java) Membandingkan string atau objek lain dengan `==`.** Itu membandingkan referensi; pakailah `equals`.
14. **(Java, Python) Berharap metode mengikat ulang variabel pemanggil.** Menugasi ulang parameter hanya mengubah salinan referensi milik metode; kembalikan objek baru sebagai gantinya.
15. **(Python) Memakai argumen bawaan yang dapat diubah.** Pakailah `None` sebagai nilai bawaan dan buatlah *list* atau *dictionary*-nya di dalam fungsi.

6.18 Ringkasan Bab

- Memori adalah deretan byte bernomor; **alamat** adalah nomor sebuah byte, dan **pointer** adalah variabel yang nilainya berupa alamat. `&x` mengambil alamat, `*p` mengikutinya, dan dalam deklarasi `T*` berarti "pointer ke `T`".
- Tipe pointer menentukan berapa byte yang dibaca dereferensi dan seberapa jauh aritmetika menggeser: `p + i` berada `i · sizeof(T)` byte lebih jauh (Persamaan (6.1)), dan `q - p` menghitung elemen (Persamaan (6.2)). Aritmetika dan pengurutan hanya terdefinisi di dalam satu larik beserta posisi satu-lewat-ujungnya.
- Nama larik meluruh menjadi pointer ke elemen pertamanya, dan `a[i]` *adalah* `*(a + i)` (Persamaan (6.3)). Larik yang dikirim ke fungsi tiba sebagai pointer dan kehilangan panjangnya. Larik dua dimensi disimpan dalam urutan utama-baris (Persamaan (6.4)).
- `p->m` adalah `(*p).m`. Rantai struktur yang ditautkan pointer `next` menyimpan urutannya di dalam pointer, bukan di memori: gagasan di balik setiap struktur bertaut dalam buku ini.
- `new` dan `new[]` membuat objek di *heap* yang hidup sampai `delete` dan `delete[]`. Larik yang menggandakan diri tumbuh dalam waktu $O(1)$ teramortisasi karena total penyalinannya di bawah $2n$ (Persamaan (6.5)).
- *Dangling pointer*, kebocoran, *double free*, `delete` yang tidak berpasangan, dereferensi null, dan akses di luar batas semuanya adalah ketidaksepakatan antara umur pointer dan umur objeknya. Semuanya termasuk kelemahan paling berbahaya dalam perangkat lunak nyata. AddressSanitizer (`-fsanitize=address`) dan Valgrind menemukannya dalam program yang berjalan dan melaporkan baris yang tepat.
- `nullptr` adalah pointer null yang bertipe. Referensi adalah alias yang wajib diinisialisasi, tidak dapat null, dan tidak dapat dipindahkan.
- *Pass by value* menyalin (dan tidak dapat mengubah pemanggil); *pass by pointer* menyalin alamat (dan dapat, melalui `*`); *pass by reference* membuat alias (dan dapat, secara langsung). Kisah *swap* membuktikannya, dan Persamaan (6.6) menjelaskan mengapa argumen besar yang hanya dibaca ditulis `const T&`.
- `const` yang dibaca dari kanan ke kiri menyatakan apa yang tidak boleh berubah; ketepatan const mengubah perubahan tak disengaja menjadi galat kompilasi.
- Kembalikan nilai, pointer ke penyimpanan milik pemanggil, atau *smart pointer*; jangan pernah alamat variabel lokal.
- `std::unique_ptr` adalah pemilik bawaan objek *heap*; `std::shared_ptr` menghitung pemilik bersama; `std::weak_ptr` mengamati dan memutus siklus. *Raw pointer* tetap layak sebagai pengamat yang bukan pemilik.

- Java mengirim segala sesuatu berdasarkan nilai, dan nilai sebuah objek adalah referensi; Python mengikat nama parameter pada objek milik pemanggil. Di kedua bahasa, fungsi dapat **memutasi** objek bersama yang dapat diubah tetapi tidak dapat **mengikat ulang** variabel pemanggil. *Garbage collection* mencegah *dangling pointer* dan *double free*, tetapi tidak mencegah kebocoran.

6.19 Soal Latihan

6.1 Gambarlah kotak dan panah sesudah setiap pernyataan, lalu berikan nilai `a`, `b`, `*p`, dan `*q` pada akhirnya:

```
int a = 5;  int* p = &a;  int* q = p;  *q = 10;
int b = *p;  p = &b;  *p = 20;  *q = *p + 1;
```

6.2 Sebuah `double d[6]` dimulai di alamat 2000 dan `sizeof(double)` bernilai 8. Berikan alamat `d + 4`, nilai `&d[5] - &d[1]`, dan alamat yang dinyatakan `d + 6`. Manakah dari ketiganya yang boleh didereferensi?

6.3 Sebuah `int g[4][6]` dimulai di alamat 5000. Dengan Persamaan (6.4), tentukan alamat `g[3][2]` dan `g[1][5]`. Elemen mana yang tinggal di alamat 5048?

6.4 Untuk setiap baris, nyatakan apakah baris itu lolos kompilasi, dan bila tidak, mengapa:

```
int x = 1, y = 2;
const int* a = &x;   *a = 5;   a = &y;
int* const b = &x;   *b = 5;   b = &y;
const int* const c = &x;   int& r = *c;
```

6.5 Ramalkan keluarannya, lalu periksa dengan menjalankan program:

```
void f(int a, int& b, int* c) { a = a + 1; b = b + 1; *c = *c + 1; c = &a; *c = 100; }
int main() { int x = 1, y = 1, z = 1; f(x, y, &z); std::cout << x << y << z; }
```

6.6 Fungsi di bawah ini mengandung tiga bug memori yang berbeda. Sebutkan masing-masing, katakan baris mana penyebabnya, dan berikan judul laporan yang akan dicetak AddressSanitizer untuk masing-masing (misalnya `heap-use-after-free`); ASan berhenti pada galat pertama, sehingga dalam praktik Anda menemuinya satu per satu sambil memperbaikinya. Kemudian tulis ulang fungsi itu dengan `std::vector` sehingga tidak ada bug yang mungkin terjadi.

```
int* scores(int n) {
    int* s = new int[n];
    for (int i = 1; i <= n; ++i) s[i] = i * 10;
    int* copy = s;
    delete s;
    return copy;
}
```

6.7 Jelaskan mengapa `for (auto x : v) x *= 2;` membiarkan `std::vector<int> v` tidak berubah, dan berikan dua cara berbeda untuk menggandakan setiap elemen.

6.8 Apa kata kompilator tentang `int& r;`? Apa yang dilakukan `int& r = a; r = b;`? Mengapa C++ melarang referensi bernilai null atau dipindahkan, dan tugas struktur data mana yang karenanya memerlukan pointer?

6.9 Sebuah larik tumbuh mulai dalam keadaan kosong. Hitunglah total banyaknya salinan elemen yang terjadi selama mendorong $n = 1000$ nilai (a) dengan aturan penggandaan pada **Pseudokode 6.1**, dan (b) dengan aturan yang menambah kapasitas sebesar 10 setiap kali penuh. Umumkan (b) untuk menunjukkan bahwa totalnya $\Theta(n^2)$.

6.10 Tulislah `void pushFront(Node*& head, const std::string& title)` yang menyisipkan simpul baru di depan sebuah rantai, dan `void freeAll(Node*& head)` yang menghapus setiap

simpul. Jelaskan mengapa `head` harus dikirim sebagai `Node*&` dan apa yang salah bila memakai `Node*` biasa. Bagaimana Java dan Python memecahkan masalah yang sama?

6.11 Ramalkan apa yang dicetak program Java ini, dan jelaskan setiap barisnya dengan **Gambar 6.10**:

```
static void work(List<String> list) { list.add("B"); list = new ArrayList<>();
list.add("C"); }
public static void main(String[] args) { List<String> x = new ArrayList<>(List.of("A"));
work(x); System.out.println(x); }
```

6.12 Tulislah fungsi yang menghitung faktorial, menghitung x^y untuk eksponen bulat y yang boleh negatif, dan menentukan apakah sebuah bilangan prima; lalu fungsi yang mencetak setiap bilangan prima dalam suatu rentang dengan memanggil fungsi uji prima tadi. Terakhir, tulislah fungsi yang mengembalikan minimum sekaligus maksimum sebuah larik, sekali melalui parameter pointer dan sekali melalui parameter referensi.

6.13 Ramalkan keluaran tiga pemanggilan berturut-turut `log_scan("a")`, `log_scan("b")`, `log_scan("c", [])` untuk `def log_scan(url, seen=[]): seen.append(url); return seen`, lalu perbaiki fungsinya.

6.14 Berikan nilai `use_count()` sesudah setiap baris, dan sebutkan di baris mana `Book` dihancurkan:

```
auto a = std::make_shared<Book>("B4001");
auto b = a;
std::shared_ptr<Book> c;
c = b;
b.reset();
a = nullptr;
std::weak_ptr<Book> w = c;
c.reset();
```

6.15 Dengan pointer saja (tanpa kurung siku), tulislah `reverse(int* first, int* last)` untuk rentang setengah terbuka, `length(const char* s)` yang dihitung dengan pengurangan pointer, dan `int* largest(int* first, int* last)` yang mengembalikan `nullptr` untuk rentang kosong.

6.16 Sebuah program menyimpan `std::vector<Book> catalog;` dan, untuk pencarian cepat berdasarkan pengarang, `std::vector<Book*> byAuthor;` yang berisi pointer ke dalam `catalog`. Buku-buku baru ditambahkan ke `catalog` dari waktu ke waktu. Jelaskan dengan tepat apa yang salah, laporan ASan mana yang akan muncul, dan usulkan tiga perbaikan yang berbeda beserta biaya masing-masing.

6.20 Penyelesaian Soal Latihan

6.1 Sesudah `int a = 5; int* p = &a; int* q = p;` kedua pointer menjangkau `a`. `*q = 10` membuat $a = 10$. `int b = *p` membuat `b` $=10$, sebuah kotak tersendiri. `p = &b` memindahkan panah `p` ke `b`; `q` masih menunjuk `a`. `*p = 20` membuat $b = 20$. `*q = *p + 1` membaca 20 melalui `p` dan menulis 21 melalui `q` ke `a`. Nilai akhirnya: $a = 21$, $b = 20$, $*p = 20$, $*q = 21$.

6.2 Menurut Persamaan (6.1), `d + 4` berada di $2000 + 4 \cdot 8 = 2032$. Menurut Persamaan (6.2), `&d[5] - &d[1]` bernilai $(2040 - 2008)/8 = 4$ elemen. `d + 6` menyatakan 2048, satu lewat ujung. `d + 4` boleh didereferensi (itulah `d[4]`); selisihnya sebuah bilangan, bukan pointer; `d + 6` boleh dihitung dan dibandingkan tetapi tidak pernah boleh didereferensi.

6.3 Dengan $C = 6$ dan `sizeof(int)` $=4$: `g[3][2]` berada di $5000 + (3 \cdot 6 + 2) \cdot 4 = 5080$ dan `g[1][5]` di $5000 + (1 \cdot 6 + 5) \cdot 4 = 5044$. Untuk alamat 5048, $(5048 - 5000)/4 = 12 = 2 \cdot 6 + 0$, jadi elemennya `g[2][0]`, elemen pertama baris 2, tepat sesudah `g[1][5]`.

6.4 `const int* a = &x;` lolos; `*a = 5` tidak (`int` yang ditunjuk bersifat hanya-baca melalui `a`); `a = &y` lolos. `int* const b = &x;` lolos; `*b = 5` lolos; `b = &y` tidak (`b` sendiri const). `const int* const c = &x;` lolos; `int& r = *c;` tidak, karena akan menciptakan nama yang dapat mengubah objek yang dijanjikan hanya-baca; `const int& r = *c;` akan baik-baik saja.

6.5 `a` adalah salinan `x`, sehingga `x` tetap 1. `b` adalah `y`, sehingga `y` menjadi 2. `c` menyimpan `&z`, sehingga `*c = *c + 1` membuat `z` $=2$. Lalu `c = &a` mengarahkan ulang pointer *lokal* ke salinan lokal `a`, dan `*c = 100` hanya mengubah salinan itu. Program mencetak `122`.

6.6 Bug 1: loop menjalankan `i` dari 1 sampai n, sehingga `s[n]` ditulisi, satu lewat ujung: `heap-buffer-overflow` (WRITE berukuran 4, 0 byte sesudah wilayahnya). Bug 2: memori dari `new[]` dibebaskan dengan `delete`: `alloc-dealloc-mismatch (operator new [] vs operator delete)`. Bug 3: fungsi mengembalikan `copy`, yang menunjuk ke dalam blok yang baru saja dibebaskan, sehingga pemakaian pertama oleh pemanggil adalah `heap-use-after-free`. Versi yang aman mengembalikan vector berdasarkan nilai, yang memiliki dan membebaskan memorinya sendiri dan tidak dapat dipakai sesudah lenyap:

```
std::vector<int> scores(int n) {
    std::vector<int> s(n);
    for (int i = 0; i < n; ++i) s[i] = (i + 1) * 10;
    return s;
}
```

6.7 `auto x` menjadikan `x` salinan setiap elemen, dan menggandakan salinan tidak mengubah apa pun di `v`. Pakailah loop melalui referensi, `for (auto& x : v) x *= 2;`, atau melalui indeks, `for (std::size_t i = 0; i < v.size(); ++i) v[i] *= 2;`.

6.8 `int& r;` adalah galat: GCC melaporkan `'r' declared as reference but not initialized`. `int& r = a; r = b;` menugasi `a` dengan nilai `b`. Referensi dirancang sebagai alias tanpa syarat, dan itulah yang membuat kompilator maupun pembaca dapat memercayainya tanpa pemeriksaan. Struktur yang tautannya kadang harus tidak menunjuk apa-apa (simpul terakhir sebuah *list*, anak yang tidak ada pada simpul pohon) atau harus diarahkan ulang selama penyisipan dan penghapusan karenanya memakai pointer.

6.9 (a) Pertumbuhan pada ukuran 1,2,4, … ,512 menyalin $1 + 2 + \cdots + 512 = 1023$ elemen, kurang dari $2n = 2000$, sesuai janji Persamaan (6.5). (b) Dengan kapasitas 10,20,30, …, pertumbuhan terjadi pada ukuran 10,20, … ,990 dan menyalin $10 + 20 + \cdots + 990 = 10 \cdot (1 + 2 + \cdots + 99) = 10 \cdot 4950 = 49500$ elemen. Secara umum, tumbuh dengan konstanta c menjalankan sekitar n/c pertumbuhan yang menyalin $c, 2c, 3c, \ldots$ elemen, dengan total $c \cdot \frac{(n/c)(n/c+1)}{2} \approx \frac{n^2}{2c} = \Theta(n^2)$.

6.10 Fungsi-fungsinya ada di **Kode 6.25** di bawah. `pushFront` harus mengubah variabel `head` milik *pemanggil* agar menunjuk simpul baru; dengan `Node* head` fungsi hanya mengubah salinan pointernya sendiri, dan *list* milik pemanggil tidak akan pernah melihat simpul baru itu (dan simpulnya bocor). `freeAll` menerima `Node*&` agar dapat meninggalkan `head` milik pemanggil bernilai `nullptr` alih-alih menggantung. Java dan Python tidak dapat mengikat ulang variabel pemanggil, sehingga versi mereka *mengembalikan* head yang baru dan pemanggil menulis `head = pushFront(head, t)`; kembaran-kembarannya di folder kode bab ini melakukan tepat itu.

6.11 Program mencetak `[A, B]`. `list.add("B")` mengikuti referensi yang disalin dan memutasi *list* milik pemanggil. `list = new ArrayList<>()` hanya mengikat ulang salinan milik metode; `"C"` masuk ke *list* baru yang dibuang ketika metode selesai.

6.12 **Kode 6.24** memuat semuanya. Faktorial memakai `long long`, yang dapat menampung $20! \approx 2.4 \times 10^{18}$ tetapi tidak 21!. Fungsi pangkat mengalikan $|y|$ kali dan membalik hasilnya untuk y negatif. Uji prima berhenti pada $d^2 \le n$, karena bilangan komposit n selalu mempunyai pembagi yang tidak lebih besar daripada $\sqrt{n}$. Kedua versi `minMax` memperlihatkan gaya pointer (pemanggil menulis `&lo, &hi`, fungsi menulis `*lo`) dan gaya referensi (tanpa simbol di titik pemanggilan).

Kode 6.24 `19_sol_functions.cpp`: penyelesaian Soal 6.12.

```cpp
// Penyelesaian: faktorial, pangkat, bilangan prima, dan dua cara
// mengembalikan dua hasil.
#include <iostream>

long long factorial(int n) {
    long long result = 1;
    for (int k = 2; k <= n; ++k) result *= k;
    return result;
}

double power(double x, int y) {           // x pangkat y, y boleh negatif
    double result = 1.0;
    int steps = (y < 0) ? -y : y;
    for (int k = 0; k < steps; ++k) result *= x;
    return (y < 0) ? 1.0 / result : result;
}

bool isPrime(int n) {
    if (n < 2) return false;
    for (int d = 2; d * d <= n; ++d) {
        if (n % d == 0) return false;
    }
    return true;
}

void printPrimes(int low, int high) {     // memakai ulang isPrime
    for (int n = low; n <= high; ++n) {
        if (isPrime(n)) std::cout << n << ' ';
    }
    std::cout << '\n';
}

void minMaxByPointer(const int* a, int n, int* lo, int* hi) {
    *lo = *hi = a[0];
    for (const int* p = a + 1; p != a + n; ++p) {
        if (*p < *lo) *lo = *p;
        if (*p > *hi) *hi = *p;
    }
}
```

Kode 6.24 (Lanjutan) *19_sol_functions.cpp: penyelesaian Soal 6.12.*

```
void minMaxByReference(const int* a, int n, int& lo, int& hi) {
    lo = hi = a[0];
    for (int i = 1; i < n; ++i) {
        if (a[i] < lo) lo = a[i];
        if (a[i] > hi) hi = a[i];
    }
}

int main() {
    std::cout << "10! = " << factorial(10) << ", 20! = " << factorial(20) << '\n';
    std::cout << "2^10 = " << power(2, 10) << ", 2^-3 = " << power(2, -3) << '\n';
    std::cout << "primes in [1, 50]: ";
    printPrimes(1, 50);

    int years[5] = {2005, 2006, 1980, 2009, 2002};
    int lo = 0, hi = 0;
    minMaxByPointer(years, 5, &lo, &hi);
    std::cout << "by pointer:   oldest " << lo << ", newest " << hi << '\n';
    minMaxByReference(years, 5, lo, hi);
    std::cout << "by reference: oldest " << lo << ", newest " << hi << '\n';
    return 0;
}
```

```
10! = 3628800, 20! = 2432902008176640000
2^10 = 1024, 2^-3 = 0.125
primes in [1, 50]: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
by pointer:   oldest 1980, newest 2009
by reference: oldest 1980, newest 2009
```

6.13 Pemanggilan-pemanggilan itu mengembalikan `['a']`, lalu `['a', 'b']` (*list* bawaan mengingat `'a'`), lalu `['c']` (sebuah *list* baru dikirim secara eksplisit). Perbaikan: `def log_scan(url, seen=None):` dengan `if seen is None: seen = []` sebagai baris pertama.

6.14 Sesudah baris 1: 1. Baris 2: 2. Baris 3: `c` kosong, tetap 2. Baris 4: 3. Baris 5: 2. Baris 6: 1 (hanya `c` yang memiliki). Baris 7: tetap 1; `weak_ptr` tidak dihitung. Baris 8: 0, dan `Book` dihancurkan di baris ini; `w.expired()` kini bernilai `true`.

6.15 Lihat **Kode 6.25**. `reverse` lebih dahulu memundurkan `last`, karena `last` berada satu lewat ujung, lalu menukar dan saling mendekat sampai kedua pointer bertemu atau bersilangan. `length` berjalan sampai terminator dan mengembalikan jarak $p - s$ menurut Persamaan (6.2). `largest` mengembalikan pointer, sehingga pemanggil dapat mengubah elemen yang ditemukan, seperti yang dilakukan program dengan mengisinya 0.

Kode 6.25 `20_sol_pointers.cpp`: penyelesaian Soal 6.10 dan 6.15.

```
// Penyelesaian: membalik dengan dua pointer, strlen lewat pengurangan,
// pointer ke elemen terbesar, dan penyisipan di depan rantai simpul.
#include <cstddef>
#include <iostream>
#include <string>

void reverse(int* first, int* last) {     // membalik [first, last)
    while (first < last) {
        --last;                           // last kini menunjuk elemen nyata
        if (first == last) break;
        int t = *first;
        *first = *last;
        *last = t;
        ++first;
    }
}

std::size_t length(const char* s) {
    const char* p = s;
    while (*p != '\0') ++p;               // berhenti di terminator
    return static_cast<std::size_t>(p - s);
}

int* largest(int* first, int* last) {     // nullptr untuk rentang kosong
    if (first == last) return nullptr;
    int* best = first;
    for (int* p = first + 1; p != last; ++p) {
        if (*p > *best) best = p;
    }
    return best;
}

struct Node {
    std::string title;
    Node* next;
};

// head dikirim melalui referensi: fungsi harus mengubah pointer milik
// pemanggil.
void pushFront(Node*& head, const std::string& title) {
    head = new Node{title, head};         // simpul baru menunjuk head lama
}
```

Kode 6.25 (Lanjutan) *20_sol_pointers.cpp: penyelesaian Soal 6.10 dan 6.15.*

```
void freeAll(Node*& head) {                 // satu delete untuk setiap new
    while (head != nullptr) {
        Node* doomed = head;
        head = head->next;                  // melangkah dulu, baru delete
        delete doomed;
    }
}

int main() {
    int a[6] = {1, 2, 3, 4, 5, 6};
    reverse(a, a + 6);
    std::cout << "reversed:";
    for (int x : a) std::cout << ' ' << x;
    reverse(a + 1, a + 4);                  // hanya a[1..3]
    std::cout << "\nmiddle three reversed back:";
    for (int x : a) std::cout << ' ' << x;

    std::cout << "\nlength(\"Pustaka Ceria\") = " << length("Pustaka Ceria") << '\n';

    int* big = largest(a, a + 6);
    *big = 0;                               // ubah elemen milik pemanggil
    std::cout << "largest set to 0:";
    for (int x : a) std::cout << ' ' << x;
    std::cout << "\nlargest of an empty range is null: " << std::boolalpha
              << (largest(a, a) == nullptr) << '\n';

    Node* head = nullptr;                   // rantai kosong
    pushFront(head, "Cantik Itu Luka");
    pushFront(head, "Laskar Pelangi");
    pushFront(head, "Bumi Manusia");
    std::cout << "chain:";
    for (Node* cur = head; cur != nullptr; cur = cur->next) std::cout << " [" << cur->title
<< ']';
    freeAll(head);
    std::cout << "\nafter freeAll, head is null: " << (head == nullptr) << '\n';
    return 0;
}
```

```
reversed: 6 5 4 3 2 1
middle three reversed back: 6 3 4 5 2 1
length("Pustaka Ceria") = 13
largest set to 0: 0 3 4 5 2 1
largest of an empty range is null: true
chain: [Bumi Manusia] [Laskar Pelangi] [Cantik Itu Luka]
after freeAll, head is null: true
```

6.16 Ketika `catalog` tumbuh melampaui kapasitasnya, `push_back` mengalokasikan blok baru, memindahkan setiap `Book` ke sana, dan membebaskan blok lama (**Gambar 6.6**). Setiap pointer di `byAuthor` lalu menunjuk ke memori yang sudah bebas; pencarian berikutnya adalah *use after free*, yang dilaporkan ASan sebagai `heap-use-after-free` dengan realokasi vector di *stack* “freed by”, persis kasus `stale` di **Kode 6.9**. Perbaikan: (1) simpan *indeks* ke dalam `catalog` alih-alih pointer; indeks tetap sah ketika vector berpindah, dengan biaya satu penjumlahan tambahan per pencarian, $\Theta(1)$. (2) Panggil `catalog.reserve(N)` untuk batas maksimum N yang diketahui sehingga tidak terjadi realokasi; ini murah tetapi gagal secara diam-diam bila N terlampaui. (3) Simpan buku-bukunya sendiri di *heap*, `std::vector<std::unique_ptr<Book>> catalog;`, sehingga pertumbuhan hanya memindahkan pointer pemilik dan setiap `Book` tetap di alamatnya; `byAuthor` lalu dapat menyimpan `Book*` mentah yang bukan pemilik dengan aman, dengan biaya satu alokasi *heap* per buku dan satu indireksi tambahan per akses.

Daftar Pustaka

[1] ISO/IEC 14882:2024, *Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2024.

[2] B. Stroustrup, *A Tour of C++*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2022.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 16 (amortized analysis and dynamic tables).

[4] MITRE, "2025 CWE Top 25 Most Dangerous Software Weaknesses," Dec. 15, 2025. [Online]. Available: https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html. [Accessed: 24-Sep-2026].

[5] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. 2012 USENIX Annual Technical Conference*, Boston, MA, USA, 2012, pp. 309–318.

[6] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. 28th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, San Diego, CA, USA, 2007, pp. 89–100, doi: 10.1145/1250734.1250746.

[7] Valgrind Developers, "Valgrind: Current releases (3.27.1, 20 May 2026)." [Online]. Available: https://valgrind.org/downloads/current.html. [Accessed: 24-Sep-2026].

[8] Microsoft, "AddressSanitizer," *Microsoft C++, C, and Assembler documentation*. [Online]. Available: https://learn.microsoft.com/en-us/cpp/sanitizers/asan. [Accessed: 24-Sep-2026].

[9] H. Sutter, "C++26 is done! — Trip report: March 2026 ISO C++ standards meeting (London Croydon, UK)," Mar. 29, 2026. [Online]. Available: https://herbsutter.com/2026/03/29/c26-is-done-trip-report-march-2026-iso-c-standards-meeting-london-croydon-uk/. [Accessed: 24-Sep-2026].

[10] T. Köppe, "Erroneous behaviour for uninitialized reads," ISO/IEC JTC1/SC22/WG21 paper P2795R5, 2024. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2795r5.html. [Accessed: 24-Sep-2026].

[11] GNU Project, "GCC 16 Release Series" (GCC 16.1 released April 30, 2026). [Online]. Available: https://gcc.gnu.org/gcc-16/. [Accessed: 24-Sep-2026].

[12] B. Stroustrup and H. Sutter, Eds., *C++ Core Guidelines*, sections F.15–F.21 (parameter passing) and R.20–R.24 (smart pointers). [Online]. Available: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines. [Accessed: 24-Sep-2026].

[13] J. Gosling *et al.*, *The Java Language Specification, Java SE 25 Edition*, Oracle, 2025, §4.3 (reference types and values) and §15.12.4 (run-time evaluation of method invocation). [Online]. Available: https://docs.oracle.com/javase/specs/jls/se25/html/. [Accessed: 24-Sep-2026].

[14] Python Software Foundation, "Execution model: naming and binding," *The Python Language Reference*, and "How do I write a function with output parameters (call by reference)?," *Programming FAQ*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/reference/executionmodel.html. [Accessed: 24-Sep-2026].

[15] B. Liskov, "A history of CLU," in *Proc. 2nd ACM SIGPLAN Conf. History of Programming Languages (HOPL-II)*, Cambridge, MA, USA, 1993, pp. 133–147, doi: 10.1145/154766.155367.

[16] G. Lindenmaier and R. Schmelter, "JEP 358: Helpful NullPointerExceptions," OpenJDK, 2019 (delivered in JDK 14, enabled by default since JDK 15). [Online]. Available: https://openjdk.org/jeps/358. [Accessed: 24-Sep-2026].

[17] Python Software Foundation, "gc — Garbage Collector interface," *The Python Standard Library*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/gc.html. [Accessed: 24-Sep-2026].

[18] Profitina, "Profitina — AI Visibility (GEO) platform." [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

Bab 7

Single Linked List

Di akhir bab ini Anda akan sanggup menjelaskan apa itu *linked list* (senarai berantai) dan dari mana gagasannya berasal; menyebutkan dengan tepat kapan *linked list* mengungguli larik dan kapan kalah, dengan ongkos yang ditulis sebagai rumus dan diperiksa dengan pengukur waktu; mendeklarasikan sebuah simpul dalam C++, Java, dan Python serta membaca dan menggambar diagram pointer yang melukiskannya; mengimplementasikan *single linked list* yang dicapai lewat satu pointer `head` beserta seluruh operasi sehari-harinya (inisialisasi, uji kosong, penyisipan di depan, di belakang, sesudah simpul tertentu, dan pada posisi terurut; penghapusan di depan, di belakang, dan berdasarkan kunci; pencarian; penelusuran; menghitung panjang; membalik urutan secara iteratif maupun rekursif; dan memusnahkannya); membangun list yang sama sekali lagi dengan simpul kepala *dummy* dan melihat kasus-kasus khususnya lenyap; menambahkan pointer `tail` dan memahami persis operasi mana yang dipercepatnya dan mana yang tidak; menutup list menjadi lingkaran dan memakainya untuk melayani pembaca secara bergiliran; membangun *stack* dan *queue* dari simpul-simpul; membuktikan bahwa program Anda tidak membocorkan memori dan tidak menyentuh memori yang sudah dikembalikan, dengan Valgrind dan AddressSanitizer; serta memilih di antara *linked list* yang sudah disediakan C++, Java, dan Python. Setiap operasi digambar sebelum dan sesudahnya, setiap ongkos diturunkan, dan setiap program lengkap, dikompilasi, dijalankan, dan diperiksa memorinya.

7.1 Dari Mana Kita Melanjutkan

Bab 5 membangun *stack* dan *queue* di atas sebuah larik. Keduanya bekerja, dan cepat, tetapi keduanya membawa sebuah anggapan diam-diam yang diwarisi dari larik di bawahnya: ukuran maksimum yang harus diputuskan sebelum elemen pertama datang. Ketika perpustakaan Pustaka Ceria memakai *queue* berbasis larik untuk permintaan pemesanan buku, seseorang harus menebak berapa banyak permintaan yang mungkin menunggu sekaligus. Tebakan terlalu kecil membuat *queue* melaporkan “penuh” padahal komputer masih punya memori bergigabita; tebakan terlalu besar membuat program memesan ruang yang tidak akan pernah dipakai.

Bab 6 lalu memberi kita alat yang menghapus tebakan itu: pointer. Kita belajar bahwa sebuah variabel dapat menyimpan *alamat*, bahwa `new` meminta sepotong memori baru kepada sistem pada saat program berjalan dan mengembalikan alamatnya, bahwa `delete` mengembalikannya, bahwa `->` mencapai sebuah medan lewat pointer, dan bahwa sebuah fungsi dapat mengubah pointer milik pemanggil bila pointer itu dilewatkan secara referensi. Bab 6 kebanyakan memakai alat-alat itu satu per satu. Bab ini menyatukannya menjadi struktur data pertama yang seluruhnya dibangun dari pointer: *linked list*.

Gagasannya cukup sederhana untuk dinyatakan dalam satu kalimat. Alih-alih menyimpan elemen-elemen berdampingan dalam satu blok memori, kita menyimpan setiap elemen dalam potongan memorinya sendiri yang kecil, yang disebut *simpul* (*node*), dan setiap simpul mengingat di mana simpul berikutnya berada. Seluruh list kemudian dicapai dari satu pointer ke simpul pertamanya. Tidak ada yang perlu dipesan di muka; setiap elemen baru meminta tepat satu simpul memori pada saat ia datang, dan setiap elemen yang pergi mengembalikan tepat sebanyak itu.

Kesederhanaan itu ada harganya, dan sebagian besar bab ini membahas harga tersebut. Sebuah list tidak dapat melompat ke elemen kelimanya seperti larik; ia harus berjalan ke sana. Setiap simpul membelanjakan bita tambahan untuk pointernya. Dan karena kini kita mengelola memori dengan tangan, satu baris yang salah tempat dapat menghilangkan seluruh rantai simpul atau membaca memori yang bukan lagi milik kita. Kita akan menjumpai setiap masalah itu dengan sengaja, menggambarnya, mengukurnya, dan memperbaikinya.

Bab 8 mengerjakan soal-soal persis dari jenis ini, yang memadukan *stack* dan *queue* dari Bab 5, pointer dari Bab 6, dan *linked list* dari bab ini. Bab 9 melanjutkan kisahnya dengan *double linked list*, yang setiap simpulnya juga mengingat simpul sebelumnya.

7.2 Rantai, Bukan Barisan: Sejarah dan Gagasan

7.2.1 Perburuan harta karun

Bayangkan perburuan harta karun yang diadakan untuk anak-anak pada sebuah pesta sekolah. Panitia tidak menjajarkan semua hadiah di satu meja panjang. Sebagai gantinya, petunjuk pertama dibagikan kepada anak-anak di gerbang: “Carilah di bawah pohon mangga besar.” Di bawah pohon mangga tergeletak sebuah hadiah kecil dan petunjuk kedua: “Sekarang carilah di balik pintu perpustakaan.” Di balik pintu itu ada hadiah lain dan petunjuk ketiga, dan seterusnya, sampai petunjuk terakhir berbunyi “Perburuan selesai.”

Tiga ciri perburuan itu penting bagi kita.

- **Setiap perhentian menyimpan dua hal:** sesuatu yang berharga (hadiahnya) dan arah menuju perhentian berikutnya (petunjuknya).
- **Perhentian-perhentian itu boleh berada di mana saja.** Mereka tidak perlu bersebelahan, berurutan, atau berjarak sama. Pohon mangga boleh berada di ujung lapangan dan pintu perpustakaan tepat di samping gerbang.
- **Rantainya hanya dapat diikuti dari awal.** Anak yang menginginkan hadiah kelima harus mengunjungi empat perhentian pertama, karena hanya petunjuk keempat yang menyebutkan di mana perhentian kelima.

Menambahkan perhentian baru itu mudah: tulislah petunjuk untuknya, lalu ubahlah satu petunjuk yang ada agar menunjuk kepadanya. Menghapus sebuah perhentian sama mudahnya: ubahlah petunjuk yang menunjuk kepadanya agar menunjuk ke perhentian sesudahnya. Tidak ada hadiah yang perlu dipindahkan.

Linked list adalah persis perburuan harta karun ini, dimainkan di dalam memori komputer. Hadiahnya adalah data, petunjuknya adalah pointer, perhentiannya adalah simpul-simpul yang tersebar di mana pun alokator memori menaruhnya, dan kartu yang dibagikan di gerbang adalah sebuah variabel pointer, yang lazimnya bernama `head`.

7.2.2 Asal-usul gagasannya

Linked list adalah salah satu struktur data tertua dalam dunia komputasi. Ia dikembangkan pada 1955 dan 1956 oleh Allen Newell, Cliff Shaw, dan Herbert A. Simon di RAND Corporation sebagai struktur data utama bahasa mereka, Information Processing Language (IPL). Mereka membutuhkannya karena program-program mereka, yaitu Logic Theory Machine yang membuktikan teorema logika simbolik, General Problem Solver, dan sebuah program pemain catur, mengolah struktur simbolik yang ukuran dan bentuknya tidak dapat diketahui sebelumnya. Gambar kotak dan panah yang masih dipakai setiap buku teks, termasuk buku ini, muncul dalam

makalah Newell dan Shaw tahun 1957, "Programming the Logic Theory Machine" [1], dan IPL sendiri kemudian didokumentasikan untuk khalayak luas [2]. Newell dan Simon menerima ACM Turing Award pada 1975, sebagian atas karya pemrosesan list ini.

Pada masa yang hampir bersamaan, Victor Yngve di Massachusetts Institute of Technology memakai *linked list* dalam COMIT, bahasa pemrograman yang ia rancang untuk penelitian linguistik dan penerjemahan mesin [3]. Pada 1958 John McCarthy, juga di MIT, memulai LISP, yang namanya sendiri berarti "*list processor*"; uraiannya pada 1960 [4] menjadikan *linked list*, yang dibangun dari sel bermedan dua, bahan dasar sebuah bahasa pemrograman utuh. Pada awal 1960-an *linked list* sudah menjadi perlengkapan baku, dan *The Art of Computer Programming* karya Knuth kemudian memberinya pembahasan cermat yang hingga kini menjadi rujukan bidang ini [5].

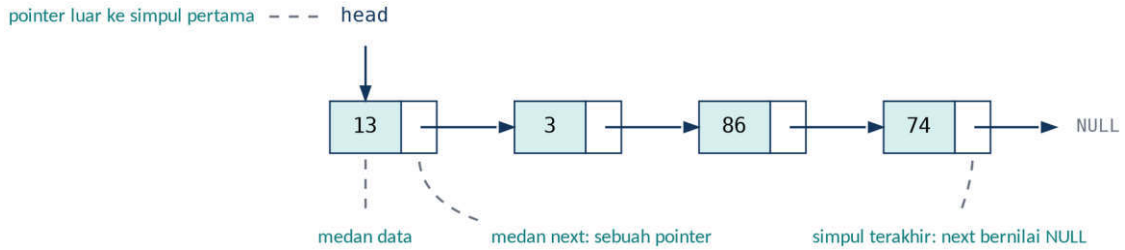
Masalah yang dihadapi para perintis itu adalah masalah yang diuraikan Subbab 7.1: kumpulan data yang tumbuh dan menyusut tanpa dapat diduga selama program berjalan. Tujuh puluh tahun kemudian jawabannya belum berubah: sebuah simpul yang menyimpan sejumlah data dan alamat simpul berikutnya.

7.2.3 Kosakatanya

Sekarang perburuan harta karun itu kita buat tepat.

> **DEFINISI.** Sebuah *linked list* adalah barisan *simpul* yang setiap simpulnya menyimpan satu elemen (*data*-nya) dan sebuah pointer, yang disebut *tautan* (*link*) atau medan `next`, ke simpul berikutnya. List itu dicapai lewat sebuah pointer eksternal ke simpul pertamanya, yang disebut `head`. Medan `next` simpul terakhir berisi pointer nol (`nullptr` di C++, `null` di Java, `None` di Python), yang menandai akhir list. List yang sama sekali tidak memiliki simpul disebut *kosong*, dan `head`-nya bernilai nol.

Gambar 7.1 memperlihatkan list berisi empat simpul yang menyimpan 13, 3, 86, dan 74. Setiap simpul digambar sebagai kotak dengan dua ruang. Ruang kiri adalah medan data. Ruang kanan adalah medan `next`; titik di dalamnya adalah pangkal sebuah panah, dan panah itu berakhir pada simpul yang alamatnya disimpan medan tersebut. Panah simpul terakhir berakhir pada kata NULL. Variabel pointer `head` berada di luar rantai dan menunjuk simpul pertama.



Gambar 7.1 *Single linked list* yang menyimpan 13, 3, 86, dan 74. Setiap simpul memiliki medan data dan medan `next`; `head` menunjuk simpul pertama, dan `next` simpul terakhir bernilai NULL.

Dua kata sifat melukiskan list pada gambar itu, dan keduanya memberi bab ini nama lengkapnya.

- ***Single linked* (bertaut tunggal).** Setiap simpul memiliki *satu* tautan yang menunjuk ke depan. Dari sebuah simpul Anda dapat mencapai simpul sesudahnya dalam satu langkah, tetapi tidak ada jalan kembali ke simpul sebelumnya. Bab 9 menambahkan tautan kedua pada setiap simpul, yang menunjuk ke belakang, sehingga diperoleh *double linked list*.
- ***Non-circular* (tidak melingkar).** Rantainya berujung: simpul terakhir menunjuk NULL. NULL itu bukan gangguan; ia adalah tanda berhenti yang diandalkan setiap gelung dalam bab ini. Di Subbab 7.9 kita akan menyambungkan simpul terakhir kembali ke simpul pertama, sehingga

diperoleh list *melingkar* (*circular*), dan setiap gelung akan memerlukan tanda berhenti yang lain.

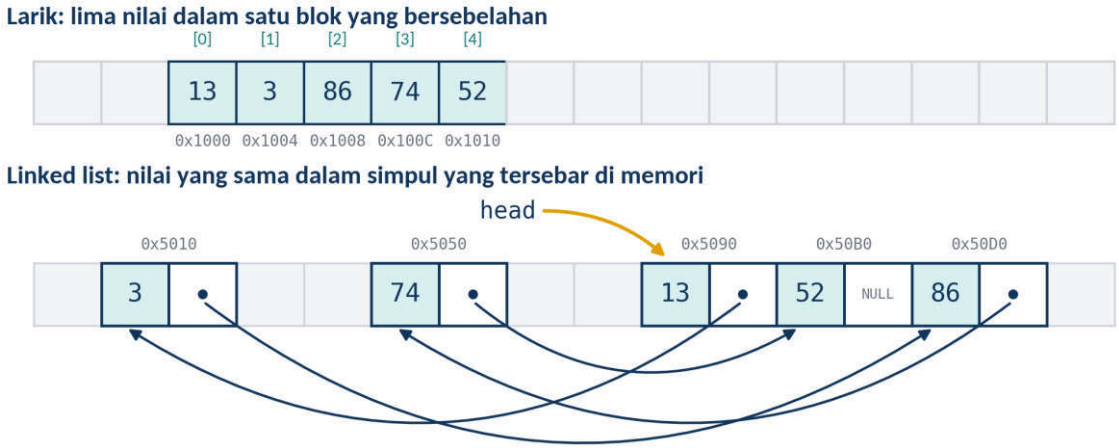
Istilah *single linked list non-circular* (SLLNC) dan *chained list* (senarai berantai) juga lazim dipakai untuk struktur pada **Gambar 7.1**. Dalam buku ini kita cukup menyebut *single linked list* dan menyebut sifat melingkarnya hanya bila berlaku.

> **DENGAN BAHASA SEHARI-HARI.** *Linked list* adalah rantai kotak. Setiap kotak menyimpan sebuah nilai dan sebuah panah ke kotak berikutnya. Anda memegang panah ke kotak pertama. Untuk menemukan apa pun, Anda mulai dari kotak pertama dan mengikuti panah-panahnya.

7.3 Larik versus Linked List

7.3.1 Dua cara menyimpan lima bilangan

Sebelum menulis kode list apa pun, mari kita bandingkan kedua struktur itu berdampingan di dalam memori, sebab setiap keunggulan dan kelemahan *linked list* bersumber dari gambar ini. **Gambar 7.2** memperlihatkan bilangan 13, 3, 86, 74, dan 52 yang disimpan dua kali: dalam larik `int` dan dalam *linked list*.



Gambar 7.2 Lima nilai yang sama dalam larik (satu blok bersebelahan, alamat berurutan) dan dalam *linked list* (simpul di mana pun alokator menaruhnya, disambung oleh pointer `next`). Alamatnya hanya ilustrasi.

Larik menempati satu blok yang bersebelahan. Elemen-elemennya berjarak 4 bita karena sebuah `int` selebar 4 bita, sehingga bila blok itu dimulai pada alamat b dan setiap elemen memakan s bita, elemen i berada pada

$$\text{alamat}(i) = b + i \cdot s \tag{7.1}$$

Bab 1 sudah menjumpai rumus ini; yang penting di sini adalah akibatnya. Untuk membaca `a[3]`, komputer melakukan satu perkalian dan satu penjumlahan lalu langsung menuju ke sana, tak peduli lariknya berisi lima elemen atau lima juta. Itulah *akses acak* (*random access*), dan ongkosnya $O(1)$.

Linked list tidak memiliki rumus semacam itu. Simpul-simpulnya berada pada alamat yang dipilih `new` pada saat setiap simpul dibuat, dan alamat-alamat itu tidak mengikuti pola apa pun: pada **Gambar 7.2** simpul pertama (13) kebetulan berada di `0x5090`, simpul kedua (3) di `0x5010`, jauh di sebelah kirinya, simpul ketiga (86) di `0x50D0`. Satu-satunya catatan tentang letak simpul $k+1$ adalah medan `next` di dalam simpul k. Untuk mencapai simpul berindeks k (dihitung dari 0), kita harus mulai dari `head` dan mengikuti k tautan:

$$T_{\text{capai}}(k) = k \text{ langkah pointer}, \qquad 0 \le k \le n-1 \tag{7.2}$$

Bila setiap indeks sama mungkinnya untuk diminta, rata-rata banyaknya langkah adalah

$$\frac{1}{n}\sum_{k=0}^{n-1} k = \frac{1}{n}\cdot\frac{(n-1)n}{2} = \frac{n-1}{2} \tag{7.3}$$

yang tumbuh sebanding dengan n. Akses berdasarkan posisi pada *linked list* karenanya $\Theta(n)$ secara rata-rata maupun pada kasus terburuk: *akses sekuensial*. Sebuah list berisi sejuta simpul rata-rata membutuhkan sekitar setengah juta langkah untuk mencapai elemen yang dicapai larik dalam satu langkah.

Sekarang balikkan pertanyaannya dan tanyakan berapa ongkos *menyisipkan* nilai baru di depan. Pada larik, setiap elemen yang ada harus bergeser satu tempat ke kanan untuk memberi ruang, karena elemen-elemennya harus tetap bersebelahan: menyisipkan pada indeks 0 dari larik berisi n elemen menggeser n elemen. Membangun larik berisi n elemen dengan selalu menyisipkan di depan karenanya menggeser

$$0 + 1 + 2 + \cdots + (n-1) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \tag{7.4}$$

elemen seluruhnya, yaitu $\Theta(n^2)$. Pada *linked list*, menyisipkan di depan membuat satu simpul dan mengubah dua pointer, berapa pun n: $O(1)$, dan $\Theta(n)$ untuk seluruh n penyisipan. Perbandingan yang sama berlaku untuk menghapus elemen pertama. Secara lebih umum, begitu kita memegang pointer ke sebuah simpul, menyisipkan sesudahnya atau menghapus simpul sesudahnya adalah $O(1)$ pada list, sedangkan operasi yang sepadan di tengah larik menggeser setiap elemen yang menyusul.

7.3.2 Perbandingannya, ciri demi ciri

Tabel 7.1 menghimpun perbedaan-perbedaannya.

Tabel 7.1 Larik dan *single linked list* dibandingkan.

Ciri	Larik	Linked list
Tata letak memori	Satu blok bersebelahan	Simpul-simpul terpisah di mana saja, disambung pointer
Ukuran	Tetap sejak dibuat (`std::vector` tumbuh dengan menyalin)	Tumbuh dan menyusut satu simpul demi satu simpul
Akses ke elemen k	Akses acak, $O(1)$, menurut Persamaan (7.1)	Akses sekuensial, $\Theta(k)$ langkah, menurut Persamaan (7.2)
Sisip atau hapus di depan	$\Theta(n)$: setiap elemen bergeser	$O(1)$: dua perubahan pointer
Sisip atau hapus sesudah posisi yang diketahui	$\Theta(n)$ pergeseran pada kasus terburuk	$O(1)$
Memori tambahan per elemen	Tidak ada	Satu pointer (8 bita pada mesin 64-bit) ditambah catatan alokator
Melepaskan strukturnya	Satu `delete[]` untuk seluruh blok	Setiap simpul harus dilepaskan satu per satu
Keramahan terhadap *cache*	Sangat baik: elemen bertetangga berbagi *cache line*	Buruk bila simpulnya tersebar

Dua baris tabel itu perlu dikomentari karena sering keliru diingat. Pertama, *linked list* kadang disebut berukuran "tak terbatas". Lebih tepat dikatakan bahwa ukurannya hanya dibatasi oleh memori yang dapat diperoleh program, dan bahwa ia tidak pernah perlu menebak ukuran maksimumnya di muka. Kedua, anggapan bahwa larik "tidak dapat dihapus" sedangkan list "mudah dihapus" justru terbalik di C++: larik yang dialokasikan secara dinamis dilepaskan dengan satu `delete[]`, sedangkan list harus mengunjungi dan melepaskan setiap simpul, yang memakan waktu $\Theta(n)$ dan menjadi sumber klasik kebocoran memori. Yang *memang* mudah pada list adalah membuang elemen *satu per satu* tanpa mengusik elemen lainnya.

Baris terakhir tabel, keramahan terhadap *cache*, tidak tampak dalam notasi O tetapi sangat tampak pada mesin sungguhan. Prosesor modern membaca memori dalam blok yang disebut *cache line*, lazimnya 64 bita, dan menyimpan *line* yang baru dipakai dalam *cache*, memori kecil yang sangat cepat [6]. Menelusuri larik memakai setiap bita dari setiap *line* yang dimuatnya, dan prosesor dapat menebak alamat berikutnya lalu mengambilnya lebih awal. Menelusuri list yang simpulnya tersebar memuat *line* baru untuk hampir setiap simpul, dan prosesor tidak dapat mengambil simpul $k+1$ sebelum selesai membaca medan `next` simpul k. Subbab 7.12 mengukur dampaknya; selisihnya dapat mencapai seratus kali lipat.

7.3.3 Memori yang dimakan setiap elemen

Sebuah simpul menyimpan datanya *dan* sebuah pointer. Pada mesin 64-bit yang diandaikan buku ini, sebuah pointer memakan $p = 8$ bita. Alokator memori juga membulatkan setiap permintaan ke atas dan menyimpan sedikit catatan miliknya sendiri; sebutlah tambahan itu a bita per simpul. Bila data satu elemen memakan d bita, maka n elemen memakan

$$M_{\text{larik}} = n\,d, \qquad M_{\text{list}} = n\,(d + p + a) \tag{7.5}$$

bita. Untuk list berisi nilai `int`, $d = 4$, dan aturan penjajaran (*alignment*) menyisipkan pengganjal sehingga pointer dimulai pada kelipatan 8, sehingga sebuah simpul menempati 16 bita sebelum alokator menambahkan apa pun; pada pustaka C GNU, alokator lalu membulatkan setiap permintaan 16 bita menjadi potongan 32 bita. Sebuah `int` di dalam list karenanya berongkos kira-kira 8 kali ukurannya, sedangkan larik `int` berongkos tepat sebesar ukurannya. Bila datanya besar, misalnya catatan buku sebesar 200 bita, pointer 8 bita itu hanyalah pajak kecil. Pegangan praktis yang lahir dari Persamaan (7.5) adalah: *linked list* berisi elemen mungil boros memori; *linked list* berisi catatan besar tidak.

> **RINGKASAN SATU KALIMAT.** Larik cepat untuk *dibaca* di mana saja dan lambat untuk *diubah* di tengah; *linked list* cepat untuk *diubah* di mana pun Anda sudah memegang pointer dan lambat untuk *dicapai* di tempat lain.

7.4 Simpul dan Pointer dalam C++

7.4.1 Mendeklarasikan simpul

Sebuah simpul adalah rekaman bermedan dua, sehingga dalam C++ ia berupa `struct`. Deklarasi tradisionalnya, warisan dari C, memberi struktur itu nama tag lalu alias yang lebih pendek dengan `typedef`:

```
typedef struct MyNode {
    int data;
    struct MyNode *next;
} TNode;
```

Bacalah dari dalam ke luar. Struktur itu memiliki medan bilangan bulat `data` dan medan `next` yang bertipe "pointer ke `struct MyNode`", yaitu pointer ke simpul dari jenis yang sama persis. Struktur yang memuat pointer ke tipenya sendiri disebut *self-referential* (merujuk diri), dan itulah seluruh akal *linked list*: sebuah simpul tidak dapat *memuat* simpul lain (simpul itu akan memuat simpul ketiga, dan seterusnya tanpa akhir, sehingga ukuran strukturnya tak hingga), tetapi ia dapat memuat *alamat* simpul lain, yang selalu 8 bita. `typedef` lalu memungkinkan kita menulis `TNode` alih-alih `struct MyNode` di tempat lain.

Dalam C++, `typedef` itu tidak diperlukan, karena nama `struct` sudah merupakan nama tipe, dan pointernya boleh memakai nama itu selagi struktur masih dideklarasikan:

```
struct Node {
    int data;
    Node *next;
};
```

Kedua bentuk itu melukiskan tata letak memori yang persis sama. Bab ini memakai `TNode` dalam program-program yang mengikuti penyajian klasik di Subbab 7.5 dan Subbab 7.8, dan `Node` yang lebih pendek di tempat lain. Begitu tipe simpul tersedia, kita memerlukan variabel yang akan menunjuk simpul pertama:

```
TNode *head;
```

`head` bukan simpul. Ia adalah pointer, 8 bita yang akan menyimpan alamat simpul pertama, atau `nullptr` selama list masih kosong. Pembedaan antara simpul dan pointer ke simpul itu adalah satu hal terpenting yang harus dipegang teguh dalam bab ini.

7.4.2 Membuat simpul dengan `new`

Simpul dibuat pada saat program berjalan dengan `new`, yang meminta kepada sistem memori secukupnya untuk satu `TNode` dan mengembalikan alamatnya. Medan-medannya kemudian diisi lewat pointer dengan `->`:

```
int newData;
// ...
TNode *newNode;
newNode = new TNode;
newNode->data = newData;
newNode->next = nullptr;
```

Sesudah baris-baris itu, `newNode` menunjuk sebuah simpul di *heap* (wilayah memori yang dikelola `new`) yang datanya `newData` dan `next`-nya `nullptr`: sebuah list sepanjang satu, belum tersambung ke mana pun. C++ memakai kata kunci `nullptr` untuk pointer nol; kode C dan C++ lama menulis `NULL` atau `0`, yang juga berfungsi tetapi dapat tertukar dengan bilangan bulat nol, sehingga kode modern memakai `nullptr`.

Sejak C++11 keempat baris itu dapat ditulis menjadi satu dengan *aggregate initialization*, yang mengisi medan sesuai urutan deklarasinya:

```
TNode *newNode = new TNode{newData, nullptr};
```

Setiap `new` pada akhirnya harus dipasangkan dengan tepat satu `delete` untuk alamat yang sama; kalau tidak, memorinya hilang sampai program berakhir. Kita akan berhati-hati soal ini sejak program pertama.

7.4.3 Tiga simpul dengan tangan

Kode 7.1 membangun list tiga simpul tanpa fungsi list sama sekali, supaya tidak ada yang menyembunyikan apa yang terjadi. Program itu membuat tiga simpul terpisah, menautkannya

dengan menulis ke medan `next` masing-masing, menelusuri rantainya dengan pointer pembantu, membaca data lewat rangkaian panah, dan akhirnya melepaskan setiap simpul.

Kode 7.1 `node_basics.cpp`: tiga simpul dibuat dengan `new`, ditautkan dengan tangan, ditelusuri, dan dilepaskan dengan `delete`.

```
// node_basics.cpp -- three nodes linked by hand, walked, and released.
#include <iostream>

struct Node {
    int data;     // the value this node carries
    Node *next;   // address of the following node, or nullptr at the end
};

int main() {
    // Make three separate nodes on the heap. They are not linked yet.
    Node *a = new Node{13, nullptr};
    Node *b = new Node{3, nullptr};
    Node *c = new Node{86, nullptr};

    // Link them: a -> b -> c -> nullptr.
    a->next = b;
    b->next = c;
    Node *head = a;   // from now on the list is reached only through head

    std::cout << "Three nodes linked by hand: ";
    for (Node *aux = head; aux != nullptr; aux = aux->next) {
        std::cout << aux->data << " -> ";
    }
    std::cout << "NULL\n";

    std::cout << "head->data             = " << head->data << '\n';
    std::cout << "head->next->data       = " << head->next->data << '\n';
    std::cout << "head->next->next->data = " << head->next->next->data << '\n';

    // Release every node: remember the next one BEFORE deleting the current one.
    int released = 0;
    while (head != nullptr) {
        Node *del = head;
        head = head->next;
        delete del;
        ++released;
    }
    std::cout << "Nodes released: " << released << '\n';
    return 0;
}
```

```
Three nodes linked by hand: 13 -> 3 -> 86 -> NULL
head->data             = 13
head->next->data       = 3
head->next->next->data = 86
Nodes released: 3
```

Tiga rincian layak dibaca pelan-pelan.

- Gelung `for (Node *aux = head; aux != nullptr; aux = aux->next)` adalah denyut jantung setiap program *linked list*. Pointer pembantu `aux` (singkatan dari *auxiliary*) mulai pada simpul pertama, mengerjakan tugasnya, lalu pindah ke simpul berikutnya dengan menyalin medan `next` simpul itu ke dirinya sendiri. Ketika ia menyalin `next` simpul terakhir, ia menjadi `nullptr` dan gelungnya berhenti.
- `head->next->next->data` dibaca dari kiri ke kanan, satu panah demi satu panah. `head` menyimpan alamat simpul yang berisi 13. `head->next` adalah medan `next` simpul itu, yang menyimpan alamat simpul berisi 3. `head->next->next` adalah medan `next` simpul *itu*, yang

menyimpan alamat simpul berisi 86, dan `->data` membaca datanya. Setiap `->` adalah satu lompatan sepanjang rantai.

- Gelung pelepasan menyimpan simpul yang akan dihapus di `del`, memajukan `head`, dan baru kemudian menghapus `del`. Menghapus lebih dulu lalu memajukan akan membaca `head->next` dari memori yang bukan lagi milik program. Subbab 7.11 memperlihatkan apa kata pemeriksa memori bila hal itu terjadi.

7.4.4 Cara lain: `malloc` dan `free`

Program C, dan program C++ yang berhubungan dengan pustaka C, mengalokasikan simpul dengan fungsi `malloc` dari `<cstdlib>` (`<stdlib.h>` di C), yang deklarasinya pada dasarnya `void *malloc(size_t size)`. Fungsi itu mengembalikan alamat `size` bita yang belum diinisialisasi, atau pointer nol bila memori habis; pemanggil mengubah alamat itu ke tipe pointer yang tepat lalu mengisi medan-medannya. Memori dari `malloc` dikembalikan dengan `free`. **Kode 7.2** memperlihatkan kedua gaya dalam satu program, dengan rekaman mahasiswa kecil yang bentuknya sama dengan simpul kita.

Kode 7.2 `malloc_node.cpp`: simpul yang dialokasikan dengan `malloc` dan dilepaskan dengan `free`, di samping simpul yang dibuat dengan `new` dan dilepaskan dengan `delete`.

```
// malloc_node.cpp -- the same node made two ways: new/delete and malloc/free.
#include <cstdio>
#include <cstdlib>   // std::malloc, std::free

typedef struct MyStruct {
    int id;
    struct MyStruct *next;
} Student;

void init(Student **p) {        // p is the ADDRESS of the caller's pointer
    *p = nullptr;
}

Student *allocate(int id) {
    Student *p = (Student *) std::malloc(sizeof(Student));
    if (p != nullptr) {         // malloc returns nullptr when memory runs out
        p->next = nullptr;
        p->id = id;
    }
    return p;
}

void add(Student **p, int id) {
    *p = allocate(id);          // changes the caller's pointer through its address
    if (*p != nullptr) std::printf("Student ID: %d\n", (*p)->id);
}

int main() {
    Student *head;
    init(&head);
    add(&head, 13);
    std::printf("sizeof(Student) = %zu bytes\n", sizeof(Student));
    std::free(head);            // memory from malloc goes back with free, never delete

    Student *other = new Student{42, nullptr};   // the C++ way: no cast, no sizeof
    std::printf("Student ID: %d (made with new)\n", other->id);
    delete other;               // memory from new goes back with delete, never free
    return 0;
}
```

```
Student ID: 13
sizeof(Student) = 16 bytes
Student ID: 42 (made with new)
```

Fungsi `init` menerima `Student **p`, sebuah *pointer ke pointer*: alamat variabel `head` milik pemanggil. Menulis `*p = nullptr` karenanya mengubah `head` itu sendiri; begitulah C, yang tidak memiliki referensi, memungkinkan sebuah fungsi mengubah pointer milik pemanggil (Bab 6 membandingkan cara ini dengan melewatkan pointer secara referensi). Perhatikan pula baris keluaran `sizeof(Student) = 16 bytes`: sebuah `int` (4 bita) ditambah sebuah pointer (8 bita) seharusnya 12, tetapi kompilator menyisipkan 4 bita pengganjal supaya pointer dimulai pada alamat yang habis dibagi 8, persis seperti yang dikatakan Subbab 7.3.3.

Ada tiga aturan yang perlu diingat. Memori dari `malloc` dikembalikan dengan `free`, memori dari `new` dikembalikan dengan `delete`, dan keduanya tidak boleh dicampur. `malloc` tidak menjalankan konstruktor, sehingga ia aman hanya untuk rekaman polos seperti `Student`; simpul yang memuat `std::string`, seperti di Subbab 7.9, harus dibuat dengan `new`. Dan hasil `malloc` harus diperiksa terhadap nilai nol sebelum dipakai, sedangkan `new` melaporkan kegagalan dengan melempar `std::bad_alloc`. Di sisa buku ini kita memakai `new` dan `delete`.

7.4.5 Cara membaca diagram pointer dalam bab ini

Setiap operasi di sisa bab ini digambar sebelum dan sesudahnya dengan gaya **Gambar 7.1**, ditambah beberapa kesepakatan.

- Simpul **teal** adalah simpul biasa. Simpul **teal lebih gelap** berbingkai teal adalah simpul yang sedang dilihat oleh pointer pembantu yang bergerak (`aux`, `cur`).
- Simpul **kuning tua** (*amber*) adalah simpul yang baru dibuat dan sedang disisipkan. Simpul **merah** sedang dihapus.
- **Panah kuning tua** adalah pointer yang baru saja diubah oleh langkah yang sedang diperlihatkan. Bila sebuah langkah mengubah dua pointer, angka kecil berwarna kuning tua menunjukkan urutan kode mengubahnya. Urutan itu bukan hiasan: dengan urutan yang salah, operasinya rusak, seperti diperlihatkan Subbab 7.11.
- Nama berhuruf `monospace` di atas sebuah simpul adalah variabel pointer (`head`, `aux`, `prev`, `newNode`), dan panahnya menunjukkan simpul yang ditunjuknya.

7.5 Single Linked List dengan Pointer Head

Linked list lengkap yang paling sederhana menyimpan tepat satu variabel pointer di luar simpul-simpulnya: `head`, yang selalu menunjuk simpul pertama, atau bernilai `nullptr` bila list kosong. Setiap simpul dicapai dari `head` dan tidak dari tempat lain. Itulah aturan pertama pemrograman list:

> **ATURAN HEAD.** Jangan pernah memindahkan `head` untuk berjalan menyusuri list. `head` adalah satu-satunya catatan tentang di mana list dimulai; bila sebuah gelung memajukan `head`, setiap simpul yang dilewatinya menjadi tak terjangkau dan hilang. Berjalanlah dengan pointer pembantu (`aux`, `cur`) dan pindahkan `head` hanya bila simpul pertamanya sendiri berubah.

Dalam subbab ini kita menulis setiap operasi list ber-*head* klasik: `init`, `isEmpty`, `frontInsertion`, `backInsertion`, `show`, `frontDeletion`, `backDeletion`, dan `clear`. Untuk masing-masing, kita lebih dulu menguraikan gagasannya, lalu memberikan prosedurnya dalam pseudokode, menggambarnya sebelum dan sesudah, dan menurunkan ongkosnya. Program C++ lengkapnya,

beserta keluarannya, menyusul di Subbab 7.5.8, dan list yang sama dalam Java dan Python di Subbab 7.5.9.

Fungsi-fungsinya menerima `head` secara referensi, `TNode *&head`, supaya fungsi yang harus mengubah simpul mana yang pertama (menyisipkan atau menghapus di depan) mengubah variabel milik pemanggil, bukan salinannya. Bab 6 memperkenalkan parameter referensi untuk tujuan persis ini. Fungsi yang hanya membaca list menerima `const TNode *head` secara nilai, yang berjanji tidak akan mengubah simpul apa pun.

7.5.1 `init` dan `isEmpty`

List kosong adalah list yang `head`-nya bernilai nol, sehingga inisialisasi cukup satu penugasan dan uji kosong cukup satu pembandingan:

```
void init(TNode *&head) { head = nullptr; }
bool isEmpty(const TNode *head) { return head == nullptr; }
```

Keduanya berjalan dalam waktu $O(1)$. `init` lebih penting daripada kelihatannya: variabel pointer lokal dalam C++ dimulai dengan bit apa pun yang kebetulan ada di memorinya, sehingga `head` yang tidak pernah diinisialisasi tidak berarti "kosong", melainkan "di suatu tempat acak", dan gelung pertama yang menelusurinya akan berkelana ke memori secara acak.

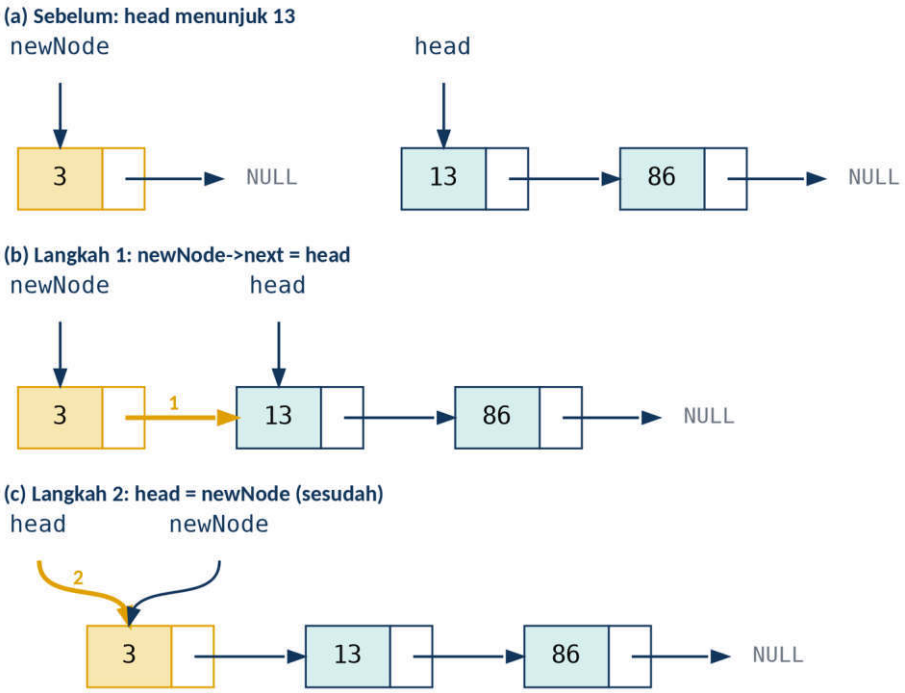
7.5.2 `frontInsertion`: simpul pertama yang baru

Untuk menyisipkan di depan, kita membuat simpul baru, membuatnya menunjuk simpul pertama saat ini, lalu membuat `head` menunjuk simpul baru itu, sehingga simpul baru menjadi yang pertama. **Pseudokode 7.1** menyatakan prosedurnya; L adalah list dan $L.head$ pointer *head*-nya.

Pseudokode 7.1 `FRONT-INSERTION`: sisipkan simpul pertama yang baru; dua penugasan pointer, dengan urutan ini.

```
FRONT-INSERTION(L, x):
 1.  z ← a new node
 2.  z.data ← x
 3.  z.next ← L.head          // langkah 1: tautkan simpul baru ke simpul pertama lama
 4.  L.head ← z               // langkah 2: simpul baru menjadi simpul pertama
```

Gambar 7.3 memperlihatkan kedua langkah itu pada list berisi 13 dan 86 ketika 3 disisipkan. Pada panel (a) simpul baru sudah ada tetapi belum ada yang menunjuknya dan `next`-nya masih NULL. Langkah 1 (panel b) membuatnya menunjuk 13; pada saat ini dua pointer menuju 13, yaitu `head` dan `next` milik simpul baru. Langkah 2 (panel c) memindahkan `head` ke simpul baru, dan list terbaca 3, 13, 86.



Gambar 7.3 `frontInsertion(3)` pada list 13, 86: (a) sebelum, (b) simpul baru ditautkan ke simpul pertama lama, (c) `head` pindah ke simpul baru.

Urutan kedua langkah itu mutlak. Bila baris 4 dijalankan sebelum baris 3, `head` sudah menunjuk simpul baru ketika baris 3 menyalinnya, simpul baru akan menunjuk *dirinya sendiri*, dan simpul-simpul lama 13 dan 86 hilang untuk selamanya. Subbab 7.11 menggambar kesalahan persis ini.

Apa yang terjadi bila list kosong? Maka `L.head` bernilai NULL, baris 3 mengisi `z.next` dengan NULL, dan baris 4 menjadikan `z` simpul pertama sekaligus satu-satunya. Dua baris yang sama benar untuk list kosong, sehingga tidak diperlukan kasus khusus. (Banyak penyajian menguji `isEmpty()` lebih dulu dan menulis `head = newNode; head->next = nullptr;` di cabang itu; hasilnya sama, dan ujinya tidak berbahaya tetapi tidak perlu. Soal Latihan 7.10 meminta Anda memeriksanya dengan cermat.)

Ongkos. Baris 1 sampai 4 masing-masing memakan waktu konstan dan tidak ada yang bergantung pada n, sehingga `frontInsertion` bernilai $O(1)$. Menyisipkan n nilai di depan karenanya berongkos $\Theta(n)$ seluruhnya, dibandingkan $\Theta(n^2)$ dari Persamaan (7.4) untuk larik. Perhatikan satu efek samping yang terlihat: karena setiap nilai baru masuk di depan nilai-nilai sebelumnya, menyisipkan 13 lalu 3 di depan menghasilkan urutan 3, 13, kebalikan dari urutan penyisipan. Penyisipan depan adalah cara kerja *stack*, fakta yang akan dipakai Subbab 7.10.

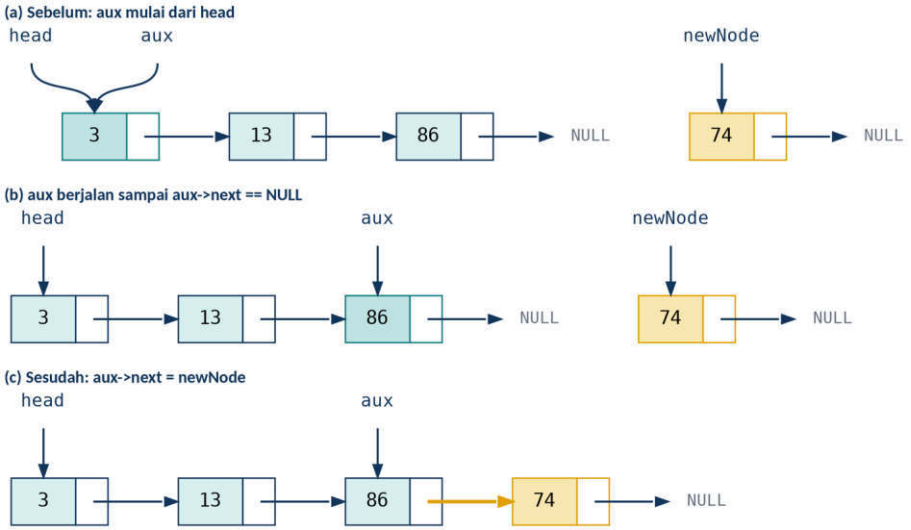
7.5.3 `backInsertion`: simpul terakhir yang baru

Untuk menyisipkan di belakang, kita harus menemukan simpul terakhir, karena simpul itulah yang `next`-nya harus berubah. List ber-*head* tidak tahu di mana simpul terakhirnya, maka ia berjalan ke sana: pointer pembantu `aux` mulai dari `head` dan maju *selama* `aux->next` *tidak NULL*. Ketika gelungnya berhenti, `aux` berada di simpul terakhir, dan satu penugasan menempelkan simpul baru sesudahnya.

Pseudokode 7.2 `BACK-INSERTION` pada list yang hanya memiliki pointer *head*: berjalan ke simpul terakhir, lalu tempelkan.

```
BACK-INSERTION(L, x):
 1.  z ← a new node
 2.  z.data ← x
 3.  z.next ← NIL              // simpul baru akan menjadi simpul terakhir
 4.  if L.head = NIL           // list kosong: simpul baru sekaligus yang pertama
 5.      L.head ← z
 6.  else
 7.      aux ← L.head
 8.      while aux.next ≠ NIL
 9.          aux ← aux.next
10.      aux.next ← z
```

Gambar 7.4 menelusuri penyisipan 74 ke dalam list 3, 13, 86. Panel (a) memperlihatkan `aux` yang mulai pada simpul pertama. Uji gelungnya melihat satu simpul *ke depan*: ia menanyakan apakah simpul sesudah `aux` ada. Sesudah dua kali maju (panel b), `aux` berada di 86 yang `next`-nya NULL, sehingga gelung berhenti di sana, bukan satu langkah lebih jauh. Baris 10 (panel c) menempelkan simpul baru.



Gambar 7.4 `backInsertion(74)` pada list 3, 13, 86: (a) `aux` mulai dari `head`, (b) `aux` berhenti pada simpul yang `next`-nya NULL, (c) `next` simpul itu menjadi simpul baru.

List kosong memang memerlukan cabangnya sendiri di sini (baris 4 dan 5), karena dengan `head` bernilai NULL tidak ada simpul tempat `aux` bermula: mengevaluasi `aux->next` pada `aux` yang nol adalah perilaku tak terdefinisi (*undefined behavior*), yang dalam praktiknya biasanya meruntuhkan program.

Mengapa syarat gelungnya harus `aux.next ≠ NIL` dan bukan `aux ≠ NIL`? Dengan `aux ≠ NIL`, gelung akan berjalan sampai `aux` sendiri menjadi NULL, satu langkah *melewati* simpul terakhir, dan baris 10 lalu mencoba menulis ke simpul yang tidak ada. Berhenti satu simpul lebih awal adalah harga *single linked list*: untuk mengubah tautan *menuju* sebuah posisi, Anda harus berdiri di simpul *sebelum*-nya.

Ongkos. Gelung berjalan sekali untuk setiap simpul kecuali yang terakhir, sehingga pada list berisi n simpul ia melakukan $n-1$ langkah maju: $\Theta(n)$. Membangun list berisi n nilai dengan penyisipan belakang berulang dari list kosong karenanya melakukan $0+1+\cdots+(n-2)$ langkah ditambah kerja konstan per penyisipan, yaitu $\Theta(n^2)$ seluruhnya menurut penjumlahan yang sama dengan Persamaan (7.4). Subbab 7.8 menghapus ongkos ini dengan pointer kedua.

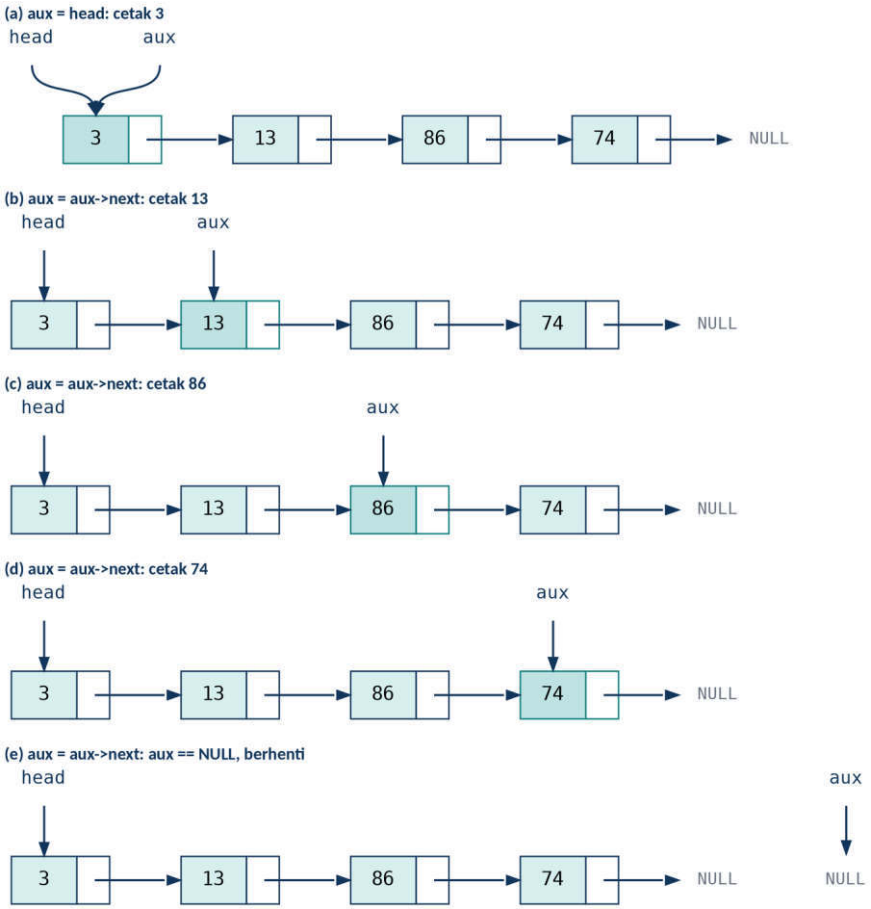
7.5.4 `show`: menyusuri list

Penelusuran (*traversal*) mengunjungi setiap simpul secara berurutan dan mengerjakan sesuatu di masing-masing; `show` mencetak datanya. **Pseudokode 7.3** adalah pola yang diikuti setiap gelung dalam bab ini.

Pseudokode 7.3 `TRAVERSE`: kunjungi setiap simpul dari yang pertama sampai yang terakhir.

```
TRAVERSE(L):
 1.  aux ← L.head
 2.  while aux ≠ NIL
 3.      visit aux            // misalnya cetak aux.data
 4.      aux ← aux.next
```

Gambar 7.5 mengikuti `aux` sepanjang list empat simpul. Di sini uji gelungnya `aux ≠ NIL`, bukan `aux.next ≠ NIL`, karena kita juga ingin mengunjungi simpul terakhir; gelung baru berakhir sesudah `aux` melewati simpul terakhir dan tiba pada NULL (panel e).



Gambar 7.5 Menelusuri list 3, 13, 86, 74: `aux` mengunjungi setiap simpul bergiliran dan gelung berakhir ketika `aux` menjadi NULL; `head` tidak pernah bergerak.

Kedua syarat gelung itu, `aux ≠ NIL` pada **Pseudokode 7.3** dan `aux.next ≠ NIL` pada **Pseudokode 7.2**, adalah dua gelung paling umum dalam pemrograman list, dan salah memilih di antara keduanya adalah galat selisih satu (*off-by-one*) yang paling sering terjadi. Cara sederhana untuk memilih: tanyakan di simpul mana gelung harus *berakhir*. Bila ia harus berakhir sesudah mengunjungi setiap simpul, ujilah `aux`; bila ia harus berakhir *berdiri di* simpul terakhir, ujilah `aux.next`.

Ongkos. Baris 3 dijalankan sekali per simpul, sehingga `show` bernilai $\Theta(n)$, dan itu optimal: mencetak n nilai tidak mungkin memakan kurang dari n langkah.

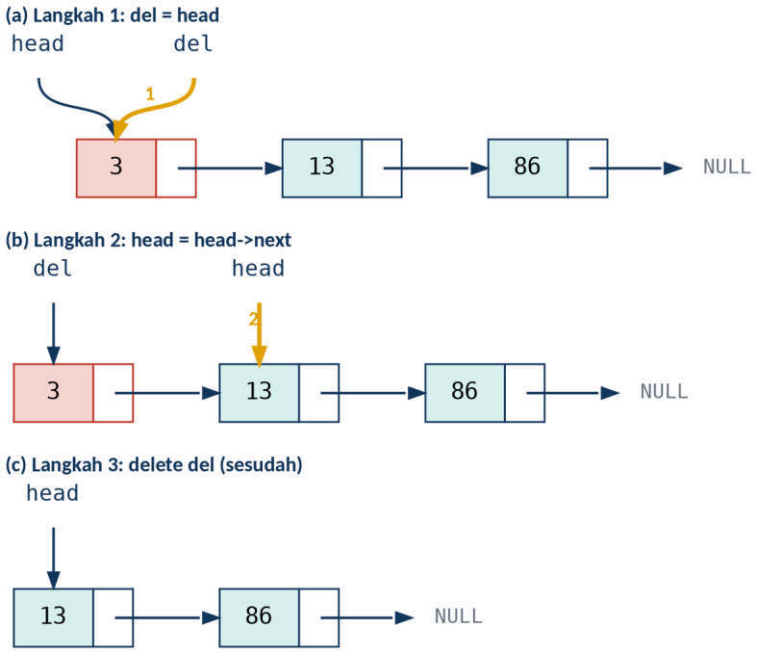
7.5.5 `frontDeletion`: membuang simpul pertama

Untuk menghapus simpul pertama kita harus mengerjakan dua hal: memindahkan `head` ke simpul kedua, supaya list tetap dimulai di tempat yang sah, dan mengembalikan simpul pertama lama dengan `delete`. Namun begitu `head` berpindah, tidak ada lagi yang menunjuk simpul pertama lama, maka kita harus lebih dulu mengingat alamatnya dalam pointer pembantu kedua, `del`.

Pseudokode 7.4 `FRONT-DELETION`: ingat simpul pertama, pindahkan `head` melewatinya, lalu lepaskan.

```
FRONT-DELETION(L):
1.  if L.head = NIL
2.      report "the list is empty" and return
3.  del ← L.head              // langkah 1: ingat simpul pertama
4.  L.head ← L.head.next      // langkah 2: simpul kedua menjadi yang pertama
5.  free del                  // langkah 3: kembalikan simpul pertama lama
```

Gambar 7.6 memperlihatkan ketiga langkah pada list 3, 13, 86. Sesudah langkah 2, simpul merah itu bukan lagi bagian list, tetapi ia masih ada dan `del` masih menunjuknya; langkah 3 melepaskannya.



Gambar 7.6 `frontDeletion()` pada list 3, 13, 86: (a) `del` mengingat simpul pertama, (b) `head` pindah ke simpul kedua, (c) simpul pertama lama dihapus.

Kedua urutan lain untuk sisa pekerjaan itu menggoda, dan keduanya salah. Menghapus `head` lebih dulu lalu menulis `head = head->next` membaca medan `next` dari simpul yang sudah dikembalikan. Memindahkan `head` lebih dulu *tanpa* menyimpan alamat lamanya membuat simpul pertama lama tak terjangkau: ia tidak akan pernah dapat dihapus, sehingga memorinya bocor. Urutan yang benar, simpan, pindahkan, hapus, layak dihafal, karena setiap penghapusan dalam bab ini mengikutinya.

List berisi satu simpul pun ditangani dengan benar: `head.next` bernilai NULL, sehingga baris 4 membuat list kosong. **Ongkos:** $O(1)$.

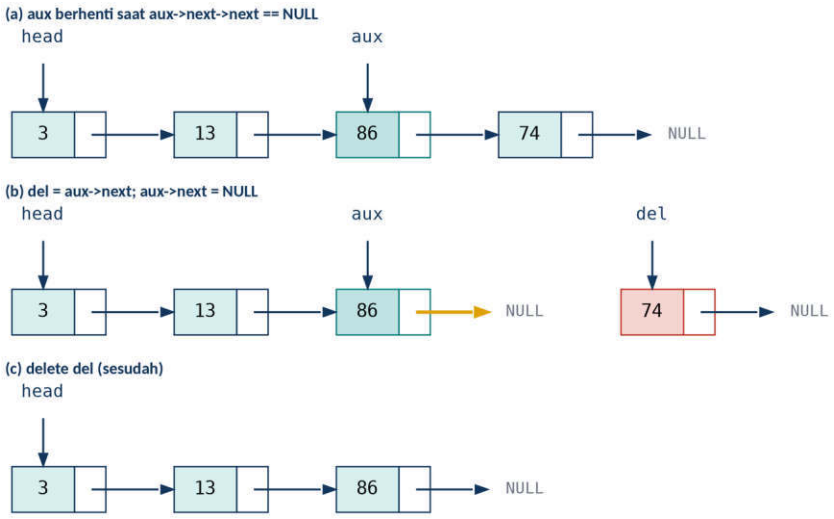
7.5.6 `backDeletion`: membuang simpul terakhir

Untuk menghapus simpul terakhir kita harus mengubah medan `next` simpul *sebelum*-nya, yang menjadi simpul terakhir baru dan harus menunjuk NULL. Simpul bertaut tunggal tidak dapat menoleh ke belakang, maka sekali lagi kita berjalan dari depan. Kali ini pointer pembantu `aux` harus berhenti di simpul kedua dari belakang, yaitu simpul yang `aux.next.next`-nya NULL.

Pseudokode 7.5 `BACK-DELETION` dengan pointer *head* saja: berhenti di simpul kedua dari belakang, putuskan tautannya, lepaskan simpul terakhir.

```
BACK-DELETION(L):
 1.  if L.head = NIL
 2.      report "the list is empty" and return
 3.  if L.head.next = NIL      // tepat satu simpul
 4.      free L.head
 5.      L.head ← NIL
 6.  else
 7.      aux ← L.head
 8.      while aux.next.next ≠ NIL
 9.          aux ← aux.next
10.      del ← aux.next        // simpul terakhir
11.      aux.next ← NIL        // aux menjadi simpul terakhir
12.      free del
```

Gambar 7.7 menelusuri penghapusan 74 dari 3, 13, 86, 74. Pada panel (a) gelung sudah berhenti dengan `aux` di 86, karena simpul sesudah 86 (74) ber-`next` NULL. Panel (b) memutus tautan dari 86 dan mengingat 74 di `del`; panel (c) memperlihatkan list sesudah `del` dilepaskan.



Gambar 7.7 `backDeletion()` pada list 3, 13, 86, 74: (a) `aux` berhenti di simpul kedua dari belakang, (b) simpul terakhir dilepas dari rantai, (c) simpul itu dihapus.

Kasus satu simpul (baris 3 sampai 5) terpisah karena syarat gelung `aux.next.next` akan mendereferensi `aux.next` yang NULL bila hanya ada satu simpul. Pada kasus itu simpul terakhir sekaligus simpul pertama, sehingga `head`-lah yang harus berubah.

Baris 11 adalah baris yang paling sering terlupa. Tanpanya, 86 tetap menunjuk simpul yang dilepaskan baris 12, dan penelusuran berikutnya berjalan lurus ke memori yang sudah dibebaskan. Subbab 7.11 menjalankan galat persis ini di bawah pemeriksa memori.

Ongkos. Gelung melakukan $n-2$ langkah maju pada list berisi $n \geq 2$ simpul, sehingga `backDeletion` bernilai $\Theta(n)$. Berbeda dengan penyisipan belakang, ongkos ini *tidak* akan hilang

ketika kita menambahkan pointer *tail* di Subbab 7.8, dan memahami mengapa demikian adalah salah satu pelajaran pokok bab ini.

7.5.7 `clear`: melepaskan setiap simpul

`clear` mengosongkan list dan mengembalikan setiap simpul kepada sistem. Ia adalah penghapusan depan yang diulang: simpan simpul pertama, pindahkan `head`, hapus simpul yang disimpan, sampai `head` bernilai NULL.

```
void clear(TNode *&head) {
    while (head != nullptr) {
        TNode *del = head;
        head = head->next;
        delete del;
    }
}
```

Gelungnya berjalan sekali per simpul, sehingga `clear` bernilai $\Theta(n)$. Ketika selesai, `head` bernilai `nullptr`, sehingga list itu merupakan list kosong yang sah dan dapat dipakai lagi. Sebuah program harus memanggil `clear` (atau `destroy` yang setara di Subbab 7.6.8) sebelum pointer terakhirnya ke list lenyap; kalau tidak, setiap simpul bocor. Subbab 7.10 memperlihatkan bagaimana kelas C++ dapat membuat hal ini berlangsung otomatis.

7.5.8 Program lengkapnya

Kode 7.3 menyatukan kedelapan operasi itu. Fungsi `main` menyisipkan 13 dan 3 di depan, 86 dan 74 di belakang, mencetak list, menghapus satu simpul di masing-masing ujung, mencetak list lagi, dan akhirnya mengosongkannya.

Kode 7.3 `sll_head.cpp`: *single linked list* dengan pointer *head*, dengan setiap operasi dari Subbab 7.5.

```cpp
// sll_head.cpp -- a singly linked list reached through one pointer, head.
#include <cstdio>

typedef struct MyNode {
    int data;
    struct MyNode *next;
} TNode;

void init(TNode *&head) {                // head is passed by reference
    head = nullptr;
}

bool isEmpty(const TNode *head) {
    return head == nullptr;
}

void frontInsertion(TNode *&head, int newData) {
    TNode *newNode = new TNode;
    newNode->data = newData;
    newNode->next = head;                // 1. new node points at the old first node
    head = newNode;                      // 2. head points at the new node
    std::printf("%d has been inserted by frontInsertion()\n", newData);
}

void backInsertion(TNode *&head, int newData) {
    TNode *newNode = new TNode;
    newNode->data = newData;
    newNode->next = nullptr;             // it will be the last node
    if (isEmpty(head)) {
        head = newNode;
    } else {
        TNode *aux = head;
        while (aux->next != nullptr) {   // walk to the last node
            aux = aux->next;
        }
        aux->next = newNode;
    }
    std::printf("%d has been inserted by backInsertion()\n", newData);
}
```

Kode 7.3 (Lanjutan) *sll_head.cpp: single linked list dengan pointer head, dengan setiap operasi dari Subbab 7.5.*

```
void show(const TNode *head) {
    if (isEmpty(head)) {
        std::printf("The linked list is empty\n");
        return;
    }
    const TNode *aux = head;              // head itself never moves
    while (aux != nullptr) {
        std::printf("%d ", aux->data);
        aux = aux->next;
    }
    std::printf("\n");
}

void frontDeletion(TNode *&head) {
    if (isEmpty(head)) {
        std::printf("The linked list is empty\n");
        return;
    }
    TNode *del = head;
    int d = del->data;
    head = head->next;                    // move head BEFORE the node disappears
    delete del;
    std::printf("%d has been deleted by frontDeletion()\n", d);
}

void backDeletion(TNode *&head) {
    if (isEmpty(head)) {
        std::printf("The linked list is empty\n");
        return;
    }
    int d;
    if (head->next == nullptr) {          // exactly one node
        d = head->data;
        delete head;
        head = nullptr;
    } else {
        TNode *aux = head;
        while (aux->next->next != nullptr) {   // stop at the second-to-last node
            aux = aux->next;
        }
        TNode *del = aux->next;
        d = del->data;
        aux->next = nullptr;              // aux becomes the new last node
        delete del;
    }
    std::printf("%d has been deleted by backDeletion()\n", d);
```

Kode 7.3 (Lanjutan) *sll_head.cpp: single linked list dengan pointer head, dengan setiap operasi dari Subbab 7.5.*

```
}

void clear(TNode *&head) {
    while (head != nullptr) {
        TNode *del = head;
        head = head->next;
        delete del;
    }
    std::printf("The linked list has been cleared\n");
}

int main() {
    TNode *head;
    init(head);

    std::printf("Front insertion...\n");
    frontInsertion(head, 13);
    frontInsertion(head, 3);

    std::printf("\nBack insertion...\n");
    backInsertion(head, 86);
    backInsertion(head, 74);

    std::printf("\nThe content of the linked list is in the following.\n");
    show(head);

    std::printf("\nFront deletion...\n");
    frontDeletion(head);
    std::printf("\nBack deletion...\n");
    backDeletion(head);

    std::printf("\nThe content of the linked list is in the following.\n");
    show(head);

    std::printf("\n");
    clear(head);
    show(head);
    return 0;
}
```

```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()

Back insertion...
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()

The content of the linked list is in the following.
3 13 86 74

Front deletion...
3 has been deleted by frontDeletion()

Back deletion...
74 has been deleted by backDeletion()

The content of the linked list is in the following.
13 86

The linked list has been cleared
The linked list is empty
```

Ikutilah list itu sepanjang eksekusi. Sesudah `frontInsertion(13)` isinya 13; sesudah `frontInsertion(3)` isinya 3, 13, karena penyisipan depan menaruh setiap nilai di depan nilai-nilai sebelumnya. Penyisipan belakang lalu menambahkan 86 dan 74, sehingga diperoleh 3, 13, 86, 74. `frontDeletion` membuang 3 dan `backDeletion` membuang 74, menyisakan 13, 86. Akhirnya `clear` melepaskan dua simpul terakhir dan `show` melaporkan list kosong. Subbab 7.11 menjalankan program ini di bawah Valgrind untuk memastikan bahwa keempat simpul telah dilepaskan.

Tabel 7.2 meringkas ongkos-ongkos yang kita turunkan.

Tabel 7.2 Waktu jalan setiap operasi list ber-*head* dengan n simpul.

Operasi	Ongkos	Alasan
`init`, `isEmpty`	$O(1)$	satu penugasan atau pembandingan
`frontInsertion`	$O(1)$	dua penugasan pointer (**Pseudokode 7.1**)
`backInsertion`	$\Theta(n)$	berjalan ke simpul terakhir (**Pseudokode 7.2**)
`show` (penelusuran)	$\Theta(n)$	mengunjungi setiap simpul (**Pseudokode 7.3**)
`frontDeletion`	$O(1)$	simpan, pindahkan `head`, hapus (**Pseudokode 7.4**)
`backDeletion`	$\Theta(n)$	berjalan ke simpul kedua dari belakang (**Pseudokode 7.5**)
`clear`	$\Theta(n)$	menghapus setiap simpul

7.5.9 List yang sama dalam Java dan Python

Java dan Python tidak memiliki pointer dalam pengertian C++, tetapi setiap variabel bertipe kelas menyimpan sebuah *referensi*, yang berperilaku seperti pointer yang tidak pernah dapat dipakai untuk aritmetika dan tidak pernah dapat menunjuk memori yang sudah dibebaskan. Sebuah simpul karenanya adalah kelas biasa dengan medan data dan medan bertipe kelas simpul itu sendiri. **Kode 7.4** adalah versi Java dari **Kode 7.3**; kelas simpulnya bersarang di dalam program, dan list-nya menjadi kelas `SinglyLinkedList` yang medan privatnya, `head`, memainkan peran parameter referensi C++.

Kode 7.4 `SllHead.java`: *single linked list* berreferensi *head* dalam Java.

```java
// SllHead.java -- a singly linked list reached through one reference, head.
public class SllHead {
    static class Node {
        int data;
        Node next;

        Node(int data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    static class SinglyLinkedList {
        private Node head;                    // null means the list is empty

        SinglyLinkedList() {                  // init
            head = null;
        }

        boolean isEmpty() {
            return head == null;
        }

        void frontInsertion(int newData) {
            head = new Node(newData, head);   // new node points at the old first node
            System.out.println(newData + " has been inserted by frontInsertion()");
        }

        void backInsertion(int newData) {
            Node newNode = new Node(newData, null);
            if (isEmpty()) {
                head = newNode;
            } else {
                Node aux = head;
                while (aux.next != null) {    // walk to the last node
                    aux = aux.next;
                }
                aux.next = newNode;
            }
            System.out.println(newData + " has been inserted by backInsertion()");
        }
```

Kode 7.4 (Lanjutan) *SllHead.java: single linked list berreferensi head dalam Java.*

```
    void show() {
        if (isEmpty()) {
            System.out.println("The linked list is empty");
            return;
        }
        StringBuilder sb = new StringBuilder();
        for (Node aux = head; aux != null; aux = aux.next) {
            sb.append(aux.data).append(' ');
        }
        System.out.println(sb);
    }

    void frontDeletion() {
        if (isEmpty()) {
            System.out.println("The linked list is empty");
            return;
        }
        int d = head.data;
        head = head.next;                  // the old first node becomes garbage
        System.out.println(d + " has been deleted by frontDeletion()");
    }

    void backDeletion() {
        if (isEmpty()) {
            System.out.println("The linked list is empty");
            return;
        }
        int d;
        if (head.next == null) {           // exactly one node
            d = head.data;
            head = null;
        } else {
            Node aux = head;
            while (aux.next.next != null) {   // stop at the second-to-last node
                aux = aux.next;
            }
            d = aux.next.data;
            aux.next = null;               // aux becomes the new last node
        }
        System.out.println(d + " has been deleted by backDeletion()");
    }
```

Kode 7.4 (Lanjutan) *SllHead.java: single linked list berreferensi head dalam Java.*

```
        void clear() {
            head = null;                        // every node is now unreachable
            System.out.println("The linked list has been cleared");
        }
    }

    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();

        System.out.println("Front insertion...");
        list.frontInsertion(13);
        list.frontInsertion(3);

        System.out.println("\nBack insertion...");
        list.backInsertion(86);
        list.backInsertion(74);

        System.out.println("\nThe content of the linked list is in the following.");
        list.show();

        System.out.println("\nFront deletion...");
        list.frontDeletion();
        System.out.println("\nBack deletion...");
        list.backDeletion();

        System.out.println("\nThe content of the linked list is in the following.");
        list.show();

        System.out.println();
        list.clear();
        list.show();
    }
}
```

Kode 7.5 adalah versi Python-nya. `__slots__` memberi tahu Python bahwa sebuah `Node` memiliki tepat dua atribut, `data` dan `next`; hal ini menghemat memori (tidak ada kamus per objek) dan mengubah atribut yang salah eja seperti `node.nxt = ...` menjadi galat, alih-alih diam-diam menciptakan atribut baru.

Kode 7.5 `sll_head.py`: *single linked list* berreferensi *head* dalam Python.

```
# sll_head.py -- a singly linked list reached through one reference, head.
class Node:
    __slots__ = ("data", "next")

    def __init__(self, data, next=None):
        self.data = data
        self.next = next


class SinglyLinkedList:
    def __init__(self):                         # init
        self.head = None                        # None means the list is empty

    def is_empty(self):
        return self.head is None

    def front_insertion(self, new_data):
        self.head = Node(new_data, self.head)  # new node points at the old first node
        print(f"{new_data} has been inserted by frontInsertion()")

    def back_insertion(self, new_data):
        new_node = Node(new_data)
        if self.is_empty():
            self.head = new_node
        else:
            aux = self.head
            while aux.next is not None:         # walk to the last node
                aux = aux.next
            aux.next = new_node
        print(f"{new_data} has been inserted by backInsertion()")

    def show(self):
        if self.is_empty():
            print("The linked list is empty")
            return
        parts = []
        aux = self.head
        while aux is not None:
            parts.append(f"{aux.data} ")
            aux = aux.next
        print("".join(parts))
```

Kode 7.5 (Lanjutan) *sll_head.py: single linked list berreferensi head dalam Python.*

```
    def front_deletion(self):
        if self.is_empty():
            print("The linked list is empty")
            return
        d = self.head.data
        self.head = self.head.next             # the old first node becomes garbage
        print(f"{d} has been deleted by frontDeletion()")

    def back_deletion(self):
        if self.is_empty():
            print("The linked list is empty")
            return
        if self.head.next is None:              # exactly one node
            d = self.head.data
            self.head = None
        else:
            aux = self.head
            while aux.next.next is not None:    # stop at the second-to-last node
                aux = aux.next
            d = aux.next.data
            aux.next = None                     # aux becomes the new last node
        print(f"{d} has been deleted by backDeletion()")

    def clear(self):
        self.head = None                        # every node is now unreachable
        print("The linked list has been cleared")


def main():
    lst = SinglyLinkedList()

    print("Front insertion...")
    lst.front_insertion(13)
    lst.front_insertion(3)

    print("\nBack insertion...")
    lst.back_insertion(86)
    lst.back_insertion(74)

    print("\nThe content of the linked list is in the following.")
    lst.show()
```

Kode 7.5 (Lanjutan) *sll_head.py: single linked list berreferensi head dalam Python.*

```
    print("\nFront deletion...")
    lst.front_deletion()
    print("\nBack deletion...")
    lst.back_deletion()

    print("\nThe content of the linked list is in the following.")
    lst.show()

    print()
    lst.clear()
    lst.show()


if __name__ == "__main__":
    main()
```

Kedua program mencetak keluaran yang persis sama dengan **Kode 7.3**. Dua perbedaan dari C++ menonjol. Pertama, tidak ada `delete`: ketika `frontDeletion` memindahkan `head` melewati simpul pertama, simpul itu menjadi tak terjangkau, dan *garbage collector* (pengumpul sampah)

mengambil kembali memorinya beberapa saat kemudian. Dengan alasan yang sama, `clear` cukup satu penugasan, `head = null`; semua simpul menjadi tak terjangkau sekaligus. Kedua, galat khas C++ di Subbab 7.11, membaca simpul yang sudah dibebaskan atau membebaskan simpul dua kali, tidak dapat terjadi, karena sebuah referensi tidak pernah dibiarkan menunjuk memori yang sudah dibebaskan. Yang *tetap* dapat terjadi di Java dan Python adalah kehilangan simpul yang ingin Anda simpan (misalnya dengan memindahkan `head` di dalam gelung penelusuran) dan menahan simpul yang ingin Anda buang (satu referensi yang terlupa membuat seluruh rantai tetap hidup), sehingga aturan *head* berlaku dalam ketiga bahasa.

7.6 Operasi Lain: Pencarian, Panjang, Penyisipan di Tengah, Penghapusan Berdasarkan Kunci, Pembalikan

Kedelapan operasi di Subbab 7.5 hanya bekerja pada kedua ujung list. Program sungguhan juga perlu menemukan sebuah nilai, menghitung simpul, menyisipkan atau menghapus di tengah, menjaga list tetap terurut, dan membalik urutannya. Subbab ini menambahkan semua itu pada list ber-*head*. Program lengkapnya, **Kode 7.6**, beserta keluarannya disajikan di Subbab 7.6.9; subbab-subbab sebelumnya menjelaskan setiap fungsi secara bergiliran. Tipe simpul di sana adalah `struct Node` yang lebih pendek dari Subbab 7.4.1.

7.6.1 `search`

Mencari di dalam *linked list* adalah pencarian sekuensial dari Bab 2 yang ditulis dengan pointer alih-alih indeks: berjalan dari simpul pertama dan berhenti pada simpul pertama yang datanya sama dengan kunci. **Pseudokode 7.6** memberikan prosedurnya.

Pseudokode 7.6 `SEARCH`: kembalikan simpul pertama yang datanya sama dengan `key`, atau NIL.

```
SEARCH(L, key):
 1.  aux ← L.head
 2.  while aux ≠ NIL and aux.data ≠ key
 3.      aux ← aux.next
 4.  return aux              // simpul yang berisi key, atau NIL bila tidak ada
```

Urutan kedua uji pada baris 2 penting. Ketika `aux` bernilai NIL, uji `aux.data ≠ key` akan membaca lewat pointer nol; karena `and` dalam C++ (`&&`) mengevaluasi operan kanannya hanya bila operan kirinya benar (*short-circuit evaluation*, evaluasi hubung-singkat), menulis `aux ≠ NIL` lebih dulu menjamin bahwa `aux.data` hanya dibaca untuk simpul yang sungguh ada.

Pada **Kode 7.6** fungsi itu mengembalikan simpulnya, yang lebih berguna daripada sekadar ya atau tidak: pemanggil dapat membaca datanya, mengubahnya, atau meneruskannya ke `insertAfter`. Fungsi itu juga melaporkan indeks simpulnya lewat parameter referensi, dan mengembalikan -1 bila kuncinya tidak ada.

Ongkos. Pada kasus terburuk (kuncinya ada di simpul terakhir atau tidak ada) gelung mengunjungi seluruh n simpul, sehingga `search` bernilai $\Theta(n)$ pada kasus terburuk dan $O(1)$ pada kasus terbaik. Berbeda dengan larik, *linked list* yang terurut *tidak* memungkinkan pencarian biner berjalan dalam waktu $O(\lg n)$: pencarian biner melompat ke elemen tengah, dan menurut Persamaan (7.2) sekadar mencapai tengah list saja sudah berongkos $n/2$ langkah.

7.6.2 `length`, secara iteratif dan rekursif

Panjang secara iteratif adalah penelusuran yang menghitung. Panjang secara rekursif bertumpu pada sebuah definisi list yang layak dinyatakan tersendiri:

> **DEFINISI (rekursif).** Sebuah list adalah *kosong* (pointernya NULL), atau sebuah *simpul pertama* yang diikuti sebuah *list* (list yang dimulai pada `next` simpul pertama).

Definisi itu langsung memberikan rumus rekursif untuk panjang $L(p)$ dari list yang dimulai pada pointer p:

$$L(p) = \begin{cases} 0 & \text{bila } p = \text{NULL} \\ 1 + L(p.\text{next}) & \text{selainnya} \end{cases} \tag{7.6}$$

dan Persamaan (7.6) diterjemahkan ke dalam C++ baris demi baris:

```
int lengthRec(const Node *head) {
    if (head == nullptr) return 0;         // the empty list has length 0
    return 1 + lengthRec(head->next);      // one node plus the rest
}
```

Untuk list 20, 30, 40 pemanggilannya terurai sebagai $L(20) = 1 + L(30) = 1 + (1 + L(40)) = 1 + (1 + (1 + L(\text{NULL}))) = 1 + 1 + 1 + 0 = 3$. Setiap pemanggilan mengerjakan kerja konstan dan ada $n + 1$ pemanggilan, sehingga kedua versi memakan waktu $\Theta(n)$. Perbedaannya terletak pada memori. Versi iteratif memakai satu pencacah. Versi rekursif menyimpan $n + 1$ pemanggilan yang menunggu di *call stack* (tumpukan pemanggilan) pada titik terdalamnya, masing-masing dengan salinan `head` dan alamat kembalinya sendiri, sehingga ia memakai memori tambahan $\Theta(n)$. *Call stack* lazimnya dibatasi beberapa megabita (8 MB adalah bawaan yang umum di Linux), sehingga penelusuran rekursif atas list berisi beberapa juta simpul dapat meluap dan meruntuhkan program. Bab 10 mempelajari rekursi secara mendalam; untuk list, pelajarannya adalah bahwa rekursi mengungkapkan definisinya dengan indah dan iterasi adalah pilihan yang aman untuk list yang panjang.

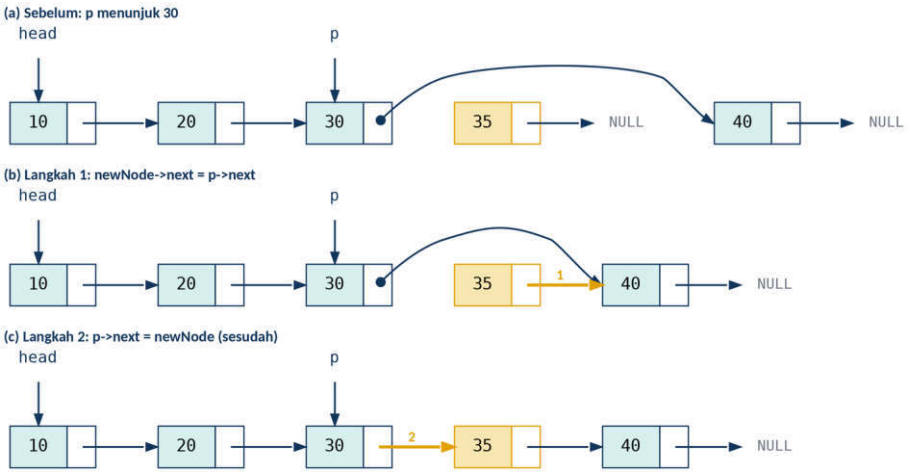
7.6.3 `insertAfter`: penyisipan di tengah

Bila diberikan pointer `p` ke sebuah simpul di dalam list, menyisipkan nilai baru sesudahnya sama sekali tidak memerlukan perjalanan: simpul baru harus menunjuk penerus lama `p`, dan `p` harus menunjuk simpul baru, seperti dinyatakan **Pseudokode 7.7**.

Pseudokode 7.7 `INSERT-AFTER`: sisipkan simpul baru tepat sesudah simpul `p` dalam waktu $O(1)$.

```
INSERT-AFTER(p, x):           // p adalah simpul list, bukan NIL
 1.  z ← a new node
 2.  z.data ← x
 3.  z.next ← p.next          // langkah 1: simpul baru menunjuk penerus p
 4.  p.next ← z               // langkah 2: p menunjuk simpul baru
```

Gambar 7.8 menyisipkan 35 sesudah simpul berisi 30. Pada panel (b) simpul baru sudah menunjuk 40 sementara 30 juga masih menunjuk 40; tidak ada yang rusak, karena list yang dibaca dari `head` belum berubah. Hanya langkah 2 (panel c) yang menjadikan simpul baru bagian dari list. Seperti pada penyisipan depan, membalik kedua langkah akan membuat simpul baru menunjuk dirinya sendiri dan memutus 40 beserta semua yang sesudahnya.



Gambar 7.8 `insertAfter(p, 35)` dengan `p` di 30: (a) sebelum, (b) simpul baru menunjuk penerus `p`, (c) `p` menunjuk simpul baru.

`insertAfter` bernilai $O(1)$. Operasi inilah yang memberi *linked list* reputasi penyisipan yang murah; perhatikan, meskipun demikian, bahwa ia murah hanya karena pemanggil sudah memegang `p`. Menemukan `p` sejak awal biasanya memerlukan `search`, yang bernilai $\Theta(n)$.

Menyisipkan *sebelum* simpul tertentu lebih sulit pada *single linked list*, karena simpul sebelumnya adalah simpul yang `next`-nya harus berubah dan kita tidak dapat melangkah mundur ke sana. Soal Latihan 7.9 memecahkan masalah itu dengan mencari memakai pointer yang tertinggal satu langkah.

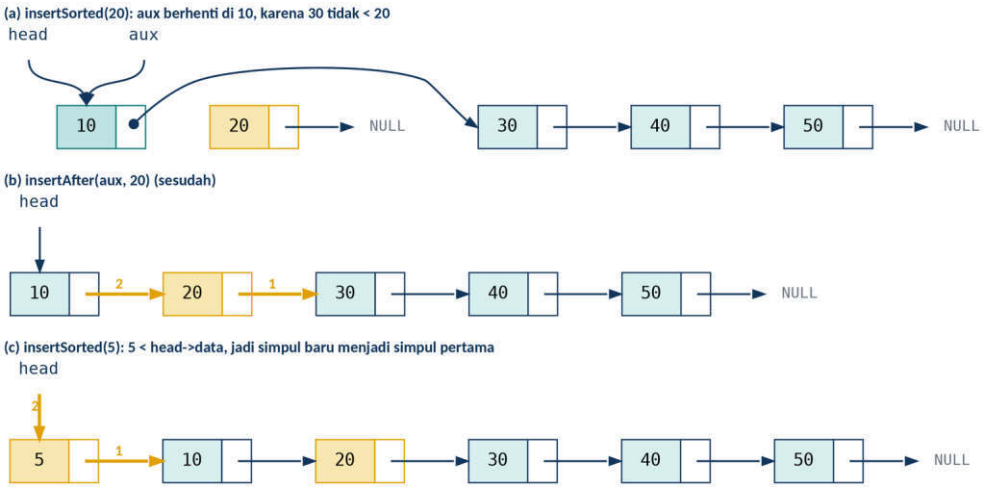
7.6.4 `insertSorted`: menjaga list tetap terurut

List yang harus tetap terurut naik (katalog yang diurutkan menurut nomor buku, misalnya) memerlukan penyisipan di tempat yang tepat. Kita menjalankan pointer `aux` ke depan selama nilai simpul *berikutnya* lebih kecil daripada nilai baru; `aux` lalu berdiri di simpul terakhir yang lebih kecil, dan kita menyisipkan sesudahnya. Satu kasus memerlukan penanganan tersendiri: bila list kosong, atau nilai baru lebih kecil daripada nilai pertama, simpul baru menjadi simpul pertama yang baru dan `head` harus berubah. **Pseudokode 7.8** menangani kasus itu lebih dulu.

Pseudokode 7.8 `INSERT-SORTED`: sisipkan `x` sehingga list tetap terurut naik.

```
INSERT-SORTED(L, x):
 1.  if L.head = NIL or x < L.head.data
 2.      FRONT-INSERTION(L, x)
 3.      return
 4.  aux ← L.head
 5.  while aux.next ≠ NIL and aux.next.data < x
 6.      aux ← aux.next
 7.  INSERT-AFTER(aux, x)
```

Gambar 7.9 memperlihatkan kedua kasus. Menyisipkan 20 ke dalam 10, 30, 40, 50 menghentikan `aux` di 10, karena simpul sesudah 10 berisi 30, yang tidak lebih kecil daripada 20 (panel a), dan `insertAfter` mengerjakan sisanya (panel b). Menyisipkan 5 jatuh ke kasus khusus pada baris 1, sehingga 5 menjadi simpul pertama (panel c).



Gambar 7.9 `insertSorted`: (a) untuk 20, `aux` berhenti di simpul terakhir yang lebih kecil daripada 20, (b) simpul baru disisipkan sesudahnya, (c) 5 lebih kecil daripada setiap nilai dan menjadi simpul pertama.

Karena baris 5 memakai `<`, nilai yang sama dengan nilai-nilai yang sudah ada disisipkan sesudah semuanya, sehingga kunci-kunci yang sama mempertahankan urutan kedatangannya: penyisipannya *stabil*, dalam pengertian Bab 3. Menyisipkan n nilai satu per satu dengan `insertSorted` sesungguhnya adalah *insertion sort* yang dikerjakan pada list, dengan kasus terburuk $\Theta(n^2)$ yang sama, tetapi tanpa satu elemen pun yang digeser.

Ongkos. Pada kasus terburuk (nilai barunya paling besar) gelung mengunjungi setiap simpul: $\Theta(n)$.

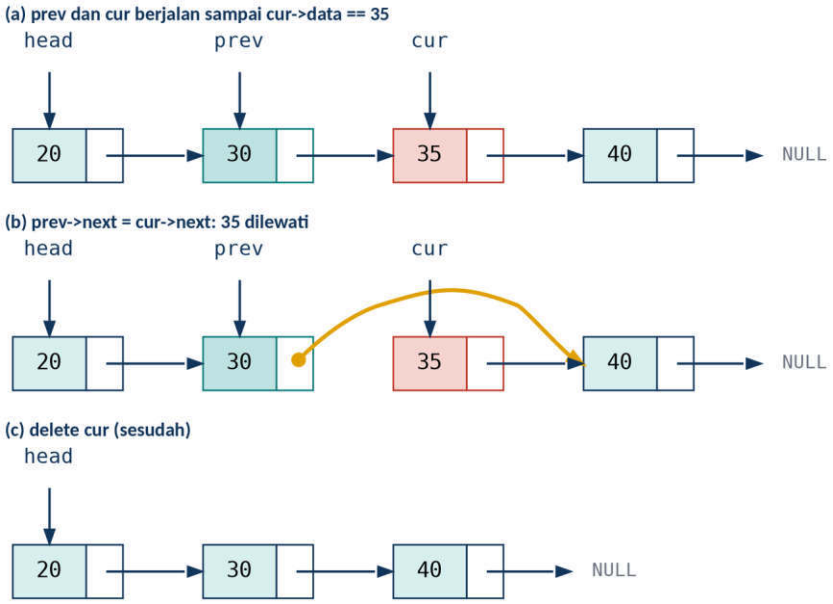
7.6.5 `deleteKey`: penghapusan berdasarkan nilai

Untuk menghapus simpul pertama yang berisi kunci tertentu, kita harus menemukan simpul itu *dan* simpul sebelumnya, karena simpul sebelumnya adalah yang `next`-nya harus berubah. Teknik bakunya, **Pseudokode 7.9**, menjalankan dua pointer bersamaan, `cur` pada calon dan `prev` satu simpul di belakangnya:

Pseudokode 7.9 `DELETE-KEY`: buang simpul pertama yang datanya sama dengan `key`, dengan pointer pengekor `prev`.

```
DELETE-KEY(L, key):
 1.  prev ← NIL
 2.  cur ← L.head
 3.  while cur ≠ NIL and cur.data ≠ key
 4.      prev ← cur
 5.      cur ← cur.next
 6.  if cur = NIL
 7.      return FALSE           // key tidak ada di list
 8.  if prev = NIL              // cur adalah simpul pertama
 9.      L.head ← cur.next
10.  else
11.      prev.next ← cur.next   // lompati cur
12.  free cur
13.  return TRUE
```

Gambar 7.10 menghapus 35 dari 20, 30, 35, 40. Ketika gelung berhenti (panel a), `cur` berada di 35 dan `prev` di 30. Baris 11 membuat 30 menunjuk 40, sehingga 35 *dilompati*: penelusuran dari `head` tidak lagi menjumpainya (panel b). Baris 12 lalu melepaskannya (panel c).



Gambar 7.10 `deleteKey(35)`: (a) `prev` dan `cur` berhenti dengan `cur` di 35, (b) `prev->next` melompatinya, (c) simpul itu dihapus.

Fungsi itu mencakup empat keadaan, dan implementasi yang benar harus menangani setiap satu di antaranya; program di Subbab 7.6.9 menguji keempatnya.

1. **Kunci berada di simpul pertama.** Gelung tidak berjalan, `prev` tetap NIL, dan baris 9 memindahkan `head`. Ini adalah penghapusan depan.
2. **Kunci berada di simpul tengah.** Baris 11 melompatinya.
3. **Kunci berada di simpul terakhir.** Baris 11 mengisi `prev.next` dengan `cur.next`, yaitu NIL, sehingga `prev` menjadi simpul terakhir yang baru. Ini adalah penghapusan belakang, tanpa kasus tersendiri.
4. **Kunci tidak ada.** `cur` berlari melewati ujung list dan fungsi melaporkan kegagalan tanpa mengubah apa pun.

Keadaan kelima, list kosong, adalah kasus 4 dengan gelung yang tidak pernah dimulai.

Ongkos. $\Theta(n)$ pada kasus terburuk, untuk perjalanannya; pelepasan tautannya sendiri $O(1)$.

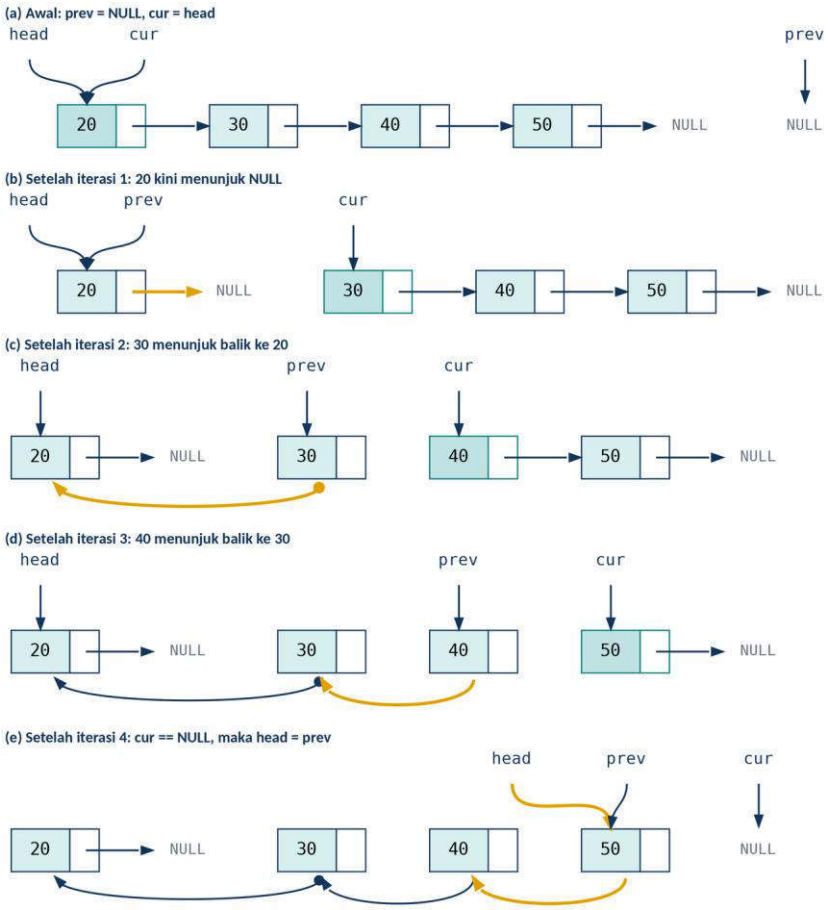
7.6.6 Membalik list secara iteratif

Untuk membalik list kita tidak memindahkan data apa pun; kita membalik setiap panah. Simpul pertama harus berakhir menunjuk NULL, setiap simpul lain menunjuk simpul yang semula mendahuluinya, dan `head` menunjuk simpul yang semula terakhir. Algoritma iteratifnya, **Pseudokode 7.10**, berjalan sekali sepanjang list dengan tiga pointer:

Pseudokode 7.10 `REVERSE-ITERATIVE`: balik list dalam satu lintasan dengan membalik setiap panah `next`.

```
REVERSE-ITERATIVE(L):
 1.  prev ← NIL
 2.  cur ← L.head
 3.  while cur ≠ NIL
 4.      nxt ← cur.next        // 1. ingat sisa list
 5.      cur.next ← prev       // 2. balikkan panah simpul ini
 6.      prev ← cur            // 3. majukan kedua pointer
 7.      cur ← nxt
 8.  L.head ← prev
```

Gambar 7.11 menelusuri keempat iterasi pada 20, 30, 40, 50. Baris 4-lah yang membuat algoritma ini bekerja: baris 5 menimpa `cur.next`, satu-satunya catatan tentang letak sisa list, sehingga catatan itu harus disimpan di `nxt` lebih dulu.



Gambar 7.11 Pembalikan iteratif atas 20, 30, 40, 50: sesudah setiap iterasi satu panah lagi menunjuk ke belakang; ketika `cur` mencapai NULL, `prev` adalah simpul pertama yang baru.

Mengapa algoritma itu benar? Sebuah *invarian gelung* (*loop invariant*), yaitu pernyataan yang benar setiap kali uji gelung hendak dievaluasi, membuat argumennya tepat.

> **INVARIAN GELUNG.** Pada awal setiap iterasi gelung baris 3 sampai 7, `prev` adalah *head* dari list yang sudah terbalik dengan benar, berisi simpul-simpul yang telah dikunjungi, dan `cur` adalah *head* dari list yang belum disentuh, berisi simpul-simpul yang belum dikunjungi; kedua list itu bersama-sama memuat setiap simpul tepat satu kali.

- **Inisialisasi.** Sebelum iterasi pertama belum ada simpul yang dikunjungi: `prev` bernilai NIL, list terbalik yang kosong, dan `cur` adalah `L.head`, seluruh list yang belum disentuh. Invariannya berlaku.
- **Pemeliharaan.** Satu iterasi melepas simpul pertama list yang belum disentuh (sesudah menyimpan sisanya di `nxt`), membuatnya menunjuk list terbalik, dan menjadikannya *head* baru list terbalik (`prev ← cur`); list yang belum disentuh kini dimulai di `nxt`. Kedua list tetap benar dan setiap simpul tetap berada di tepat satu di antaranya.
- **Terminasi.** Gelung berhenti ketika `cur` bernilai NIL, yaitu ketika list yang belum disentuh kosong. Menurut invariannya, `prev` saat itu adalah *head* seluruh list dalam keadaan terbalik, dan baris 8 menjadikannya *head* yang baru.

Ongkos. Setiap iterasi $O(1)$ dan ada n iterasi, sehingga pembalikan berongkos waktu $\Theta(n)$ dan memori tambahan $O(1)$: tiga pointer, berapa pun panjangnya.

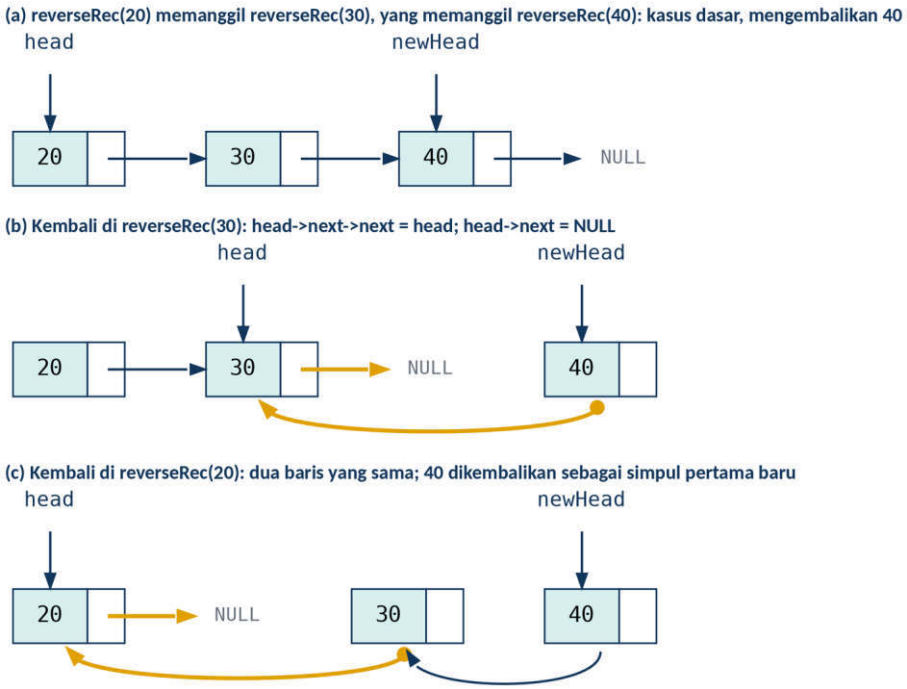
7.6.7 Membalik list secara rekursif

Definisi rekursif di Subbab 7.6.2 menyarankan algoritma kedua. Untuk membalik list yang dimulai pada `head`: bila list itu berisi nol atau satu simpul, ia sudah terbalik; selainnya, balikkan sisa list (semua yang sesudah `head`), lalu tempelkan `head` pada ujung sisa yang sudah terbalik itu. **Pseudokode 7.11** menuliskannya.

Pseudokode 7.11 `REVERSE-RECURSIVE`: balikkan sisa list, lalu gantungkan simpul pertama pada ujungnya.

```
REVERSE-RECURSIVE(head):      // mengembalikan head dari list yang terbalik
 1.  if head = NIL or head.next = NIL
 2.      return head
 3.  newHead ← REVERSE-RECURSIVE(head.next)
 4.  head.next.next ← head    // simpul kedua lama kini menunjuk balik ke head
 5.  head.next ← NIL          // head menjadi simpul terakhir
 6.  return newHead
```

Baris 4 adalah baris yang cerdik. Sesudah baris 3 membalik sisanya, `head.next` masih menunjuk simpul yang *semula* kedua, yang kini menjadi simpul *terakhir* dari sisa yang terbalik. Maka `head.next.next ← head` membuat simpul itu menunjuk balik ke `head`, dan baris 5 mengakhiri list pada `head`. Tidak diperlukan perjalanan ke ujung sisa yang terbalik. **Gambar 7.12** memperlihatkan ketiga pemanggilan pada 20, 30, 40 ketika semuanya kembali.



Gambar 7.12 Pembalikan rekursif atas 20, 30, 40: pemanggilan terdalam mengembalikan 40; setiap pemanggilan yang kembali membalik satu panah dan memutus tautan maju simpulnya sendiri.

Setiap pemanggilan mengerjakan kerja konstan selain pemanggilan rekursifnya, sehingga waktu jalannya $R(n)$ pada list berisi n simpul memenuhi

$$R(n) = R(n-1) + c, \quad R(1) = c \quad \Rightarrow \quad R(n) = c\,n = \Theta(n) \tag{7.7}$$

dengan c ongkos konstan baris 1, 2, 4, 5, dan 6 (uraikan rekurensinya: $R(n) = R(n-2) + 2c = \cdots = R(1) + (n-1)c = nc$). Waktunya sama dengan versi iteratif, tetapi, seperti `lengthRec`, kedalaman rekursinya n, sehingga memori tambahannya $\Theta(n)$ dan list yang sangat panjang dapat meluapkan *call stack*. Versi iteratif adalah yang dipakai dalam produksi; versi rekursif adalah

yang perlu dipahami, karena pola yang sama, “selesaikan sisanya, lalu betulkan simpul pertama”, muncul kembali di Bab 10 dan dalam algoritma pohon di Bab 11.

7.6.8 `destroy`

`destroy` melepaskan setiap simpul, persis seperti `clear` di Subbab 7.5.7, dan pada **Kode 7.6** ia juga mengembalikan banyaknya simpul yang dilepaskannya, supaya keluaran dapat memastikan hitungannya. Apa pun namanya, sebuah list harus dimusnahkan tepat satu kali, dan sesudah dimusnahkan `head`-nya harus NULL supaya tidak ada yang tanpa sengaja memakai simpul-simpul yang sudah dibebaskan.

7.6.9 Program lengkapnya

Kode 7.6 melatih setiap fungsi subbab ini. Program itu membangun list terurut dari 40, 10, 30, 50, 20 dengan `insertSorted`, mencari kunci yang ada dan yang tidak ada, menyisipkan 35 sesudah 30, mengukur panjang dengan kedua cara, menghapus kunci di simpul pertama, di tengah, di simpul terakhir, dan kunci yang tidak ada, membalik list dua kali (sekali dengan masing-masing cara), lalu memusnahkannya.

Kode 7.6 `sll_ops.cpp`: pencarian, panjang, penyisipan sesudah simpul dan secara terurut, penghapusan berdasarkan kunci, pembalikan iteratif dan rekursif, serta pemusnahan.

```cpp
// sll_ops.cpp -- search, length, insert after, sorted insert, delete by key,
// reverse (iterative and recursive) and destroy on a head-pointer list.
#include <cstdio>

struct Node {
    int data;
    Node *next;
};

void show(const char *label, const Node *head) {
    std::printf("%-22s", label);
    for (const Node *aux = head; aux != nullptr; aux = aux->next) {
        std::printf("%d -> ", aux->data);
    }
    std::printf("NULL\n");
}

// Returns the first node holding key, or nullptr. Also reports its index.
Node *search(Node *head, int key, int &index) {
    index = 0;
    for (Node *aux = head; aux != nullptr; aux = aux->next, ++index) {
        if (aux->data == key) return aux;
    }
    index = -1;
    return nullptr;
}

int lengthIter(const Node *head) {
    int n = 0;
    for (const Node *aux = head; aux != nullptr; aux = aux->next) ++n;
    return n;
}

int lengthRec(const Node *head) {
    if (head == nullptr) return 0;         // the empty list has length 0
    return 1 + lengthRec(head->next);      // one node plus the rest
}

// Inserts a new node right after node p (p must not be nullptr).
void insertAfter(Node *p, int newData) {
    Node *newNode = new Node{newData, p->next};   // 1. new node points at p's successor
    p->next = newNode;                            // 2. p points at the new node
}
```

Kode 7.6 (Lanjutan) *sll_ops.cpp: pencarian, panjang, penyisipan sesudah simpul dan secara terurut, penghapusan berdasarkan kunci, pembalikan iteratif dan rekursif, serta pemusnahan.*

```
// Keeps the list in ascending order.
void insertSorted(Node *&head, int newData) {
    if (head == nullptr || newData < head->data) {    // new first node
        head = new Node{newData, head};
        return;
    }
    Node *aux = head;
    while (aux->next != nullptr && aux->next->data < newData) {
        aux = aux->next;                   // aux: last node smaller than newData
    }
    insertAfter(aux, newData);
}

// Deletes the first node holding key. Returns true if one was found.
bool deleteKey(Node *&head, int key) {
    Node *prev = nullptr;
    Node *cur = head;
    while (cur != nullptr && cur->data != key) {
        prev = cur;
        cur = cur->next;
    }
    if (cur == nullptr) return false;      // key is not in the list
    if (prev == nullptr) head = cur->next; // deleting the first node
    else prev->next = cur->next;           // bypass cur
    delete cur;
    return true;
}

void reverseIter(Node *&head) {
    Node *prev = nullptr;
    Node *cur = head;
    while (cur != nullptr) {
        Node *nxt = cur->next;             // 1. remember the rest of the list
        cur->next = prev;                  // 2. turn this arrow around
        prev = cur;                        // 3. step forward
        cur = nxt;
    }
    head = prev;
}
```

Kode 7.6 (Lanjutan) *sll_ops.cpp: pencarian, panjang, penyisipan sesudah simpul dan secara terurut, penghapusan berdasarkan kunci, pembalikan iteratif dan rekursif, serta pemusnahan.*

```
// Reverses the list that starts at head and returns its new first node.
Node *reverseRec(Node *head) {
    if (head == nullptr || head->next == nullptr) return head;
    Node *newHead = reverseRec(head->next);    // reverse everything after head
    head->next->next = head;                   // the old second node now points back
    head->next = nullptr;                      // head is the new last node
    return newHead;
}

int destroy(Node *&head) {
    int released = 0;
    while (head != nullptr) {
        Node *del = head;
        head = head->next;
        delete del;
        ++released;
    }
    return released;
}

int main() {
    Node *head = nullptr;
    int keys[] = {40, 10, 30, 50, 20};
    for (int k : keys) {
        insertSorted(head, k);
        char label[32];
        std::snprintf(label, sizeof label, "insertSorted(%d):", k);
        show(label, head);
    }

    int index;
    Node *p = search(head, 30, index);
    std::printf("search(30): found at index %d\n", index);
    search(head, 99, index);
    std::printf("search(99): index %d (not found)\n", index);

    insertAfter(p, 35);
    show("insertAfter(30, 35):", head);
    std::printf("lengthIter = %d, lengthRec = %d\n", lengthIter(head), lengthRec(head));
```

Kode 7.6 (Lanjutan) *sll_ops.cpp: pencarian, panjang, penyisipan sesudah simpul dan secara terurut, penghapusan berdasarkan kunci, pembalikan iteratif dan rekursif, serta pemusnahan.*

```
    int toDelete[] = {10, 35, 50, 99};
    for (int k : toDelete) {
        bool ok = deleteKey(head, k);
        char label[32];
        std::snprintf(label, sizeof label, "deleteKey(%d) %s:", k, ok ? "ok" : "no");
        show(label, head);
    }

    reverseIter(head);
    show("reverseIter:", head);
    head = reverseRec(head);
    show("reverseRec:", head);

    std::printf("destroy released %d nodes\n", destroy(head));
    show("after destroy:", head);
    return 0;
}
```

```
insertSorted(40):     40 -> NULL
insertSorted(10):     10 -> 40 -> NULL
insertSorted(30):     10 -> 30 -> 40 -> NULL
insertSorted(50):     10 -> 30 -> 40 -> 50 -> NULL
insertSorted(20):     10 -> 20 -> 30 -> 40 -> 50 -> NULL
search(30): found at index 2
search(99): index -1 (not found)
insertAfter(30, 35):  10 -> 20 -> 30 -> 35 -> 40 -> 50 -> NULL
lengthIter = 6, lengthRec = 6
deleteKey(10) ok:     20 -> 30 -> 35 -> 40 -> 50 -> NULL
deleteKey(35) ok:     20 -> 30 -> 40 -> 50 -> NULL
deleteKey(50) ok:     20 -> 30 -> 40 -> NULL
deleteKey(99) no:     20 -> 30 -> 40 -> NULL
reverseIter:          40 -> 30 -> 20 -> NULL
reverseRec:           20 -> 30 -> 40 -> NULL
destroy released 3 nodes
after destroy:        NULL
```

Setiap baris keluaran dapat dicocokkan dengan gambar-gambarnya. Kelima penyisipan terurut menghasilkan 10, 20, 30, 40, 50, kasus pada **Gambar 7.9**. 30 berada pada indeks 2. Sesudah `insertAfter`, 35 duduk di antara 30 dan 40 seperti pada **Gambar 7.8**. Keempat penghapusan melatih keempat kasus Subbab 7.6.5 secara berurutan: simpul pertama (10), simpul tengah (35), simpul terakhir (50), dan kunci yang tidak ada (99, "no"). Membalik 20, 30, 40 menghasilkan 40, 30, 20, dan membalik sekali lagi memulihkan 20, 30, 40. Akhirnya tiga simpul dilepaskan dan list kosong. Versi Java dan Python-nya, `SllOps.java` dan `sll_ops.py` di folder kode bab ini, mencetak baris-baris yang sama.

Tabel 7.3 meringkas ongkos-ongkos subbab ini.

Tabel 7.3 Ongkos operasi-operasi Subbab 7.6 pada list berisi n simpul.

Operasi	Waktu	Memori tambahan
`search`	$\Theta(n)$ terburuk, $O(1)$ terbaik	$O(1)$
`lengthIter` / `lengthRec`	$\Theta(n)$ / $\Theta(n)$	$O(1)$ / $\Theta(n)$ *call stack*
`insertAfter(p, x)`	$O(1)$	$O(1)$
`insertSorted`	$\Theta(n)$ terburuk	$O(1)$
`deleteKey`	$\Theta(n)$ terburuk	$O(1)$
`reverseIter` / `reverseRec`	$\Theta(n)$ / $\Theta(n)$	$O(1)$ / $\Theta(n)$ *call stack*
`destroy`	$\Theta(n)$	$O(1)$

7.7 Simpul Kepala Dummy: Lebih Sedikit Kasus Khusus

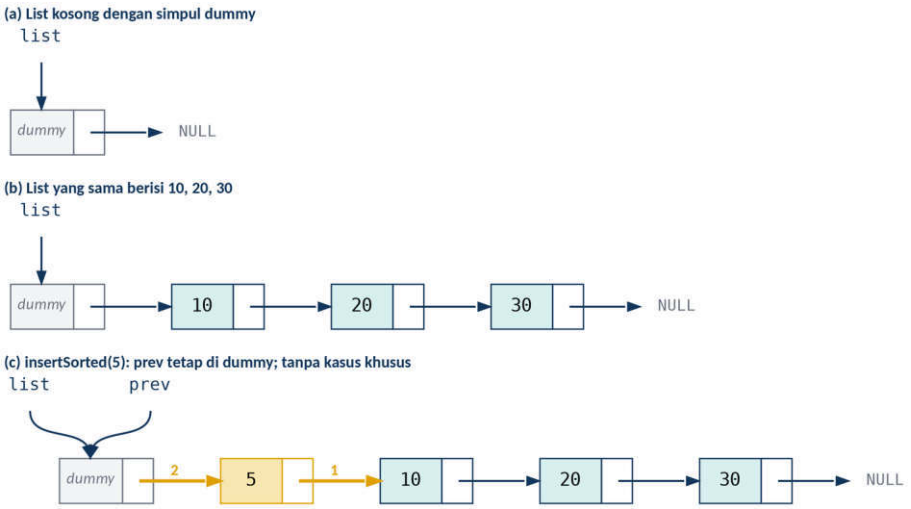
7.7.1 Dari mana kasus-kasus khusus berasal

Tengoklah kembali prosedur-prosedur di Subbab 7.5 dan 7.6 dan hitunglah kasus-kasus khususnya: `backInsertion` memiliki cabang untuk list kosong, `insertSorted` cabang untuk "menjadi simpul pertama yang baru", `deleteKey` cabang untuk "menghapus simpul pertama", `backDeletion` cabang untuk "tepat satu simpul". Semuanya berakar pada sebab yang sama. Setiap simpul kecuali yang pertama ditunjuk oleh medan `next` simpul sebelumnya, tetapi simpul pertama ditunjuk oleh `head`, yang merupakan benda dari jenis lain: sebuah variabel, bukan medan di dalam simpul. Maka setiap kali sebuah operasi dapat mengubah simpul *mana* yang pertama, ia memerlukan dua versi dari penugasan yang sama, satu untuk `head` dan satu untuk `prev->next`.

Obat klasiknya adalah membuat simpul nyata pertama serupa dengan setiap simpul lain dengan menaruh sebuah simpul di depannya. Simpul tambahan ini disebut *simpul dummy*, *simpul header*, atau *sentinel* (penjaga). Ia dibuat ketika list dibuat, tidak pernah dihapus selama list masih ada, dan medan datanya diabaikan.

> **DEFINISI.** *Single linked list dengan simpul kepala dummy* selalu diawali satu simpul tambahan yang datanya tidak bermakna. Pointer list menunjuk simpul *dummy* itu, elemen nyata pertama adalah `dummy->next`, dan list kosong jika dan hanya jika `dummy->next` bernilai NULL. List yang dicapai lewat pointer biasa ke simpul nyata pertamanya, seperti di Subbab 7.5 dan 7.6, adalah list *tanpa* simpul *dummy*.

Gambar 7.13 memperlihatkan list kosong, yang kini bukan lagi pointer nol melainkan satu simpul *dummy* (panel a), list yang sama berisi tiga nilai (panel b), dan penyisipan 5 di depan elemen-elemen nyata (panel c). Pada panel (c) simpul sebelum titik penyisipan adalah simpul *dummy*, sehingga penyisipannya adalah "sisipkan sesudah `prev`" yang biasa: tidak diperlukan penugasan ke pointer list.



Gambar 7.13 List dengan simpul kepala *dummy*: (a) list kosong berupa satu simpul *dummy*, (b) list berisi 10, 20, 30, (c) menyisipkan 5 sebelum 10 adalah penyisipan sesudah *dummy*, tanpa kasus khusus.

7.7.2 Operasi tanpa kasus khusus

Dengan adanya *dummy*, setiap simpul yang menyimpan data memiliki pendahulu, sehingga pointer `prev` selalu dapat mulai dari *dummy*. **Kode 7.7** menulis ulang operasi-operasinya. Bandingkan `insertSorted` dan `deleteKey` dengan padanannya di **Kode 7.6**: masing-masing kini tinggal satu gelung dan satu perubahan tautan, dan pointer list tidak pernah ditugasi lagi sesudah `init`, sehingga fungsi-fungsinya menerimanya secara nilai.

Kode 7.7 `sll_dummy.cpp`: *single linked list* dengan simpul kepala *dummy*; penyisipan dan penghapusan tidak memerlukan kasus khusus untuk elemen pertama.

```cpp
// sll_dummy.cpp -- a singly linked list with a dummy (sentinel) head node.
// The dummy node holds no real data; the first real node is dummy->next.
#include <cstdio>

struct Node {
    int data;
    Node *next;
};

Node *init() {                          // an empty list is ONE node: the dummy
    return new Node{0, nullptr};
}

bool isEmpty(const Node *dummy) {
    return dummy->next == nullptr;
}

void show(const char *label, const Node *dummy) {
    std::printf("%-20s[dummy] -> ", label);
    for (const Node *aux = dummy->next; aux != nullptr; aux = aux->next) {
        std::printf("%d -> ", aux->data);
    }
    std::printf("NULL\n");
}

void insertFront(Node *dummy, int x) {       // no special case for an empty list
    dummy->next = new Node{x, dummy->next};
}

void insertBack(Node *dummy, int x) {
    Node *aux = dummy;                       // start at the dummy, never nullptr
    while (aux->next != nullptr) aux = aux->next;
    aux->next = new Node{x, nullptr};
}

void insertSorted(Node *dummy, int x) {      // one loop, no "new first node" branch
    Node *prev = dummy;
    while (prev->next != nullptr && prev->next->data < x) prev = prev->next;
    prev->next = new Node{x, prev->next};
}
```

Kode 7.7 (Lanjutan) *sll_dummy.cpp: single linked list dengan simpul kepala dummy; penyisipan dan penghapusan tidak memerlukan kasus khusus untuk elemen pertama.*

```
bool deleteKey(Node *dummy, int key) {       // prev always exists: it may be the dummy
    Node *prev = dummy;
    while (prev->next != nullptr && prev->next->data != key) prev = prev->next;
    if (prev->next == nullptr) return false;
    Node *del = prev->next;
    prev->next = del->next;
    delete del;
    return true;
}

bool deleteFront(Node *dummy) {
    if (isEmpty(dummy)) return false;
    Node *del = dummy->next;
    dummy->next = del->next;
    delete del;
    return true;
}

void destroy(Node *&dummy) {                 // frees the real nodes AND the dummy
    while (dummy != nullptr) {
        Node *del = dummy;
        dummy = dummy->next;
        delete del;
    }
}

int main() {
    Node *list = init();
    show("init:", list);
    std::printf("isEmpty = %s\n", isEmpty(list) ? "true" : "false");

    insertSorted(list, 30);
    insertSorted(list, 10);
    insertSorted(list, 20);
    show("insertSorted x3:", list);
    insertFront(list, 5);
    insertBack(list, 99);
    show("front 5, back 99:", list);
```

Kode 7.7 (Lanjutan) *sll_dummy.cpp: single linked list dengan simpul kepala dummy; penyisipan dan penghapusan tidak memerlukan kasus khusus untuk elemen pertama.*

```
    deleteKey(list, 5);                      // the first real node: same code path
    deleteKey(list, 20);                     // a middle node
    deleteKey(list, 99);                     // the last node
    show("delete 5, 20, 99:", list);
    deleteFront(list);
    deleteFront(list);
    show("deleteFront x2:", list);
    std::printf("isEmpty = %s\n", isEmpty(list) ? "true" : "false");

    destroy(list);
    std::printf("destroyed, list == nullptr is %s\n", list == nullptr ? "true" : "false");
    return 0;
}
```

```
init:               [dummy] -> NULL
isEmpty = true
insertSorted x3:    [dummy] -> 10 -> 20 -> 30 -> NULL
front 5, back 99:   [dummy] -> 5 -> 10 -> 20 -> 30 -> 99 -> NULL
delete 5, 20, 99:   [dummy] -> 10 -> 30 -> NULL
deleteFront x2:     [dummy] -> NULL
isEmpty = true
destroyed, list == nullptr is true
```

Keluarannya menampilkan *dummy* secara tersurat sebagai `[dummy]`. Menghapus 5, elemen nyata pertama, menjalankan kode yang persis sama dengan menghapus 20 di tengah atau 99 di ujung; demikian pula menyisipkan 5 di depan dengan `insertFront`, yang tidak lain adalah `insertAfter(dummy, x)`. Dua hal berubah di tempat lain. `isEmpty` kini menguji `dummy->next`, dan `destroy` ikut melepaskan *dummy*, itulah sebabnya, sesudah list dimusnahkan, program melaporkan bahwa `list` bernilai `nullptr`. Versi Java dan Python-nya, `SllDummy.java` dan `sll_dummy.py`, mencetak keluaran yang sama.

Harga *dummy* adalah satu simpul memori per list, dan satu aturan yang harus dihormati setiap fungsi: *dummy* bukan data, sehingga penelusuran dimulai dari `dummy->next`, dan `size` tidak boleh menghitungnya. Untuk program dengan banyak list pendek (tabel *hash* di Bab 15 dapat menampung ribuan list), simpul tambahan per list itu dapat berarti; untuk satu list panjang, tidak.

7.7.3 Cara ketiga: pointer ke tautan

C dan C++ menawarkan teknik ketiga yang menghapus kasus-kasus khusus *tanpa* simpul *dummy*. Pengamatannya adalah bahwa yang diubah oleh penyisipan atau penghapusan selalu sebuah *tautan*, variabel bertipe `Node *`, yang berupa `head` itu sendiri atau medan `next` suatu simpul. Maka alih-alih menjalankan pointer ke *simpul* sebelum posisinya, kita menjalankan pointer ke *tautan* yang menunjuk posisi itu: sebuah `Node **link`, yang mula-mula `&head` lalu menjadi `&(node->next)` untuk simpul-simpul berikutnya. Menulis `*link = ...` lalu mengubah `head` atau sebuah medan `next`, mana pun yang sedang dialamati `link`, dengan baris kode yang sama. **Kode 7.8** menerapkan gagasan itu pada penyisipan terurut, penghapusan berdasarkan kunci, dan penghapusan setiap kemunculan sebuah kunci.

Kode 7.8 `sll_link.cpp`: menjalankan pointer ke tautan (`Node **`) menghapus kasus khusus untuk simpul pertama tanpa simpul *dummy*.

```
// sll_link.cpp -- no dummy node and no special case: walk a pointer to the LINK.
// link is the address of "the pointer that points at the current node":
// first &head, later &(some node)->next.
#include <cstdio>

struct Node {
    int data;
    Node *next;
};

void show(const char *label, const Node *head) {
    std::printf("%-18s", label);
    for (; head != nullptr; head = head->next) std::printf("%d -> ", head->data);
    std::printf("NULL\n");
}

void insertSorted(Node **link, int x) {
    while (*link != nullptr && (*link)->data < x) link = &(*link)->next;
    *link = new Node{x, *link};          // rewrites head OR some node's next
}

bool deleteKey(Node **link, int key) {
    while (*link != nullptr && (*link)->data != key) link = &(*link)->next;
    if (*link == nullptr) return false;
    Node *del = *link;
    *link = del->next;                   // the link now skips the deleted node
    delete del;
    return true;
}

int removeAll(Node **link, int key) {    // delete every node holding key
    int removed = 0;
    while (*link != nullptr) {
        if ((*link)->data == key) {
            Node *del = *link;
            *link = del->next;           // do NOT advance: *link is a new node now
            delete del;
            ++removed;
        } else {
            link = &(*link)->next;
        }
    }
    return removed;
}
```

Kode 7.8 (Lanjutan) *sll_link.cpp: menjalankan pointer ke tautan (Node **) menghapus kasus khusus untuk simpul pertama tanpa simpul dummy.*

```
int main() {
    Node *head = nullptr;
    int keys[] = {30, 10, 20, 20, 5, 20};
    for (int k : keys) insertSorted(&head, k);
    show("sorted:", head);

    deleteKey(&head, 5);                 // the first node: head itself changes
    show("deleteKey(5):", head);
    int n = removeAll(&head, 20);
    std::printf("removeAll(20) removed %d nodes\n", n);
    show("now:", head);

    while (head != nullptr) deleteKey(&head, head->data);   // empty the list
    show("emptied:", head);
    return 0;
}
```

```
sorted:           5 -> 10 -> 20 -> 20 -> 20 -> 30 -> NULL
deleteKey(5):     10 -> 20 -> 20 -> 20 -> 30 -> NULL
removeAll(20) removed 3 nodes
now:              10 -> 30 -> NULL
emptied:          NULL
```

Pada `insertSorted(&head, x)`, bila nilai barunya paling kecil, gelung tidak bergerak dan `*link = new Node{x, *link}` menugasi `head`; selainnya `link` menyimpan alamat medan `next` dari simpul terakhir yang lebih kecil dan baris yang sama menugasi medan itu. `removeAll` memperlihatkan teknik ini pada puncaknya: ketika menghapus sebuah simpul, ia *tidak* memajukan `link`, karena `*link` kini berisi simpul sesudah yang dihapus, yang juga harus diperiksa; ketiga 20 yang berurutan terhapus dengan benar. Teknik ini sering dipuji sebagai contoh "selera yang baik" (*good taste*) dalam kode tingkat rendah, karena ia mengganti sebuah `if` tentang simpul pertama dengan pilihan yang lebih baik tentang apa yang ditunjuk [7]. Teknik ini tidak memiliki padanan di Java maupun Python, yang tidak dapat mengambil alamat sebuah medan; di sana simpul *dummy*-lah alatnya.

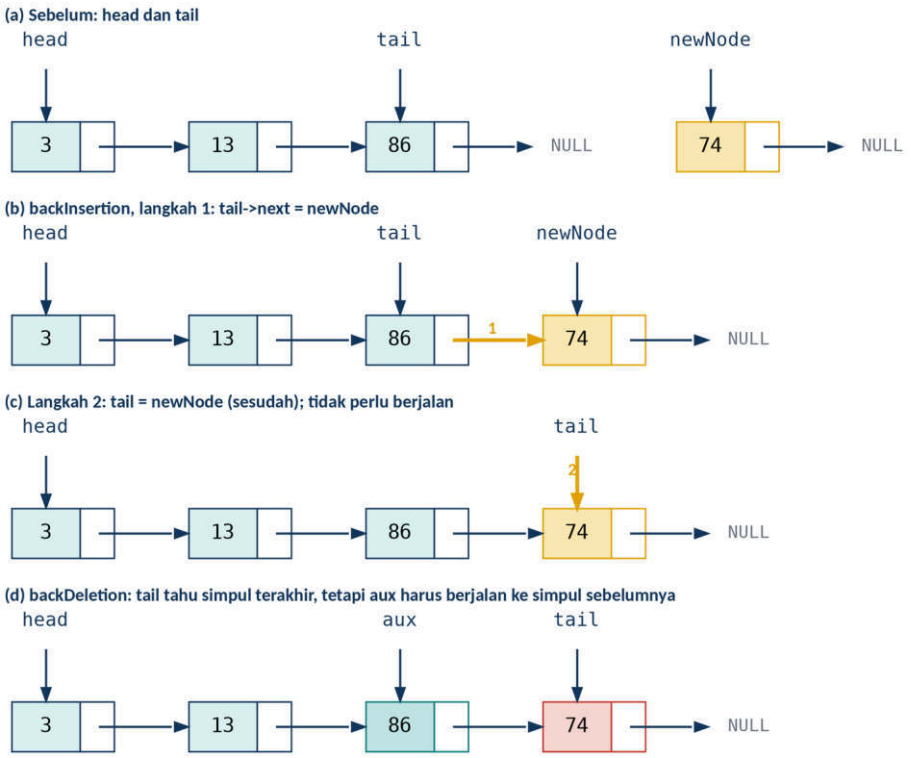
7.7.4 Varian mana yang dipakai

Ketiga varian memiliki ongkos asimtotik yang sama. Pilihannya menyangkut kejelasan dan memori: pointer *head* biasa paling sederhana untuk dijelaskan dan tidak berongkos tambahan; simpul *dummy* memberikan kode terpendek dan paling kecil peluang salahnya dalam setiap bahasa, dengan harga satu simpul tambahan; pointer ke tautan memberikan kode sependek itu tanpa simpul tambahan, tetapi hanya di C dan C++, dan menuntut pembaca nyaman dengan `Node **`. *List* pustaka di Subbab 7.12 dibangun dengan gagasan *sentinel* semacam ini di dalamnya: `before_begin()` milik `std::forward_list` mengembalikan iterator ke posisi *sebelum* elemen pertama justru supaya `insert_after` dan `erase_after` dapat bekerja di depan tanpa kasus khusus.

7.8 Pointer Head dan Tail

7.8.1 Gagasannya

Penyisipan belakang pada list ber-*head* berongkos $\Theta(n)$ karena list tidak tahu di mana simpul terakhirnya. Obat yang jelas adalah mengingatnya: simpan pointer kedua, `tail`, yang selalu menunjuk simpul terakhir. List kini memiliki dua pointer eksternal, dan setiap operasi harus menjaga keduanya tetap benar. Hasilnya diperlihatkan pada panel (a) sampai (c) **Gambar 7.14**: penyisipan belakang menempelkan simpul baru pada `tail` lalu memindahkan `tail`, tanpa perjalanan sama sekali.



Gambar 7.14 List dengan `head` dan `tail`: (a) sebelum `backInsertion(74)`, (b) simpul terakhir lama menunjuk simpul baru, (c) `tail` pindah ke sana; (d) `backDeletion` tetap harus menjalankan `aux` ke simpul sebelum `tail`.

Uji kosong kini dapat melihat pointer mana pun; rumusan klasiknya menguji `tail == nullptr`. Invarian yang harus dipertahankan setiap fungsi adalah:

> **INVARIAN HEAD-TAIL.** List kosong dan `head` serta `tail` keduanya NULL, atau `head` menunjuk simpul pertama, `tail` menunjuk simpul terakhir, dan `tail->next` bernilai NULL. Pada list yang berisi tepat satu simpul, `head` dan `tail` menunjuk simpul yang sama.

Invarian itu menunjukkan di mana kasus-kasus khusus yang baru berada: setiap operasi yang dapat memindahkan list antara keadaan "kosong" dan "satu simpul" harus mengubah *kedua* pointer. Penyisipan depan ke list kosong harus mengisi `tail` di samping `head`; penghapusan depan atas satu-satunya simpul harus mengembalikan `tail` ke NULL di samping `head`. Melupakan yang kedua membuat `tail` menunjuk simpul yang sudah dibebaskan, dan penyisipan belakang berikutnya menulis ke memori yang sudah dibebaskan.

7.8.2 Mengapa `tail` tidak dapat mempercepat penghapusan belakang

Godaannya adalah menyimpulkan bahwa `tail` juga membuat penghapusan belakang menjadi $O(1)$. Tidak demikian, dan alasannya layak dipahami dengan tepat. `tail` langsung memberi tahu kita simpul *mana* yang dihapus. Tetapi menghapusnya berarti simpul *sebelum*-nya harus menjadi simpul terakhir yang baru: `next`-nya harus menjadi NULL dan `tail` harus pindah ke sana. Simpul bertaut tunggal mencatat siapa yang datang sesudahnya, tidak pernah siapa yang datang sebelumnya, dan tidak ada bagian lain dari list yang mencatatnya. Satu-satunya cara menemukan simpul yang `next`-nya sama dengan `tail` adalah berjalan dari `head` sampai kita menjumpainya, seperti diperlihatkan panel (d) **Gambar 7.14**. Maka penghapusan belakang tetap $\Theta(n)$ dengan pointer *tail*: $n - 2$ langkah pada list berisi $n \geq 2$ simpul.

Dapatkah pointer ketiga, ke simpul kedua dari belakang, memperbaikinya? Hanya untuk satu penghapusan: sesudah penghapusan itu kita akan memerlukan simpul *ketiga* dari belakang, dan

seterusnya. Perbaikan sejatinya adalah memberi setiap simpul pointer ke pendahulunya, yaitu *double linked list* di Bab 9.

7.8.3 Programnya

Kode 7.9 mengikuti penyajian klasik dengan pointer global `head` dan `tail`. `backDeletion` menghitung langkahnya supaya keluaran memperlihatkan ongkos $n - 2$ secara langsung.

Kode 7.9 `sll_head_tail.cpp`: *single linked list* dengan `head` dan `tail`; penyisipan belakang $O(1)$, penghapusan belakang tetap berjalan.

```
// sll_head_tail.cpp -- a singly linked list with two pointers: head and tail.
#include <cstdio>

typedef struct MyNode {
    int data;
    struct MyNode *next;
} TNode;

TNode *head, *tail;

void init() {
    head = nullptr;
    tail = nullptr;
}

bool isEmpty() {
    return tail == nullptr;
}

void frontInsertion(int newData) {
    TNode *newNode = new TNode{newData, head};
    if (isEmpty()) tail = newNode;        // the only node is also the last one
    head = newNode;
    std::printf("%d has been inserted by frontInsertion()\n", newData);
}

void backInsertion(int newData) {         // O(1): no walk, tail knows the end
    TNode *newNode = new TNode{newData, nullptr};
    if (isEmpty()) {
        head = tail = newNode;
    } else {
        tail->next = newNode;             // 1. old last node points at the new one
        tail = newNode;                   // 2. tail moves to the new node
    }
    std::printf("%d has been inserted by backInsertion()\n", newData);
}

void show() {
    if (isEmpty()) {
        std::printf("The linked list is empty\n");
        return;
    }
    for (TNode *aux = head; aux != nullptr; aux = aux->next) std::printf("%d ", aux->data);
    std::printf("\n");
}
```

Kode 7.9 (Lanjutan) *sll_head_tail.cpp: single linked list dengan head dan tail; penyisipan belakang O(1), penghapusan belakang tetap berjalan.*

```
void frontDeletion() {
    if (isEmpty()) {
        std::printf("The linked list is empty\n");
        return;
    }
    TNode *del = head;
    int d = del->data;
    if (head == tail) {                  // one node: the list becomes empty
        head = tail = nullptr;
    } else {
        head = head->next;
    }
    delete del;
    std::printf("%d has been deleted by frontDeletion()\n", d);
}

void backDeletion() {                    // still O(n): who points at tail?
    if (isEmpty()) {
        std::printf("The linked list is empty\n");
        return;
    }
    TNode *del = tail;
    int d = del->data;
    int steps = 0;
    if (head == tail) {
        head = tail = nullptr;
    } else {
        TNode *aux = head;
        while (aux->next != tail) {      // find the node just before tail
            aux = aux->next;
            ++steps;
        }
        aux->next = nullptr;
        tail = aux;
    }
    delete del;
    std::printf("%d has been deleted by backDeletion() after %d step(s)\n", d, steps);
}
```

Kode 7.9 (Lanjutan) *sll_head_tail.cpp: single linked list dengan head dan tail; penyisipan belakang O(1), penghapusan belakang tetap berjalan.*

```
void clear() {
    TNode *aux = head;
    while (aux != nullptr) {
        TNode *del = aux;
        aux = aux->next;
        delete del;
    }
    head = tail = nullptr;
    std::printf("The linked list has been cleared\n");
}

int main() {
    init();
    frontInsertion(13);
    frontInsertion(3);
    backInsertion(86);
    backInsertion(74);
    backInsertion(52);
    backInsertion(61);
    show();

    frontDeletion();
    backDeletion();
    backDeletion();
    backDeletion();
    show();
    std::printf("head->data = %d, tail->data = %d\n", head->data, tail->data);

    clear();
    show();
    return 0;
}
```

```
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()
52 has been inserted by backInsertion()
61 has been inserted by backInsertion()
3 13 86 74 52 61
3 has been deleted by frontDeletion()
61 has been deleted by backDeletion() after 3 step(s)
52 has been deleted by backDeletion() after 2 step(s)
74 has been deleted by backDeletion() after 1 step(s)
13 86
head->data = 13, tail->data = 86
The linked list has been cleared
The linked list is empty
```

Sesudah keenam penyisipan, list berisi 3, 13, 86, 74, 52, 61. Penghapusan depan membuang 3, menyisakan lima simpul, dan ketiga penghapusan belakang memakan 3, 2, dan 1 langkah pada list berisi 5, 4, dan 3 simpul: tepat $n - 2$ setiap kali. Baris data terakhir menegaskan invariannya: `head` berisi 13 dan `tail` berisi 86, simpul pertama dan terakhir dari list sisa 13, 86.

Tabel 7.4 membandingkan varian-varian yang sudah dibangun.

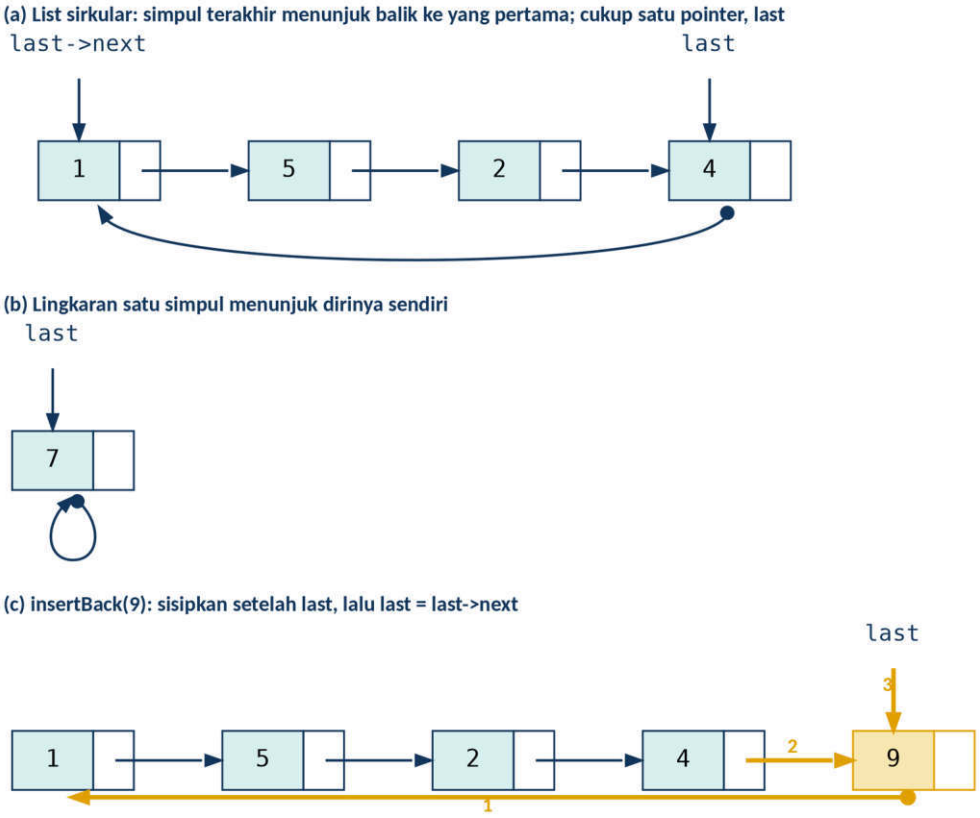
Tabel 7.4 Varian-varian *single linked list* dibandingkan, untuk list berisi n simpul.

Operasi	Head saja	Head dan tail	Simpul kepala dummy
Sisip di depan	$O(1)$	$O(1)$	$O(1)$
Sisip di belakang	$\Theta(n)$	$O(1)$	$\Theta(n)$ (atau $O(1)$ dengan *tail*)
Hapus di depan	$O(1)$	$O(1)$	$O(1)$
Hapus di belakang	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Cari, hapus berdasarkan kunci	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Kasus khusus untuk simpul pertama	ya	ya, dan untuk `tail`	tidak ada
Memori tambahan per list	1 pointer	2 pointer	1 pointer dan 1 simpul

7.9 Single Linked List Melingkar

7.9.1 Menutup rantai

Pada list-list sejauh ini, `next` simpul terakhir bernilai NULL. Bila kita membuatnya menunjuk kembali ke simpul pertama, rantainya menjadi cincin: *single linked list melingkar* (*circular*). Tidak ada yang berubah pada simpulnya; hanya nilai yang disimpan dalam satu medan `next`. Panel (a) **Gambar 7.15** memperlihatkan list melingkar berisi empat simpul.



Gambar 7.15 *Single linked list* melingkar: (a) empat simpul, dicapai lewat satu pointer `last`, dengan simpul pertama di `last->next`; (b) lingkaran satu simpul menunjuk dirinya sendiri; (c) menyisipkan di belakang adalah penyisipan sesudah `last` yang diikuti pemindahan `last`.

Cincin tidak memiliki simpul pertama yang alami, jadi ke mana pointer eksternalnya harus menunjuk? Jawaban terbaik yang mengejutkan adalah: ke simpul *terakhir*. Dari `last`, simpul pertama hanya berjarak satu langkah, `last->next`, sehingga satu pointer memberikan akses $O(1)$ ke *kedua* ujung, tugas yang memerlukan dua pointer di Subbab 7.8. Dengan `last` kita dapat:

- **menyisipkan di depan** dalam $O(1)$: tautkan simpul baru di antara `last` dan `last->next`;
- **menyisipkan di belakang** dalam $O(1)$: sisipkan di depan, lalu majukan `last` satu langkah ke simpul baru, sehingga simpul itu menjadi simpul terakhir (panel c);
- **menghapus di depan** dalam $O(1)$: lepaskan `last->next` dari rantai.

Menghapus di belakang tetap $\Theta(n)$, dengan alasan Subbab 7.8.2: simpul sebelum `last` hanya dapat ditemukan dengan berjalan mengelilingi cincin.

Dua kasus memerlukan kehati-hatian. List melingkar yang kosong adalah `last == nullptr`. Lingkaran berisi *satu* simpul harus menunjuk dirinya sendiri (panel b): `next`-nya adalah alamatnya sendiri. Menghapus satu-satunya simpul karenanya harus mengembalikan `last` ke `nullptr`, yang dikenali dengan uji "simpul yang akan dihapus adalah penerusnya sendiri".

7.9.2 Penelusuran memerlukan tanda berhenti yang baru

Gelung penelusuran **Pseudokode 7.3**, `while (aux != nullptr)`, tidak akan pernah berhenti pada list melingkar, karena tidak ada medan `next` yang bernilai NULL: gelung itu akan mencetak cincinnya selamanya. Penelusuran melingkar harus berhenti ketika kembali ke tempat ia bermula. Karena list kosong tidak memiliki simpul awal, gelung hanya dijalankan bila list tidak kosong, dan ditulis sebagai `do`-`while`, yang menguji *sesudah* badan gelung, sehingga simpul pertama dicetak sebelum pertanyaan "apakah kita sudah kembali ke awal?" diajukan untuk pertama kali:

```
const Node *aux = last->next;          // start at the first node
do {
    visit(aux);
    aux = aux->next;
} while (aux != last->next);           // stop when we come back to the start
```

`while (aux != last->next)` biasa akan langsung gagal sebelum mengunjungi apa pun, karena `aux` *dimulai* pada `last->next`. Menghitung simpul dan berhenti sesudah `size` langkah adalah alternatif yang sama baiknya bila list menyimpan ukurannya.

7.9.3 Layanan bergiliran di meja baca

Cincin adalah struktur yang alami untuk apa pun yang dilayani *bergiliran, berulang-ulang*: setiap peserta mendapat giliran singkat, lalu peserta berikutnya, dan sesudah peserta terakhir peserta pertama mendapat giliran lagi. Sistem operasi membagi prosesor di antara program-program yang berjalan dengan cara persis seperti ini; tekniknya disebut *round-robin scheduling* (penjadwalan bergiliran), dan setiap giliran disebut *time slice* (jatah waktu).

Pustaka Ceria memiliki satu meja baca dengan alat pembaca berkaca pembesar khusus, dan empat pengunjung ingin memakainya: Ani selama 5 menit, Budi 2, Citra 4, dan Dewi, yang datang dengan kartu prioritas sehingga ditaruh di depan, 1 menit. Supaya adil, setiap pengunjung mendapat paling lama 2 menit sekali giliran lalu pindah ke belakang lingkaran; pengunjung yang waktunya habis meninggalkan lingkaran. **Kode 7.10** menyimulasikan sesi itu dengan list melingkar. Pointer `prev` selalu berdiri pada simpul *sebelum* pengunjung yang sedang dilayani, persis yang diperlukan untuk menghapus pengunjung itu dalam $O(1)$ ketika ia selesai, dengan `deleteAfter(prev)`.

Kode 7.10 `csll.cpp`: *single linked list* melingkar yang dicapai lewat `last`, dipakai untuk layanan bergiliran dengan jatah waktu 2 menit.

```cpp
// csll.cpp -- a circular singly linked list kept through ONE pointer, last.
// The first node is always last->next. Used here for a round-robin reading room.
#include <cstdio>
#include <string>

struct Reader {
    std::string name;
    int minutesLeft;          // reading time the reader still needs
};

struct Node {
    Reader data;
    Node *next;
};

bool isEmpty(const Node *last) { return last == nullptr; }

void insertFront(Node *&last, const Reader &r) {
    Node *newNode = new Node{r, nullptr};
    if (isEmpty(last)) {
        newNode->next = newNode;          // a circle of one node points at itself
        last = newNode;
    } else {
        newNode->next = last->next;       // 1. new node points at the old first node
        last->next = newNode;             // 2. last node points at the new first node
    }
}

void insertBack(Node *&last, const Reader &r) {
    insertFront(last, r);                 // put it between last and first ...
    last = last->next;                    // ... and call it the last node: O(1)
}

// Removes the node that follows prev (prev->next) and keeps last correct.
void deleteAfter(Node *&last, Node *prev) {
    Node *del = prev->next;
    if (del == prev) {                    // the only node
        last = nullptr;
    } else {
        prev->next = del->next;
        if (del == last) last = prev;     // deleting the last node moves last back
    }
    delete del;
}
```

Kode 7.10 (Lanjutan) *csll.cpp: single linked list melingkar yang dicapai lewat last, dipakai untuk layanan bergiliran dengan jatah waktu 2 menit.*

```
void show(const char *label, const Node *last) {
    std::printf("%s", label);
    if (isEmpty(last)) {
        std::printf(" (empty)\n");
        return;
    }
    const Node *aux = last->next;         // start at the first node
    do {                                  // stop when we come back to the start
        std::printf(" %s(%d)", aux->data.name.c_str(), aux->data.minutesLeft);
        aux = aux->next;
    } while (aux != last->next);
    std::printf("\n");
}

void destroy(Node *&last) {
    while (!isEmpty(last)) deleteAfter(last, last);   // delete the first node
}

int main() {
    Node *last = nullptr;
    insertBack(last, {"Ani", 5});
    insertBack(last, {"Budi", 2});
    insertBack(last, {"Citra", 4});
    insertFront(last, {"Dewi", 1});
    show("Queue for the one reading desk:", last);

    const int slice = 2;                  // each turn lasts at most 2 minutes
    int clock = 0;
    Node *prev = last;                    // the node BEFORE the one being served
    while (!isEmpty(last)) {
        Node *cur = prev->next;
        int used = cur->data.minutesLeft < slice ? cur->data.minutesLeft : slice;
        cur->data.minutesLeft -= used;
        clock += used;
        std::printf("t=%2d  %-5s read %d min", clock, cur->data.name.c_str(), used);
        if (cur->data.minutesLeft == 0) {
            std::printf(" and is done\n");
            deleteAfter(last, prev);      // prev stays; its next is the following reader
        } else {
            std::printf(", %d min left\n", cur->data.minutesLeft);
            prev = cur;                   // advance around the circle
        }
    }
    show("After the session:", last);
```

Kode 7.10 (Lanjutan) *csll.cpp: single linked list melingkar yang dicapai lewat last, dipakai untuk layanan bergiliran dengan jatah waktu 2 menit.*

```
    insertBack(last, {"Eko", 3});
    insertBack(last, {"Fajar", 3});
    show("Evening readers:", last);
    destroy(last);
    show("After destroy:", last);
    return 0;
}
```

```
Queue for the one reading desk: Dewi(1) Ani(5) Budi(2) Citra(4)
t= 1  Dewi  read 1 min and is done
t= 3  Ani   read 2 min, 3 min left
t= 5  Budi  read 2 min and is done
t= 7  Citra read 2 min, 2 min left
t= 9  Ani   read 2 min, 1 min left
t=11  Citra read 2 min and is done
t=12  Ani   read 1 min and is done
After the session: (empty)
Evening readers: Eko(3) Fajar(3)
After destroy: (empty)
```

Telusurilah putaran pertama. Dewi memerlukan 1 menit, membacanya, lalu pergi ($t = 1$). Ani membaca 2 dari 5 menitnya ($t = 3$). Budi membaca 2 menit penuhnya lalu pergi ($t = 5$). Citra membaca 2 dari 4 menitnya ($t = 7$). Cincin kini hanya berisi Ani (sisa 3 menit) dan Citra (sisa 2), dan layanan berlanjut mengelilinginya tanpa kode khusus untuk "kita sudah sampai di ujung", karena cincin tidak berujung: Ani membaca 2 ($t = 9$), Citra selesai ($t = 11$), dan Ani selesai terakhir ($t = 12$). Totalnya, 12 menit, sama dengan $1 + 5 + 2 + 4$: tidak ada waktu yang terbuang, bagaimanapun cincinnya menyusut. Program lalu memakai kembali cincin kosong itu untuk dua pembaca malam dan memusnahkannya; versi Java dan Python-nya mencetak keluaran yang sama.

Dua penerapan klasik lain dari list melingkar muncul dalam soal latihan: masalah Josephus (Soal Latihan 7.11), ketika orang-orang yang berdiri melingkar keluar satu per satu, dan pendeteksian siklus (Soal Latihan 7.6), yang menjawab pertanyaan "apakah list ini tanpa sengaja menjadi melingkar?" tanpa mengubahnya.

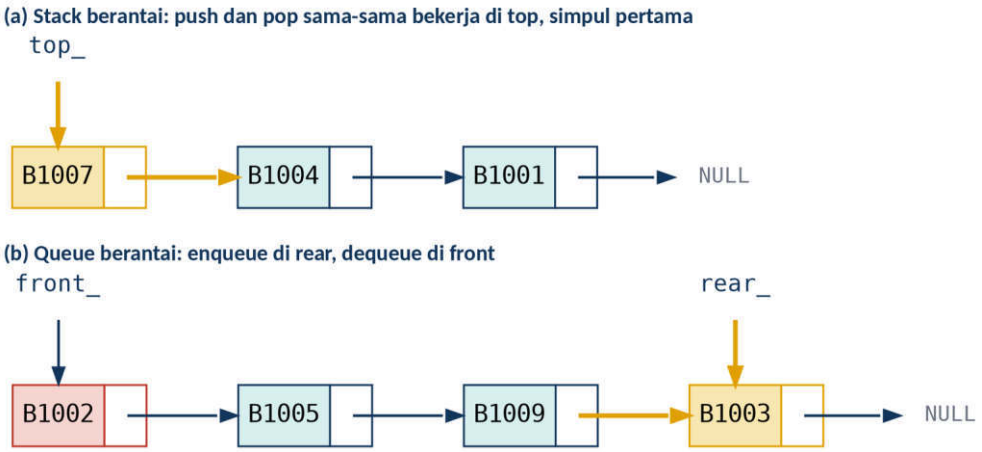
7.10 Stack dan Queue yang Dibangun di Atas Linked List

7.10.1 Kecocokannya

Bab 5 mendefinisikan *stack* (masuk terakhir, keluar pertama: `push`, `pop`, `peek`) dan *queue* (masuk pertama, keluar pertama: `enqueue`, `dequeue`) dan membangun keduanya di atas larik, dengan kapasitas tetap dan, untuk *queue*, aritmetika indeks melingkar yang menghindari pergeseran. *Linked list* memberikan kedua struktur itu tanpa kapasitas dan tanpa aritmetika indeks, karena operasi yang mereka butuhkan tepat merupakan operasi $O(1)$ di **Tabel 7.4**.

- ***Stack*** memerlukan satu ujung tempat elemen ditambahkan sekaligus dibuang. Depan *single linked list* sangat cocok: `push` adalah penyisipan depan dan `pop` adalah penghapusan depan, keduanya $O(1)$. Puncak *stack* adalah simpul pertama, dan satu pointer, `top_`, sudah cukup.
- ***Queue*** memerlukan elemen ditambahkan di satu ujung dan dibuang di ujung lainnya. Dengan list ber-*head* dan ber-*tail*, penyisipan di belakang dan penghapusan di depan keduanya $O(1)$, sehingga `enqueue` menyisipkan pada *tail* (`rear_`) dan `dequeue` menghapus pada *head* (`front_`). Penugasan sebaliknya, menambah di depan dan membuang di belakang, akan menjadikan `dequeue` penghapusan belakang yang $\Theta(n)$; arahnya penting.

Gambar 7.16 memperlihatkan keduanya, berisi nomor identitas buku dari katalog Pustaka Ceria.



Gambar 7.16 (a) *Stack* berantai: `push` dan `pop` bekerja pada simpul pertama, `top_`. (b) *Queue* berantai: `enqueue` menautkan simpul baru sesudah `rear_`, `dequeue` membuang simpul pada `front_`.

7.10.2 Troli pengembalian dan antrean pemesanan Pustaka Ceria

Di Pustaka Ceria, buku yang dikembalikan ditumpuk di atas troli dan dikembalikan ke rak dari atas ke bawah, sehingga buku yang terakhir dikembalikan adalah yang pertama dirakkan: sebuah *stack*. Pembaca yang memesan buku yang sedang dipinjam menunggu dalam antrean pemesanan dan dilayani sesuai urutan permintaan: sebuah *queue*. **Kode 7.11** mengimplementasikan keduanya sebagai kelas C++ yang dibangun dari simpul bertaut tunggal.

Kode 7.11 `linked_stack_queue.cpp`: `LinkedStack` dan `LinkedQueue` yang dibangun dari simpul bertaut tunggal, dengan destruktor yang melepaskan setiap simpul.

```cpp
// linked_stack_queue.cpp -- a stack and a queue built on singly linked nodes,
// used by the Pustaka Ceria library: a cart of returned books and a hold queue.
#include <iostream>
#include <stdexcept>
#include <string>

struct Book {
    std::string id;
    std::string title;
};

class LinkedStack {                       // top of the stack = first node
    struct Node { Book data; Node *next; };
    Node *top_ = nullptr;
    int size_ = 0;
public:
    LinkedStack() = default;
    LinkedStack(const LinkedStack &) = delete;             // no accidental copies
    LinkedStack &operator=(const LinkedStack &) = delete;
    ~LinkedStack() { while (!isEmpty()) pop(); }           // frees every node

    bool isEmpty() const { return top_ == nullptr; }
    int size() const { return size_; }
    void push(const Book &b) {            // front insertion: O(1)
        top_ = new Node{b, top_};
        ++size_;
    }
    Book pop() {                          // front deletion: O(1)
        if (isEmpty()) throw std::underflow_error("pop from an empty stack");
        Node *del = top_;
        Book b = del->data;
        top_ = top_->next;
        delete del;
        --size_;
        return b;
    }
    const Book &peek() const {
        if (isEmpty()) throw std::underflow_error("peek at an empty stack");
        return top_->data;
    }
};
```

Kode 7.11 (Lanjutan) *linked_stack_queue.cpp: LinkedStack dan LinkedQueue yang dibangun dari simpul bertaut tunggal, dengan destruktor yang melepaskan setiap simpul.*

```
class LinkedQueue {                       // front = first node, rear = last node
    struct Node { Book data; Node *next; };
    Node *front_ = nullptr;
    Node *rear_ = nullptr;
    int size_ = 0;
public:
    LinkedQueue() = default;
    LinkedQueue(const LinkedQueue &) = delete;
    LinkedQueue &operator=(const LinkedQueue &) = delete;
    ~LinkedQueue() { while (!isEmpty()) dequeue(); }

    bool isEmpty() const { return front_ == nullptr; }
    int size() const { return size_; }
    void enqueue(const Book &b) {         // back insertion with a tail: O(1)
        Node *n = new Node{b, nullptr};
        if (isEmpty()) front_ = n;
        else rear_->next = n;
        rear_ = n;
        ++size_;
    }
    Book dequeue() {                      // front deletion: O(1)
        if (isEmpty()) throw std::underflow_error("dequeue from an empty queue");
        Node *del = front_;
        Book b = del->data;
        front_ = front_->next;
        if (front_ == nullptr) rear_ = nullptr;   // the queue became empty
        delete del;
        --size_;
        return b;
    }
};

int main() {
    LinkedStack cart;                     // returned books, the last one lands on top
    cart.push({"B1001", "Laskar Pelangi"});
    cart.push({"B1004", "Bumi Manusia"});
    cart.push({"B1007", "Filosofi Kopi"});
    std::cout << "Books on the return cart: " << cart.size() << '\n';
    std::cout << "Top of the cart: " << cart.peek().title << '\n';
    while (!cart.isEmpty()) {
        Book b = cart.pop();
        std::cout << "  shelve " << b.id << " " << b.title << '\n';
    }
```

Kode 7.11 (Lanjutan) *linked_stack_queue.cpp: LinkedStack dan LinkedQueue yang dibangun dari simpul bertaut tunggal, dengan destruktor yang melepaskan setiap simpul.*

```
    LinkedQueue holds;                    // hold requests, first come first served
    holds.enqueue({"B1002", "Negeri 5 Menara"});
    holds.enqueue({"B1005", "Ronggeng Dukuh Paruk"});
    holds.enqueue({"B1009", "Cantik Itu Luka"});
    std::cout << "Hold requests waiting: " << holds.size() << '\n';
    std::cout << "  serve " << holds.dequeue().title << '\n';
    holds.enqueue({"B1003", "Pulang"});
    while (!holds.isEmpty()) std::cout << "  serve " << holds.dequeue().title << '\n';

    try {
        holds.dequeue();
    } catch (const std::underflow_error &e) {
        std::cout << "Error caught: " << e.what() << '\n';
    }

    holds.enqueue({"B1010", "Gadis Kretek"});   // left in the queue on purpose:
    cart.push({"B1011", "Sang Pemimpi"});       // the destructors free these nodes
    std::cout << "Leaving main with " << holds.size() + cart.size() << " nodes still
linked\n";
    return 0;
}
```

```
Books on the return cart: 3
Top of the cart: Filosofi Kopi
  shelve B1007 Filosofi Kopi
  shelve B1004 Bumi Manusia
  shelve B1001 Laskar Pelangi
Hold requests waiting: 3
  serve Negeri 5 Menara
  serve Ronggeng Dukuh Paruk
  serve Cantik Itu Luka
  serve Pulang
Error caught: dequeue from an empty queue
Leaving main with 2 nodes still linked
```

Keluarannya memperlihatkan kedua disiplin itu. Troli menerima Laskar Pelangi, Bumi Manusia, dan Filosofi Kopi lalu merakkannya dalam urutan terbalik. Antrean pemesanan melayani Negeri 5 Menara lebih dulu, dan Pulang, yang dipesan terakhir, dilayani terakhir. Meminta satu permintaan lagi dari antrean yang kosong memunculkan eksepsi `std::underflow_error`, yang ditangkap `main`.

Program ini memperkenalkan tiga fitur yang tidak dimiliki program-program sebelumnya dalam bab ini, dan masing-masing menangkal bahaya yang nyata.

- **Tipe simpulnya privat.** `struct Node` dideklarasikan di dalam setiap kelas, sehingga tidak ada kode di luar kelas yang dapat mencapai sebuah simpul dan merusak tautannya. Kelas-kelas berbasis larik di Bab 5 menyembunyikan lariknya dengan alasan yang sama.
- **Destruktor melepaskan simpul-simpulnya.** `~LinkedStack()` melakukan `pop` sampai *stack* kosong, dan `~LinkedQueue()` melakukan `dequeue` sampai *queue* kosong. C++ memanggil destruktor secara otomatis ketika masa hidup sebuah objek berakhir, di sini pada akhir `main`, sehingga dua simpul yang sengaja ditinggalkan di dalam struktur ("Leaving main with 2 nodes still linked") dilepaskan tanpa pemanggilan tersurat. Mengikatkan pelepasan sumber daya pada masa hidup objek dengan cara ini disebut *RAII* (*resource acquisition is initialization*), dan itulah jawaban C++ yang idiomatis atas masalah lupa memanggil `clear` di Subbab 7.5.7.
- **Penyalinan dilarang.** Baris `LinkedStack(const LinkedStack &) = delete;` dan operator penugasan padanannya memberi tahu kompilator agar tidak membangkitkan salinan bawaannya, yang hanya akan menyalin pointer `top_`. Dua *stack* lalu berbagi simpul yang sama,

dan ketika kedua destruktornya berjalan, setiap simpul akan dihapus dua kali. Inilah *rule of three* (aturan tiga): kelas yang memerlukan destruktor hampir selalu memerlukan konstruktor salinan dan penugasan salinan yang benar, atau harus melarang keduanya.

`dequeue` memuat satu-satunya baris halus pada *queue* itu, `if (front_ == nullptr) rear_ = nullptr;`. Ketika simpul terakhir keluar, `front_` menjadi NULL dengan sendirinya, tetapi `rear_` masih menunjuk simpul yang baru saja dihapus. Kelas ini kebetulan tidak membaca `rear_` selama `front_` bernilai NULL, sehingga ia akan selamat tanpa baris itu; tetapi versi yang menguji kekosongan dengan `rear_ == nullptr`, seperti list *head-tail* di Subbab 7.8 menguji `tail`, akan menulis `rear_->next` ke memori yang sudah dibebaskan pada `enqueue` berikutnya. Mengembalikan kedua pointer ke NULL setiap kali *queue* menjadi kosong membuat setiap uji semacam itu aman. Soal Latihan 7.12 meminta Anda mengikuti kegagalan itu sampai tuntas dan menyebutkan apa yang akan dilaporkan pemeriksa memori.

Versi Java-nya, `LinkedStackQueue.java`, memakai `record Book` dan melempar `NoSuchElementException`; versi Python-nya, `linked_stack_queue.py`, memakai `dataclass` beku dan memunculkan `IndexError`. Keduanya mencetak keluaran yang sama dan tidak memerlukan destruktor, karena *garbage collector* mengambil kembali simpul-simpulnya.

7.10.3 Berantai atau berbasis larik?

Kedua implementasi memberikan `push`, `pop`, `enqueue`, dan `dequeue` dalam $O(1)$. Versi berantai tidak pernah penuh dan memakai memori sebanding dengan banyaknya elemen saat itu. Versi larik memakai lebih sedikit memori per elemen (tanpa pointer) dan dalam praktiknya lebih cepat, karena elemen-elemennya berbagi *cache line* dan tidak memanggil alokator pada setiap operasi. Larik yang dapat tumbuh, misalnya `std::vector` yang dipakai sebagai *stack*, memperoleh `push` $O(1)$ *teramortisasi*, artinya sesekali pertumbuhan ulang yang mahal dibayar oleh banyak `push` yang murah. Karena itulah `std::stack` dan `std::queue` pada pustaka baku C++ secara bawaan memakai `std::deque`, barisan larik berukuran tetap, bukan list; `collections.deque` milik Python menempuh pendekatan campuran yang serupa, seperti dijelaskan Subbab 7.12. *Stack* dan *queue* berantai tetap pilihan yang tepat bila elemennya besar, bila kapasitas tetap tidak dapat diterima dan jeda pertumbuhan ulang harus dihindari, atau bila simpul-simpulnya sudah menjadi bagian dari struktur berantai lain.

7.11 Keamanan Memori: Membuktikan Tidak Ada Kebocoran

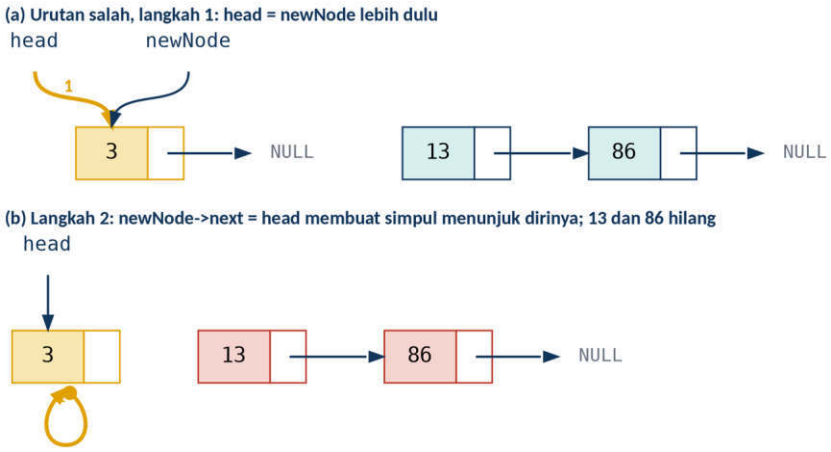
7.11.1 Mengapa “hasilnya sudah benar” belum cukup

Setiap program C++ dalam bab ini memanggil `new` dan `delete` secara langsung, dan kesalahan pada salah satunya biasanya *sama sekali tidak menimbulkan gejala yang terlihat*. Simpul yang bocor tidak mengubah keluaran. Simpul yang dibaca sesudah dihapus sering masih berisi nilai lamanya, karena belum ada yang menimpanya, sehingga program mencetak jawaban yang benar, hari ini, pada mesin ini. Program yang sama dapat runtuh besok, dengan opsi kompilator yang lain atau masukan yang berbeda, atau, lebih buruk lagi, diam-diam merusak data lain. Empat jenis kesalahan mencakup hampir semua galat memori pada *linked list*.

1. **Kebocoran memori** (*memory leak*). Sebuah simpul tidak pernah dihapus. Penyebab khasnya adalah lupa memanggil `clear`, atau memindahkan satu-satunya pointer ke sebuah simpul sebelum menghapusnya.

2. **Pemakaian sesudah pembebasan** (*use after free*). Sebuah simpul dibaca atau ditulis sesudah `delete`. Penyebab khasnya adalah pointer kedua (`second` yang disimpan, `tail` yang basi, `aux` pembantu) yang masih memegang alamat simpul yang dihapus lewat pointer pertama.
3. **Tautan menggantung** (*dangling link*). Kasus khusus pemakaian sesudah pembebasan di dalam list itu sendiri: sebuah simpul dihapus tetapi sebuah medan `next` (atau `tail`) masih menunjuknya, seperti ketika baris 11 **Pseudokode 7.5** terlupa.
4. **Pembebasan ganda** (*double free*). Simpul yang sama dihapus dua kali, biasanya karena dua pointer ke simpul itu sama-sama melewati `clear`.

Kesalahan kelima bukanlah galat memori dengan sendirinya tetapi menghasilkan galat pertama dalam skala besar: mengerjakan dua penugasan pointer dengan urutan yang salah. **Gambar 7.17** memperlihatkan `frontInsertion` dengan kedua barisnya tertukar. Sesudah `head = newNode`, satu-satunya pointer ke simpul pertama lama hilang; `newNode->next = head` lalu membuat simpul baru menunjuk dirinya sendiri. List kini terbaca 3, 3, 3, selamanya, dan simpul 13 dan 86 bocor.



Gambar 7.17 Penyisipan depan dengan kedua penugasannya dalam urutan yang salah: (a) `head` berpindah lebih dulu, (b) simpul baru lalu menunjuk dirinya sendiri dan simpul lama 13 dan 86 tidak akan pernah dapat dicapai lagi.

Karena galat-galat ini bersembunyi, kita memeriksanya dengan perkakas yang mengawasi setiap alokasi dan setiap akses memori selama program berjalan. **Kode 7.12** adalah program kecil yang memuat keempat galat itu, dipilih lewat argumen baris perintah; tanpa argumen, program berjalan dengan benar.

Kode 7.12 `memory_bugs.cpp`: satu eksekusi yang benar dan empat galat memori yang disengaja (`leak`, `uaf`, `dangle`, `double`).

```cpp
// memory_bugs.cpp -- four classic linked-list memory bugs, one per argument.
//   ./memory_bugs        correct program
//   ./memory_bugs leak   forgets to free the list
//   ./memory_bugs uaf    keeps a pointer into the list after clearing it
//   ./memory_bugs dangle deletes the last node but leaves a pointer to it
//   ./memory_bugs double frees the same nodes twice
#include <cstdio>
#include <cstring>

struct Node {
    int data;
    Node *next;
};

Node *build() {                          // 1 -> 2 -> 3 -> NULL
    Node *head = nullptr;
    for (int x = 3; x >= 1; --x) head = new Node{x, head};
    return head;
}

void show(const Node *head) {
    for (; head != nullptr; head = head->next) std::printf("%d ", head->data);
    std::printf("\n");
}

void clearRight(Node *&head) {
    while (head != nullptr) {
        Node *del = head;
        head = head->next;               // step first ...
        delete del;                      // ... then free
    }
}

void backDeletionWrong(Node *head) {     // assumes at least two nodes
    Node *aux = head;
    while (aux->next->next != nullptr) aux = aux->next;
    delete aux->next;                    // BUG: aux->next is not set to nullptr
}
```

Kode 7.12 (Lanjutan) *memory_bugs.cpp: satu eksekusi yang benar dan empat galat memori yang disengaja (leak, uaf, dangle, double).*

```
int main(int argc, char *argv[]) {
    const char *mode = argc > 1 ? argv[1] : "ok";
    Node *head = build();
    show(head);
    if (std::strcmp(mode, "leak") == 0) {
        return 0;                          // BUG: the three nodes are never freed
    } else if (std::strcmp(mode, "uaf") == 0) {
        Node *second = head->next;         // a second way into the list
        clearRight(head);                  // head is nullptr now, but second is not
        show(second);                      // BUG: reads freed nodes
    } else if (std::strcmp(mode, "dangle") == 0) {
        backDeletionWrong(head);
        show(head);                        // walks into the freed node
        clearRight(head);
    } else if (std::strcmp(mode, "double") == 0) {
        Node *copy = head;                 // copies the pointer, not the nodes
        clearRight(copy);                  // frees all three nodes
        clearRight(head);                  // BUG: frees them again
    } else {
        clearRight(head);
    }
    std::printf("done (%s)\n", mode);
    return 0;
}
```

7.11.2 Valgrind

Valgrind menjalankan program yang tidak diubah di atas prosesor tiruan dan memeriksa setiap akses memori dan setiap alokasi; perkakas bawaannya, Memcheck, melaporkan akses baca dan akses tulis yang tidak sah, pemakaian memori yang sudah dibebaskan, pembebasan ganda, dan kebocoran [8]. Valgrind berjalan di Linux (dan di WSL pada Windows); program berjalan 20 sampai 50 kali lebih lambat di bawahnya, yang tidak menjadi soal untuk pengujian. Rilis terkininya per September 2026 adalah Valgrind 3.27.1, yang diterbitkan pada 20 Mei 2026 [8]; eksekusi di bawah ini memakai rilis 3.22 yang dikemas bersama Ubuntu 24.04, yang format keluarannya sama. Kompilasilah dengan `-g` supaya laporan menyebutkan baris sumber, dan dengan `-O0` supaya baris-barisnya tepat sesuai dengan sumbernya:

```
g++ -std=c++17 -g -O0 -Wall -Wextra -o memory_bugs memory_bugs.cpp
valgrind --leak-check=full ./memory_bugs leak
```

```
==1776== Memcheck, a memory error detector
==1776== Command: ./memory_bugs leak
==1776==
1 2 3
==1776==
==1776== HEAP SUMMARY:
==1776==     in use at exit: 48 bytes in 3 blocks
==1776==   total heap usage: 5 allocs, 2 frees, 77,872 bytes allocated
==1776==
==1776== 48 (16 direct, 32 indirect) bytes in 1 blocks are definitely lost in loss record 2
of 2
==1776==    at 0x4846FA3: operator new(unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1776==    by 0x10920F: build() (memory_bugs.cpp:17)
==1776==    by 0x109369: main (memory_bugs.cpp:42)
==1776==
==1776== LEAK SUMMARY:
==1776==    definitely lost: 16 bytes in 1 blocks
==1776==    indirectly lost: 32 bytes in 2 blocks
==1776==      possibly lost: 0 bytes in 0 blocks
==1776==    still reachable: 0 bytes in 0 blocks
==1776==         suppressed: 0 bytes in 0 blocks
==1776==
==1776== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Bacalah laporannya dari bawah ke atas. `LEAK SUMMARY` menyatakan 16 bita *definitely lost* (pasti hilang) dan 32 bita *indirectly lost* (hilang secara tidak langsung). Pembedaannya mendidik. Hanya simpul pertama yang "pasti" hilang: ia ditunjuk oleh `head`, dan `head` lenyap ketika `main` kembali. Dua simpul lainnya hilang "secara tidak langsung": pointer ke keduanya masih ada, tetapi hanya di dalam simpul pertama, yang dirinya sendiri hilang. Maka list yang bocor tampil sebagai satu kebocoran pasti seukuran satu simpul ditambah kebocoran tidak langsung untuk sisa rantainya, dan memperbaiki kebocoran pasti itu (dengan memanggil `clear`) memperbaiki semuanya. Catatan di atas `LEAK SUMMARY` menyebutkan di mana memori yang hilang itu dialokasikan: oleh `operator new`, yang dipanggil dari `build()` pada baris 17 `memory_bugs.cpp`, yang dipanggil dari `main` pada baris 42. Perhatikan pula `total heap usage: 5 allocs, 2 frees`: tiga dari lima alokasi adalah simpul-simpul kita, dan dua lainnya milik pustaka *run-time* C++, yang membebaskannya sendiri.

Eksekusi pemakaian-sesudah-pembebasan lebih mengkhawatirkan, karena programnya tampak bekerja:

```
valgrind --leak-check=full ./memory_bugs uaf
```

```
==1786== Memcheck, a memory error detector
==1786== Command: ./memory_bugs uaf
==1786==
==1786== Invalid read of size 4
==1786==    at 0x109247: show(Node const*) (memory_bugs.cpp:22)
==1786==    by 0x1093DB: main (memory_bugs.cpp:49)
==1786==  Address 0x4e260d0 is 0 bytes inside a block of size 16 free'd
==1786==    at 0x484A61D: operator delete(void*, unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1786==    by 0x1092C3: clearRight(Node*&) (memory_bugs.cpp:30)
==1786==    by 0x1093CF: main (memory_bugs.cpp:48)
==1786==  Block was alloc'd at
==1786==    at 0x4846FA3: operator new(unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1786==    by 0x10920F: build() (memory_bugs.cpp:17)
==1786==    by 0x109369: main (memory_bugs.cpp:42)
==1786==
1 2 3
2 3
done (uaf)
==1786==
==1786== HEAP SUMMARY:
==1786==     in use at exit: 0 bytes in 0 blocks
==1786==   total heap usage: 5 allocs, 5 frees, 77,872 bytes allocated
==1786==
==1786== All heap blocks were freed -- no leaks are possible
==1786==
==1786== ERROR SUMMARY: 4 errors from 2 contexts (suppressed: 0 from 0)
```

Program mencetak `2 3`, nilai-nilai yang benar, dari simpul-simpul yang sudah dihapusnya. Valgrind melaporkan setiap akses buruk dengan tiga jejak tumpukan: di mana pembacaan tidak sah terjadi (`show`, baris 22, dipanggil dari `main` pada baris 49), di mana blok itu dibebaskan (`clearRight`, baris 30, dipanggil dari baris 48), dan di mana ia dialokasikan (`build`, baris 17). Bersama-sama ketiganya menceritakan kisah lengkapnya: simpul-simpul dibuat oleh `build`, dilepaskan oleh `clearRight(head)`, dan dibaca sesudahnya lewat pointer basi `second`. (Eksekusi itu menemukan dua konteks galat: pembacaan 4 bita atas `data` yang diperlihatkan di atas dan pembacaan 8 bita atas `next` pada baris yang sama, yang laporannya berbentuk sama dan dihilangkan di sini.) Ketika program yang sama dikompilasi dengan `-O2` dan dijalankan *tanpa* Valgrind pada mesin yang dipakai untuk buku ini, ia runtuh dengan *segmentation fault*; dengan pengaturan lain ia dapat mencetak sampah atau tampak bekerja. Ketidakpastian itulah yang membuat perkakas ini diperlukan.

Akhirnya, bukti untuk program yang benar. **Kode 7.3** di bawah Valgrind diakhiri dengan:

```
==3636==
==3636== HEAP SUMMARY:
==3636==     in use at exit: 0 bytes in 0 blocks
==3636==   total heap usage: 6 allocs, 6 frees, 77,888 bytes allocated
==3636==
==3636== All heap blocks were freed -- no leaks are possible
==3636==
==3636== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

`All heap blocks were freed -- no leaks are possible` dan `ERROR SUMMARY: 0 errors` bersama-sama adalah sertifikat yang kita inginkan; keenam alokasinya adalah keempat simpul list dan dua milik pustaka *run-time*. Setiap program C++ dalam bab ini dijalankan di bawah Valgrind dengan `--leak-check=full` dan menghasilkan kedua baris itu.

7.11.3 AddressSanitizer

AddressSanitizer (ASan) menempuh pendekatan sebaliknya: alih-alih meniru prosesor, kompilator menyisipkan pemeriksaan ke dalam program itu sendiri, dengan menandai bita-bita di sekitar setiap alokasi dan setiap blok yang sudah dibebaskan sebagai beracun dalam sebuah *shadow memory* (memori bayangan) [9]. Program yang dihasilkan berjalan hanya sekitar dua kali lebih lambat daripada biasa, bekerja di Linux, macOS, dan Windows (dengan GCC, Clang, dan kompilator Microsoft), dan langsung melaporkan akses buruk yang pertama. Pendampingnya, LeakSanitizer, melaporkan kebocoran ketika program keluar, dan `-fsanitize=undefined` menambahkan pemeriksaan perilaku tak terdefinisi seperti luapan bilangan bertanda. Opsi `strip_path_prefix` memperpendek nama berkas dalam laporan.

```
g++ -std=c++17 -g -O1 -Wall -Wextra -fsanitize=address,undefined \
    -fno-omit-frame-pointer -o memory_bugs memory_bugs.cpp
ASAN_OPTIONS=strip_path_prefix=$PWD/ ./memory_bugs dangle
```

```
=================================================================
==1805==ERROR: AddressSanitizer: heap-use-after-free on address 0x502000000010 at pc
0x55d758b734e0 bp 0x7fffe07441a0 sp 0x7fffe0744190
READ of size 4 at 0x502000000010 thread T0
    #0 0x55d758b734df in show(Node const*) memory_bugs.cpp:22
    #1 0x55d758b73cac in main memory_bugs.cpp:52

0x502000000010 is located 0 bytes inside of 16-byte region [0x502000000010,0x502000000020)
freed by thread T0 here:
    #0 0x7f1fd44ff5e8 in operator delete(void*, unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:164
    #1 0x55d758b73831 in backDeletionWrong(Node*) memory_bugs.cpp:37
    #2 0x55d758b73c8d in main memory_bugs.cpp:51

previously allocated by thread T0 here:
    #0 0x7f1fd44fe548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
    #1 0x55d758b733d7 in build() memory_bugs.cpp:17
    #2 0x55d758b739ba in main memory_bugs.cpp:42

SUMMARY: AddressSanitizer: heap-use-after-free memory_bugs.cpp:22 in show(Node const*)
```

(Bingkai di dalam kode awal-jalan pustaka C, yang menyusul `main` pada setiap jejak, dan peta *shadow memory* yang dicetak ASan sesudah ringkasannya dihilangkan.) Inilah galat tautan menggantung dari `backDeletionWrong`: simpul terakhir dihapus pada baris 37, tetapi simpul sebelumnya masih menunjuknya, sehingga `show` pada baris 22 membaca medan `data`-nya. ASan melaporkan jenis galatnya (`heap-use-after-free`), ukuran aksesnya (`READ of size 4`, sebuah `int`), letak alamat di dalam blok yang dibebaskan (`0 bytes inside of 16-byte region`, medan `data` pada ofset 0 dari simpul 16 bita), dan tiga jejak tumpukan yang sama dengan Valgrind, lalu menghentikan program.

Eksekusi pembebasan ganda memperlihatkan bagaimana pembebasan ganda biasanya tertangkap lebih awal lagi:

```
ASAN_OPTIONS=strip_path_prefix=$PWD/ ./memory_bugs double
```

```
=================================================================
==1806==ERROR: AddressSanitizer: heap-use-after-free on address 0x502000000058 at pc
0x55c58ece3651 bp 0x7ffce90d0d90 sp 0x7ffce90d0d80
READ of size 8 at 0x502000000058 thread T0
    #0 0x55c58ece3650 in clearRight(Node*&) memory_bugs.cpp:29
    #1 0x55c58ece3a98 in main memory_bugs.cpp:57

0x502000000058 is located 8 bytes inside of 16-byte region [0x502000000050,0x502000000060)
freed by thread T0 here:
    #0 0x7f1f660ff5e8 in operator delete(void*, unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:164
    #1 0x55c58ece35bf in clearRight(Node*&) memory_bugs.cpp:30
    #2 0x55c58ece3a8e in main memory_bugs.cpp:56

previously allocated by thread T0 here:
    #0 0x7f1f660fe548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
    #1 0x55c58ece33d7 in build() memory_bugs.cpp:17
    #2 0x55c58ece39ba in main memory_bugs.cpp:42

SUMMARY: AddressSanitizer: heap-use-after-free memory_bugs.cpp:29 in clearRight(Node*&)
```

`clearRight(head)` yang kedua (baris 57) dimulai dengan membaca `head->next` pada baris 29 dari simpul pertama, yang sudah dilepaskan oleh `clearRight(copy)` yang pertama (baris 56), sehingga ASan berhenti pada pembacaan itu, `8 bytes inside` simpulnya, yaitu medan `next`-nya, bahkan sebelum `delete` yang kedua sempat dicoba. Tanpa ASan, eksekusi yang sama lazimnya dibatalkan oleh pustaka C dengan pesan `free(): double free detected in tcache 2`, yang menyebutkan apa yang terjadi tetapi tidak di mana. Kebocoran, akhirnya, dilaporkan LeakSanitizer ketika program keluar:

```
ASAN_OPTIONS=strip_path_prefix=$PWD/ ./memory_bugs leak
```

```
=================================================================
==1810==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 16 byte(s) in 1 object(s) allocated from:
    #0 0x7f4b9c8fe548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
    #1 0x55ed654b03d7 in build() memory_bugs.cpp:17
    #2 0x55ed654b09ba in main memory_bugs.cpp:42

Indirect leak of 32 byte(s) in 2 object(s) allocated from:
    #0 0x7f4b9c8fe548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
    #1 0x55ed654b03d7 in build() memory_bugs.cpp:17
    #2 0x55ed654b09ba in main memory_bugs.cpp:42

SUMMARY: AddressSanitizer: 48 byte(s) leaked in 3 allocation(s).
```

Kebocoran "*direct*" dan "*indirect*" berpadanan tepat dengan "*definitely*" dan "*indirectly lost*" pada Valgrind. Setiap program C++ dalam bab ini juga dibangun dengan `-fsanitize=address,undefined` dan dijalankan: semuanya mencetak keluaran normalnya dan tidak ada yang lain.

7.11.4 Daftar periksa untuk setiap fungsi linked list

Keempat galat itu memiliki empat kebiasaan penangkalnya, dan kebiasaan kelima menangkap apa yang terlewat oleh yang lain.

1. **Simpan, lalu pindahkan, lalu hapus.** Sebelum menghapus sebuah simpul, simpan alamatnya di `del`, pindahkan setiap pointer yang menuju ke sana (`head`, `next` milik pendahulu, `tail`), dan hapus paling akhir.

2. **Jangan tinggalkan pointer ke simpul yang sudah dibebaskan.** Sesudah penghapusan, tanyakan pointer mana saja yang memegang alamat simpul itu: `head`, `tail`, `next` milik pendahulu, pembantu yang disimpan. Setiap satu harus dipindahkan atau diisi `nullptr`.
3. **Setiap `new` memiliki tepat satu pemilik yang menghapusnya.** Dalam list, pemilik simpul adalah list itu; lepaskan semua simpul di tepat satu tempat (`clear`, `destroy`, atau destruktor) dan pastikan bahwa menyalin list menyalin simpul-simpulnya atau dilarang (Subbab 7.10.2).
4. **Periksa keempat bentuk.** Ujilah setiap fungsi pada list kosong, list satu simpul, dan pada simpul pertama, tengah, dan terakhir. Sebagian besar galat list tinggal di tepat salah satu kasus itu.
5. **Jalankan perkakasnya.** Kompilasilah pengujian dengan `-fsanitize=address,undefined`, atau jalankan di bawah `valgrind --leak-check=full`, setiap kali. Rangkaian uji yang lulus tetapi tidak pernah dijalankan di bawah pemeriksa memori membuktikan jauh lebih sedikit daripada kelihatannya.

7.12 Linked List dalam Pustaka Baku

7.12.1 C++: `std::forward_list` dan `std::list`

Pustaka baku C++ menyediakan kedua jenis *linked list* [10]. `std::forward_list<T>`, yang ditambahkan dalam C++11, adalah *single linked list* yang dirancang tanpa ongkos tambahan dibandingkan dengan yang ditulis tangan: objeknya pada dasarnya satu pointer (ke simpul pertama) dan setiap simpul menyimpan elemen dan satu pointer `next`. `std::list<T>` adalah *double linked list*, struktur Bab 9. **Kode 7.13** memakai keduanya.

Kode 7.13 `std_lists.cpp`: `std::forward_list` yang bertaut tunggal dan `std::list` yang bertaut ganda.

```cpp
// std_lists.cpp -- the standard library's linked lists: std::forward_list
// (singly linked) and std::list (doubly linked).
#include <forward_list>
#include <iostream>
#include <iterator>
#include <list>

template <typename Seq>
void print(const char *label, const Seq &s) {
    std::cout << label;
    for (int x : s) std::cout << ' ' << x;
    std::cout << '\n';
}

int main() {
    std::forward_list<int> f = {3, 1, 4, 1, 5};
    print("start:                ", f);
    f.push_front(9);                          // O(1): a new first node
    print("add 9 at the front:   ", f);
    f.insert_after(f.begin(), 2);             // O(1): after a known node
    print("insert 2 after first:", f);
    f.remove(1);                              // O(n): every node holding 1
    print("remove every 1:       ", f);
    f.reverse();                              // O(n): relinks, copies nothing
    print("reverse:              ", f);
    f.sort();                                 // O(n log n): merge sort on nodes
    print("sort:                 ", f);
    // forward_list has no size(): counting the nodes is a walk, O(n)
    std::cout << "length:                " << std::distance(f.begin(), f.end()) << '\n';

    std::list<int> d(f.begin(), f.end());     // doubly linked: both ends are O(1)
    d.push_back(7);
    d.push_front(0);
    d.pop_back();
    print("doubly linked:        ", d);
    std::cout << "front " << d.front() << ", back " << d.back()
              << ", size " << d.size() << '\n';
    return 0;
}
```

```
start:                3 1 4 1 5
add 9 at the front:   9 3 1 4 1 5
insert 2 after first: 9 2 3 1 4 1 5
remove every 1:       9 2 3 4 5
reverse:              5 4 3 2 9
sort:                 2 3 4 5 9
length:               5
doubly linked:        0 2 3 4 5 9
front 0, back 9, size 6
```

`std::forward_list` menawarkan tepat operasi-operasi yang murah pada *single linked list*, dan menamainya dengan jujur. Penyisipan dan penghapusan bernama `insert_after` dan `erase_after`, karena, seperti diperlihatkan Subbab 7.6.3, *single linked list* hanya dapat mengubah tautan *sesudah* simpul yang diketahui; `before_begin()` menyediakan posisi sebelum elemen pertama supaya elemen pertama tidak memerlukan kasus khusus, gagasan simpul *dummy* dari Subbab 7.7. Ada `push_front` tetapi tidak ada `push_back`, dan, yang mula-mula mengejutkan, tidak ada `size()`: menyimpan hitungan akan berongkos satu medan tambahan dan satu pembaruan pada setiap operasi, yang ditolak rancangannya, sehingga panjangnya harus dihitung dengan berjalan (`std::distance`), dalam waktu $\Theta(n)$. `remove`, `reverse`, dan `sort` adalah fungsi anggota karena ketiganya dapat menautkan ulang simpul alih-alih memindahkan elemen; `sort` adalah *merge sort*, $O(n \lg n)$, yang tidak memerlukan akses acak.

`std::list` menambahkan `push_back`, `pop_back`, `back()`, dan `size()` berwaktu konstan (diwajibkan sejak C++11), semuanya mungkin karena setiap simpul mengenal pendahulunya. Kedua kontainer berbagi satu jaminan penting yang tidak dimiliki larik: menyisipkan atau menghapus elemen tidak pernah memindahkan elemen lain, sehingga pointer dan iterator ke elemen-elemen itu tetap sah. Jaminan itu, bukan kecepatan, sering menjadi alasan sesungguhnya untuk memilih list di C++.

Per September 2026, seri rilis GCC terbaru adalah GCC 16, yang rilis pertamanya, 16.1, terbit pada April 2026 dan menjadikan C++20 mode bahasa bawaannya [11]. Tidak ada bagian bab ini yang bergantung padanya: `std::forward_list` dan `std::list` berperilaku identik dalam C++17 dan setiap standar sesudahnya.

7.12.2 Java: `java.util.LinkedList`

`LinkedList<E>` milik Java adalah *double linked list* yang mengimplementasikan antarmuka `List` dan `Deque`, dan, sejak Java 21, `SequencedCollection`, yang menambahkan `addFirst`, `addLast`, `getFirst`, `getLast`, dan `reversed()` pada setiap koleksi berurutan [12]. Kelas itu tidak tersinkronisasi, sehingga `LinkedList` yang dipakai bersama oleh beberapa *thread* harus dilindungi oleh pemanggilnya. **Kode 7.14** mengerjakan operasi yang sama dengan **Kode 7.13** dan mencetak keluaran yang sama.

Kode 7.14 `LibraryLists.java`: `java.util.LinkedList` yang dipakai sebagai list dan sebagai *deque*.

```
// LibraryLists.java -- Java's built-in linked list, java.util.LinkedList
// (doubly linked), used as a list and as a deque.
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

public class LibraryLists {
    static void print(String label, List<Integer> s) {
        StringBuilder sb = new StringBuilder(label);
        for (int x : s) sb.append(' ').append(x);
        System.out.println(sb);
    }

    public static void main(String[] args) {
        LinkedList<Integer> f = new LinkedList<>(List.of(3, 1, 4, 1, 5));
        print("start:                ", f);
        f.addFirst(9);                          // O(1)
        print("add 9 at the front:  ", f);
        f.add(1, 2);                            // walks to index 1, then links
        print("insert 2 after first:", f);
        f.removeIf(x -> x == 1);                // O(n)
        print("remove every 1:      ", f);
        Collections.reverse(f);                 // O(n)
        print("reverse:              ", f);
        Collections.sort(f);                    // copies to an array, sorts, copies back
        print("sort:                 ", f);
        System.out.println("length:               " + f.size());   // O(1): kept in a field

        LinkedList<Integer> d = new LinkedList<>(f);
        d.addLast(7);
        d.addFirst(0);
        d.removeLast();
        print("doubly linked:       ", d);
        System.out.println("front " + d.getFirst() + ", back " + d.getLast() + ", size " +
d.size());
    }
}
```

Pustaka Java tidak memiliki kelas *single linked list*; `LinkedList` melayani kedua peran, dengan harga pointer kedua per simpul, dan ia menyimpan referensi ke objek `Integer` terbungkus alih-alih bilangannya sendiri. `size()`-nya $O(1)$ karena hitungannya disimpan dalam sebuah medan. Perhatikan komentar pada `add(1, 2)`, yang harus berjalan ke indeks 1, dan pada `Collections.sort`, yang menyalin elemen-elemen ke larik, mengurutkan larik itu, lalu menuliskan elemen-elemennya kembali, karena mengurutkan lewat akses sekuensial list akan lambat. Dalam praktiknya `ArrayList` (untuk list) dan `ArrayDeque` (untuk *stack* dan *queue*) lebih cepat daripada `LinkedList` untuk hampir setiap beban kerja, dengan alasan *cache* di Subbab 7.3.2; `LinkedList` terutama menguntungkan bila elemen disisipkan atau dibuang di tengah lewat iterator yang sudah berada di posisi itu.

7.12.3 Python: mengapa tidak ada linked list bawaan, dan `collections.deque`

`list` bawaan Python sama sekali bukan *linked list*: ia adalah larik dinamis berisi referensi, dengan pengindeksan $O(1)$ dan `append` $O(1)$ teramortisasi, tetapi `insert(0, x)` dan `pop(0)` bernilai $\Theta(n)$ karena menggeser setiap elemen. Python tidak memiliki kelas *linked list* bawaan, dan alasannya baik. Pada CPython setiap objek sudah membawa kepala miliknya sendiri (hitungan referensi dan pointer tipe), sehingga objek simpul seperti pada **Kode 7.5** berongkos sekitar 48 bita bahkan dengan `__slots__`, dan setiap langkah `aux = aux.next` dijalankan sebagai *bytecode* yang ditafsirkan. *Linked list* dalam Python murni karenanya lambat dan boros memori dibandingkan dengan `list` bawaan, yang diimplementasikan dalam C.

Bila sebuah program memerlukan apa yang menjadi keunggulan *linked list*, penyisipan dan pembuangan murah di kedua ujung, jawaban Python adalah `collections.deque`. Dokumentasinya menjanjikan kinerja "kira-kira sama, $O(1)$, ke arah mana pun" untuk `append` dan `pop` di kedua ujung, sedangkan akses berindeks "melambat menjadi $O(n)$ di tengah" [13]. Di dalamnya, CPython mengimplementasikan *deque* sebagai *double linked list berisi blok-blok berukuran tetap*, masing-masing memuat 64 referensi [14]. Rancangan campuran ini mempertahankan janji *linked list* bahwa `append` dan `pop` tidak pernah memindahkan elemen lain, sambil memangkas ongkos pointer 64 kali lipat dan menjaga elemen-elemen bertetangga dalam blok memori yang sama demi *cache*. **Kode 7.15** mengerjakan operasi yang sama sekali lagi.

Kode 7.15 `library_lists.py`: `collections.deque`, padanan bawaan Python yang paling dekat dengan *linked list*.

```
# library_lists.py -- Python has no built-in linked list class;
# collections.deque (a doubly linked list of 64-slot blocks) is the closest.
from collections import deque


def show(label, s):
    print(label + "".join(f" {x}" for x in s))


def main():
    f = deque([3, 1, 4, 1, 5])
    show("start:                ", f)
    f.appendleft(9)                         # O(1)
    show("add 9 at the front:   ", f)
    f.insert(1, 2)                          # O(n) in general
    show("insert 2 after first:", f)
    f = deque(x for x in f if x != 1)       # deque has no remove-all: rebuild
    show("remove every 1:       ", f)
    f.reverse()                             # in place, O(n)
    show("reverse:              ", f)
    f = deque(sorted(f))                    # deque has no sort(): sorted() makes a list
    show("sort:                 ", f)
    print(f"length:                {len(f)}")   # O(1)

    d = deque(f)
    d.append(7)
    d.appendleft(0)
    d.pop()
    show("doubly linked:        ", d)
    print(f"front {d[0]}, back {d[-1]}, size {len(d)}")


if __name__ == "__main__":
    main()
```

Deque tidak dapat mengerjakan segala yang dapat dikerjakan `list`: ia tidak memiliki metode `sort` (program membangun *deque* terurut dari `sorted(f)`), dan `insert(i, x)` di tengah bernilai $O(n)$. Menulis kelas simpul sendiri, seperti dalam bab ini, tetap merupakan hal yang tepat dalam Python ketika Anda sedang mempelajari strukturnya, atau ketika simpul-simpulnya menjadi bagian dari struktur berantai yang lebih besar yang tidak disediakan pustaka mana pun, seperti pohon dan graf di bab-bab selanjutnya.

Tabel 7.5 membandingkan ketiga pustaka itu.

Tabel 7.5 *Linked list* pada pustaka baku C++, Java, dan Python.

	C++ `std::forward_list`	C++ `std::list`	Java `LinkedList`	Python `collections.deque`
Struktur	bertaut tunggal	bertaut ganda	bertaut ganda	*double linked list* berisi blok 64 slot
Sisip / hapus di depan	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Sisip / hapus di belakang	tidak disediakan	$O(1)$	$O(1)$	$O(1)$
Sisip sesudah posisi yang diketahui	$O(1)$ `insert_after`	$O(1)$ `insert`	$O(1)$ lewat `ListIterator.add`	$O(n)$ `insert`
Akses berdasarkan indeks	tidak disediakan	tidak disediakan	$\Theta(n)$ `get(i)`	$O(1)$ di ujung, $O(n)$ di tengah
Ukuran	$\Theta(n)$ berjalan	$O(1)$	$O(1)$	$O(1)$
Pengurutan di tempat	`sort()`, *merge sort*	`sort()`, *merge sort*	lewat `Collections.sort` (salinan larik)	tidak disediakan

7.12.4 Mengukur: larik melawan list pada mesin sungguhan

Ongkos-ongkos di Subbab 7.3 meramalkan dua hal: menyisipkan di depan berongkos total $\Theta(n^2)$ untuk larik dan $\Theta(n)$ untuk list, sedangkan menelusuri semua elemen berongkos $\Theta(n)$ untuk keduanya. **Kode 7.16** menguji kedua ramalan itu. Bagian 1 menyisipkan n nilai di depan sebuah `std::vector` dan sebuah `std::forward_list`. Bagian 2 menjumlahkan dua juta nilai yang disimpan dalam larik, dalam list yang simpul-simpulnya kebetulan berdampingan di memori, dan dalam simpul-simpul *yang sama* yang ditautkan dalam urutan acak, yang merupakan rupa list berumur panjang sesudah banyak penyisipan dan penghapusan. Setiap waktu adalah yang terbaik dari lima eksekusi.

Kode 7.16 `bench.cpp`: penyisipan depan dan penelusuran diukur waktunya untuk larik dan *linked list*.

```cpp
// bench.cpp -- array versus linked list on a real machine.
// Part 1: insert n values at the FRONT. Part 2: walk n values and add them up.
#include <algorithm>
#include <chrono>
#include <cstdio>
#include <forward_list>
#include <numeric>
#include <random>
#include <vector>

struct Node {
    long long data;
    Node *next;
};

template <typename F>
double ms(F f) {                          // best of 5 runs, in milliseconds
    double best = 1e300;
    for (int r = 0; r < 5; ++r) {
        auto t0 = std::chrono::steady_clock::now();
        f();
        auto t1 = std::chrono::steady_clock::now();
        best = std::min(best, std::chrono::duration<double, std::milli>(t1 - t0).count());
    }
    return best;
}

int main() {
    std::printf("Part 1: n insertions at the front\n");
    std::printf("%8s %14s %14s\n", "n", "vector (ms)", "list (ms)");
    for (int n : {20000, 40000, 80000, 160000}) {
        double tv = ms([&] {
            std::vector<int> v;
            for (int i = 0; i < n; ++i) v.insert(v.begin(), i);   // shifts everything
        });
        double tf = ms([&] {
            std::forward_list<int> f;
            for (int i = 0; i < n; ++i) f.push_front(i);          // relinks one pointer
        });
        std::printf("%8d %14.2f %14.2f\n", n, tv, tf);
    }

    const int n = 2000000;
    std::vector<long long> v(n);
    std::iota(v.begin(), v.end(), 0);
```

Kode 7.16 (Lanjutan) *bench.cpp: penyisipan depan dan penelusuran diukur waktunya untuk larik dan linked list.*

```
    std::vector<Node> pool(n);            // all nodes in one block of memory
    std::vector<int> order(n);
    std::iota(order.begin(), order.end(), 0);
    auto link = [&](const std::vector<int> &ord) {   // chain the nodes in this order
        for (int i = 0; i < n; ++i) {
            pool[ord[i]].data = i;
            pool[ord[i]].next = (i + 1 < n) ? &pool[ord[i + 1]] : nullptr;
        }
        return &pool[ord[0]];
    };
    auto walk = [](const Node *h) {
        long long s = 0;
        for (; h != nullptr; h = h->next) s += h->data;
        return s;
    };

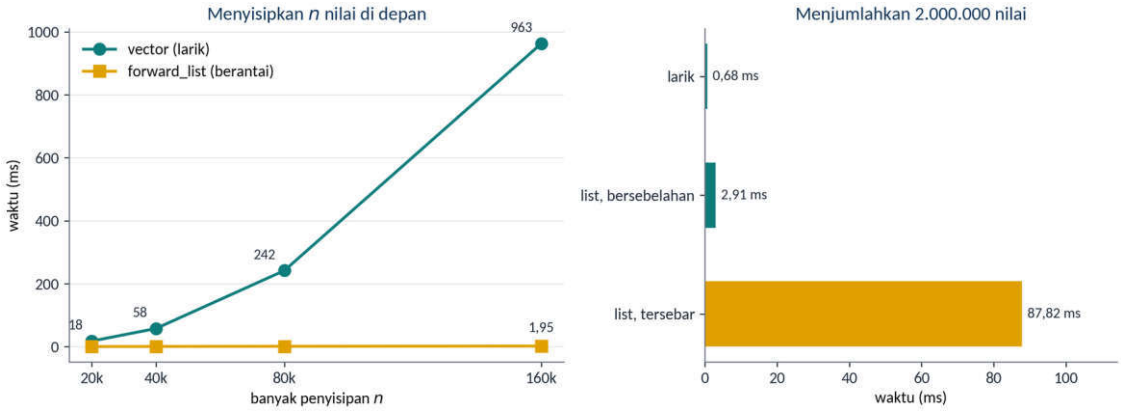
    long long s1 = 0, s2 = 0, s3 = 0;
    double ta = ms([&] { s1 = std::accumulate(v.begin(), v.end(), 0LL); });
    Node *inOrder = link(order);
    double tb = ms([&] { s2 = walk(inOrder); });
    std::shuffle(order.begin(), order.end(), std::mt19937(2026));
    Node *scattered = link(order);
    double tc = ms([&] { s3 = walk(scattered); });

    std::printf("\nPart 2: sum of %d values (all sums equal: %s)\n", n,
                (s1 == s2 && s2 == s3) ? "yes" : "no");
    std::printf("  array                        %8.2f ms\n", ta);
    std::printf("  list, nodes side by side     %8.2f ms\n", tb);
    std::printf("  list, nodes scattered        %8.2f ms\n", tc);
    return 0;
}
```

```
Part 1: n insertions at the front
       n    vector (ms)      list (ms)
   20000          17.71           0.27
   40000          57.58           0.62
   80000         241.76           1.12
  160000         962.67           1.95

Part 2: sum of 2000000 values (all sums equal: yes)
  array                          0.68 ms
  list, nodes side by side       2.91 ms
  list, nodes scattered         87.82 ms
```

Gambar 7.18 menggambarkan eksekusi itu. Bilangan persisnya bergantung pada mesin (eksekusi ini memakai server bersama Intel Xeon 2,1 GHz) dan bervariasi puluhan persen dari eksekusi ke eksekusi, tetapi bentuknya tidak.



Gambar 7.18 Hasil pengukuran **Kode 7.16**: penyisipan depan kuadratik untuk larik dan linear untuk list (kiri); menjumlahkan dua juta nilai paling cepat pada larik dan sekitar seratus kali lebih lambat pada list yang simpulnya tersebar (kanan).

- **Penyisipan depan.** Setiap penggandaan n mengalikan waktu *vector* sekitar 4 kali (18, 58, 242, 963 ms), tanda tangan $\Theta(n^2)$ dari Persamaan (7.4), dan waktu list sekitar 2 kali (0,27; 0,62; 1,12; 1,95 ms), tanda tangan $\Theta(n)$. Pada $n = 160{,}000$ list kira-kira 500 kali lebih cepat.
- **Penelusuran.** Ketiga penelusuran bernilai $\Theta(n)$, namun larik memakan 0,68 ms, list bersebelahan 2,91 ms, dan list tersebar 87,82 ms, sekitar 130 kali larik. Satu-satunya perbedaan antara dua yang terakhir adalah letak simpul-simpulnya di memori: persis efek *cache* dari Subbab 7.3.2. Notasi O, yang mengabaikan faktor konstan, tidak dapat melihatnya.

Versi Java dan Python di folder kode (`Bench.java`, `bench.py`) memperlihatkan bentuk yang sama. Dalam satu eksekusi pada mesin yang sama, `ArrayList.add(0, x)` di Java memakan 11,5; 57,9; 232; dan 989 ms untuk keempat ukuran dibandingkan paling lama 1 ms untuk `LinkedList.addFirst`, dan menelusuri dua juta simpul buatan sendiri yang diacak memakan sekitar 100 ms dibandingkan 0,7 ms untuk `long[]`. Di Python, `list.insert(0, x)` memakan 58, 234, 961, dan 4.095 ms, sedangkan `deque.appendleft` memakan 0,4 sampai 4,8 ms dan list simpul buatan sendiri 3,3 sampai 32 ms.

Pengukuran ini merangkum kearifan praktis bidang ini, yang juga telah diperagakan dengan cara yang sama oleh perancang C++ [15]: untuk sebagian besar beban kerja, larik (`std::vector`, `ArrayList`, `list` Python) lebih cepat daripada *linked list*, bahkan untuk penyisipan di tengah, karena menemukan posisinya berongkos perjalanan pada list dan perjalanan itu lambat. *Linked list* unggul ketika penyisipan dan penghapusan terjadi pada posisi yang sudah dipegang program (depan, belakang, simpul yang ditemukan sebelumnya), ketika elemen tidak boleh berpindah, ketika elemennya besar, atau ketika strukturnya sendiri berupa rantai referensi, seperti *stack*, *queue*, dan jadwal melingkar dalam bab ini.

7.13 Profitina dalam Praktik

Profitina adalah platform visibilitas AI (GEO): ia mengukur seberapa siap situs web sebuah merek untuk dikutip oleh mesin jawaban AI dan membantu memperbaiki apa yang ditemukannya [16]. Servernya adalah aplikasi Python yang dibangun dengan FastAPI, dan beberapa fiturnya memiliki bentuk yang dipelajari bab ini. Kita menguraikan bentuk setiap masalah dan golongan struktur yang cocok dengannya; pilihan implementasi yang sesungguhnya di dalam produk adalah urusan produk itu sendiri.

- **Umpan yang tumbuh di depan.** Komunitas *real-time* Profitina mengantarkan unggahan baru ke peramban yang tersambung dengan *Server-Sent Events*, dan unggahan baru langsung tampil.

Pada umpan yang ditampilkan dari yang terbaru, setiap unggahan yang datang masuk ke *atas*. Operasi yang paling sering dikerjakan umpan semacam itu tepat berupa `frontInsertion`, dan operasi yang dikerjakan pembaca adalah penelusuran dari butir terbaru. Struktur dengan penyisipan $O(1)$ di depan dan pembuangan murah atas butir-butir tertua (untuk menyimpan hanya k butir terbaru di memori) adalah pasangan yang alami; di Python itu berupa `collections.deque` dengan panjang maksimum, yang otomatis membuang dari ujung lain ketika penuh.

- **Antrean pemindaian ulang.** Fitur analitik historis memindai ulang setiap merek secara otomatis dengan irama yang bergantung pada tingkat langganannya. Pekerjaan yang sudah jatuh tempo menunggu dalam antrean masuk-pertama-keluar-pertama sampai seorang pekerja mengambilnya, *queue* berantai di Subbab 7.10: `enqueue` di belakang ketika sebuah pekerjaan jatuh tempo, `dequeue` di depan ketika seorang pekerja senggang, keduanya $O(1)$ dan tidak pernah "penuh". (Bila pekerjaan-pekerjaan memiliki tingkat kemendesakan yang berbeda, antrean prioritas yang dibangun di atas *heap* di Bab 14 menggantikan *queue* biasa.)
- **Bergiliran dengan adil.** Ketika sedikit pekerja harus berbagi banyak merek, layanan bergiliran, list melingkar di Subbab 7.9, memberi setiap merek satu giliran sebelum merek mana pun mendapat giliran kedua, sehingga merek dengan situs besar tidak dapat membuat merek lain kelaparan.
- **Riwayat yang hanya bertambah.** Tren skor mingguan sebuah merek adalah barisan yang hanya ditambah di belakang dan dibaca dari minggu tertua ke terbaru ketika *dashboard* menggambar grafiknya atau mengekspornya sebagai CSV: `backInsertion` dengan pointer *tail* dan penelusuran maju, list *head-tail* di Subbab 7.8. Menjelajah mundur dari minggu mana pun, satu minggu demi satu minggu, memerlukan tautan ke simpul sebelumnya, *double linked list* di Bab 9.

Pengukuran yang menutup Subbab 7.12 berlaku pula pada produk. Dalam Python, list simpul yang ditulis tangan jarang menjadi pilihan tercepat; `list` bawaan dan `collections.deque`, yang diimplementasikan dalam C, biasanya lebih cepat. Pelajaran bab ini tetap berlaku utuh: kenali ujung barisan mana yang disentuh operasi-operasinya, dan pilihlah struktur yang operasi murahnya tepat operasi-operasi itu.

7.14 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Memindahkan `head` untuk menyusuri list. Gelung seperti `while (head != nullptr) { ...; head = head->next; }` di dalam `show` menghilangkan seluruh list. Berjalanlah dengan `aux`; pindahkan `head` hanya bila simpul pertama berubah (Subbab 7.5).

Memakai syarat gelung yang salah. `while (aux != nullptr)` berakhir *sesudah* simpul terakhir (pakailah untuk mengunjungi setiap simpul); `while (aux->next != nullptr)` berakhir *pada* simpul terakhir (pakailah untuk menambahkan di belakang); `while (aux->next->next != nullptr)` berakhir pada simpul kedua dari belakang (pakailah untuk menghapus simpul terakhir). Memakai yang pertama di tempat yang kedua diperlukan membuat `aux` bernilai NULL, dan `aux->next` berikutnya meruntuhkan program.

Mendereferensi NULL. Setiap `p->field` mensyaratkan `p != nullptr`. List kosong dan list satu simpul adalah tempat hal ini keliru: perjalanan `backInsertion` dan `aux->next->next` milik `backDeletion` sama-sama memerlukan penjaga untuk keduanya (Subbab 7.5.3 dan 7.5.6). Dalam syarat majemuk, taruhlah uji nol lebih dulu: `aux != nullptr && aux->data != key`.

Penugasan dalam urutan yang salah. Pada `frontInsertion` dan `insertAfter`, tautkan simpul baru ke penerusnya *sebelum* mengarahkan pendahulunya ke simpul baru; kalau tidak, simpul baru menunjuk dirinya sendiri dan sisa list hilang (**Gambar 7.17**).

Menghapus sebelum memindahkan. `delete head; head = head->next;` membaca simpul yang sudah dibebaskan. Simpan, pindahkan, lalu hapus (Subbab 7.5.5).

Lupa memutus tautan terakhir. Sesudah menghapus simpul terakhir, `next` simpul terakhir yang baru harus diisi `nullptr`, dan `tail` harus berpindah. Kalau tidak, list berujung di memori yang sudah dibebaskan (**Kode 7.12**, mode `dangle`).

Melupakan pointer kedua. Pada list *head-tail*, menyisipkan ke list kosong harus mengisi `tail`, dan menghapus satu-satunya simpul harus mengembalikan `tail`. Pada *queue* berantai, `dequeue` atas simpul terakhir harus mengembalikan `rear_` (Subbab 7.10.2).

Memakai uji NULL untuk menghentikan penelusuran melingkar. Tidak ada medan `next` dalam cincin yang bernilai NULL, sehingga `while (aux != nullptr)` tidak pernah berakhir. Pakailah `do-while` yang berhenti ketika kembali ke awal, atau hitunglah simpulnya (Subbab 7.9.2).

Membocorkan list. Setiap simpul yang dibuat dengan `new` harus dihapus tepat satu kali. Panggillah `clear` atau `destroy` sebelum pointer terakhir ke list lenyap, atau lebih baik lagi, biarkan destruktor mengerjakannya, dan larang atau implementasikan penyalinan (Subbab 7.10.2).

Memercayai keluaran yang benar. Program yang membaca memori yang sudah dibebaskan dapat mencetak jawaban yang benar. Jalankan setiap program *linked list* di bawah Valgrind atau AddressSanitizer (Subbab 7.11).

Berekursi atas list yang sangat panjang. `lengthRec` dan `reverseRec` memakai satu bingkai tumpukan per simpul; untuk list berisi jutaan simpul pakailah versi iteratifnya (Subbab 7.6.2).

Memilih list untuk akses acak. Mengindeks list bernilai $\Theta(n)$ (Persamaan (7.3)), dan gelung atas indeks, `for (i = 0; i < n; i++) get(i)`, pun menjadi $\Theta(n^2)$. Bila program kebanyakan membaca berdasarkan posisi, yang diperlukannya adalah larik.

7.15 Ringkasan Bab

- ***Linked list*** menyimpan setiap elemen dalam sebuah **simpul** yang juga memegang pointer ke simpul berikutnya; list dicapai lewat pointer eksternal, `head`, dan `next` simpul terakhir bernilai NULL. Gagasannya berasal dari 1955 dan 1956 (IPL karya Newell, Shaw, dan Simon), dipakai oleh COMIT karya Yngve, dan menjadi dasar LISP.
- **Larik versus list.** Larik memberikan akses $O(1)$ berdasarkan posisi (Persamaan (7.1)) tetapi penyisipan dan penghapusan $\Theta(n)$ di depan atau di tengah (Persamaan (7.4)); list memberikan penyisipan dan penghapusan $O(1)$ pada posisi mana pun yang sudah dipegangnya, tetapi akses $\Theta(n)$ berdasarkan posisi (Persamaan (7.2), Persamaan (7.3)), satu pointer tambahan per elemen (Persamaan (7.5)), dan perilaku *cache* yang buruk bila simpulnya tersebar.
- **List ber-*head*.** `init`, `isEmpty`, `frontInsertion`, `frontDeletion`, dan `insertAfter` bernilai $O(1)$; `backInsertion`, `backDeletion`, `show`, `search`, `insertSorted`, `deleteKey`, `length`, `reverse`, dan `clear` bernilai $\Theta(n)$ pada kasus terburuk (**Tabel 7.2**, **Tabel 7.3**).

- **Dua aturan urutan.** Saat menyisipkan, tautkan simpul baru ke penerusnya sebelum menautkan pendahulunya ke simpul baru. Saat menghapus, simpan simpulnya, pindahkan setiap pointer melewatinya, dan hapus paling akhir.
- **Pembalikan** membalik setiap panah dalam satu lintasan dengan tiga pointer (waktu $\Theta(n)$, memori $O(1)$), atau secara rekursif dengan membalik sisanya lalu menggantungkan simpul pertama pada ujungnya (Persamaan (7.7), waktu dan tumpukan $\Theta(n)$).
- **Simpul kepala *dummy***, atau di C++ **pointer ke tautan**, menghapus kasus-kasus khusus untuk simpul pertama.
- **Pointer *tail*** membuat penyisipan belakang $O(1)$ tetapi tidak penghapusan belakang, yang memerlukan pendahulu simpul terakhir; hal itu menuntut *double linked list* di Bab 9.
- **List melingkar** yang dicapai lewat `last` memberikan akses $O(1)$ ke kedua ujung dengan satu pointer; penelusurannya berhenti ketika kembali ke awal, dan ia memodelkan layanan bergiliran secara alami.
- ***Stack* berantai** melakukan `push` dan `pop` di depan; ***queue* berantai** melakukan `enqueue` di *tail* dan `dequeue` di *head*; keempat operasi itu $O(1)$ dan tidak pernah melaporkan "penuh". Di C++, destruktor melepaskan simpul-simpulnya dan penyalinan harus dilarang atau diimplementasikan (*rule of three*).
- **Keamanan memori** harus diperiksa, bukan diandaikan: Valgrind dan AddressSanitizer menemukan kebocoran, pemakaian sesudah pembebasan, tautan menggantung, dan pembebasan ganda, lengkap dengan baris sumbernya.
- **Pustaka.** C++ memiliki `std::forward_list` (bertaut tunggal, `insert_after`, tanpa `size()`) dan `std::list`; Java memiliki `LinkedList` yang bertaut ganda; Python tidak memiliki *linked list* bawaan, dan `collections.deque`, *double linked list* berisi blok 64 slot, mencakup kasus-kasus pemakaiannya. Diukur pada mesin sungguhan, larik unggul dalam penelusuran hingga dua orde besaran, dan list unggul dalam penyisipan depan dengan selisih yang sama.

7.16 Soal Latihan

7.1 Mulai dari list ber-*head* yang kosong, kerjakan `frontInsertion(5)`, `backInsertion(8)`, `frontInsertion(2)`, `backInsertion(6)`, `frontDeletion()`, `backDeletion()`, `frontInsertion(9)`. Gambarlah list sesudah setiap operasi dan sebutkan list yang dihasilkan.

7.2 Untuk list berisi n simpul, sebutkan waktu jalan: (a) penyisipan di belakang dengan `head` saja; (b) penyisipan di belakang dengan `head` dan `tail`; (c) penghapusan di belakang dengan `head` dan `tail`; (d) penghapusan di depan pada list melingkar yang dicapai lewat `last`; (e) membangun list berisi n nilai dengan `n` kali pemanggilan `backInsertion` dengan `head` saja. Beri alasan setiap jawaban dalam satu kalimat, dan untuk (e) sebutkan banyaknya langkah `aux` secara tepat.

7.3 Tulislah `int countKey(const Node *head, int key)`, yang mengembalikan banyaknya simpul yang berisi `key`. Berapa waktu jalannya?

7.4 Tulislah `Node *nth(Node *head, int k)`, yang mengembalikan simpul berindeks k (dihitung dari 0), atau `nullptr` bila list berisi kurang dari $k+1$ simpul. Berapa langkah yang diperlukannya, menurut Persamaan (7.2)?

7.5 Tulislah `Node *middle(Node *head)`, yang mengembalikan simpul tengah dari list yang tidak kosong dalam *satu* lintasan dan tanpa mengetahui panjangnya. Untuk panjang genap, kembalikan yang kedua dari dua simpul tengah. (Petunjuk: gerakkan dua pointer, yang satu dua kali lebih cepat daripada yang lain.)

7.6 Sebuah galat dapat membuat list tanpa sengaja menjadi melingkar, dan kemudian setiap penelusuran di Subbab 7.5 berputar tanpa akhir. Tulislah `bool hasCycle(const Node *head)`, yang memutuskan apakah mengikuti `next` dari `head` pernah kembali ke simpul yang sudah dikunjungi, dalam waktu $O(n)$ dan memori tambahan $O(1)$, tanpa mengubah list.

7.7 Tulislah `Node *mergeSorted(Node *a, Node *b)`, yang menggabungkan dua list terurut naik menjadi satu list terurut naik dengan menautkan ulang simpul-simpulnya, tanpa membuat atau menghapus simpul apa pun. Berapa waktu jalannya untuk list sepanjang m dan n?

7.8 Tulislah `int dedupSorted(Node *head)`, yang menghapus nilai-nilai berulang dari list terurut naik sehingga setiap nilai muncul sekali, dan mengembalikan banyaknya simpul yang dihapus.

7.9 Tulislah `bool insertBefore(Node *&head, int key, int x)`, yang menyisipkan simpul baru berisi `x` tepat *sebelum* simpul pertama yang berisi `key`, dan mengembalikan `false` (tanpa mengubah apa pun) bila `key` tidak ada. Pastikan fungsi itu bekerja ketika `key` berada di simpul pertama.

7.10 Versi penyisipan depan yang lazim dijumpai di buku teks adalah

```
void frontInsertion(int newData) {
    TNode *newNode = new TNode;
    newNode->data = newData;
    newNode->next = nullptr;
    if (isEmpty()) {
        head = newNode;
        head->next = nullptr;
    } else {
        newNode->next = head;
        head = newNode;
    }
}
```

Tunjukkan bahwa cabang `if` itu tidak perlu, yaitu bahwa dua baris cabang `else` saja sudah benar untuk list kosong. Lalu jelaskan mengapa penyederhanaan yang sama *tidak* mungkin dilakukan pada `backInsertion`.

7.11 Dalam *masalah Josephus*, n orang bernomor 1 sampai n berdiri melingkar; mulai dari orang ke-1, setiap orang ke-k keluar, dan lingkarannya merapat, sampai tersisa satu orang. Pecahkan masalah itu dengan *single linked list* melingkar: cetak urutan keluarnya orang-orang dan orang yang bertahan untuk $n = 7$, $k = 3$. Berapa waktu jalannya untuk n dan k umum?

7.12 Pada `LinkedQueue` di **Kode 7.11**, lakukan dua perubahan: hapus baris `if (front_ == nullptr) rear_ = nullptr;` dari `dequeue`, dan biarkan `isEmpty` menguji `rear_ == nullptr` alih-alih `front_ == nullptr` (seperti list *head-tail* di Subbab 7.8 menguji `tail`). Uraikan barisan tiga operasi yang sesudahnya program memakai memori yang sudah dibebaskan, sebutkan galat mana dari keempat galat di Subbab 7.11.1 itu, dan sebutkan apa yang akan dilaporkan AddressSanitizer. Mengapa kelas aslinya selamat dari perubahan pertama saja, dan mengapa tetap salah untuk menghilangkan baris itu?

7.13 `reverseRec` dan `reverseIter` keduanya berjalan dalam waktu $\Theta(n)$. Jelaskan mengapa `reverseIter` lebih disukai untuk list berisi sepuluh juta simpul. Perkirakan memori tumpukan yang diperlukan `reverseRec` bila setiap bingkai pemanggilan memakan 48 bita.

7.14 Untuk setiap tugas Pustaka Ceria berikut, sebutkan apakah larik (atau `std::vector`) atau *linked list* yang lebih tepat, dan mengapa, berdasarkan ongkos-ongkos di **Tabel 7.1**: (a) katalog berisi 50.000 buku, yang dicari berdasarkan posisi sesudah diurutkan; (b) antrean pembaca yang menunggu satu-satunya meja baca, yang berubah setiap beberapa menit; (c) riwayat *undo* dari

suntingan seorang pustakawan, yang hanya pernah membatalkan suntingan terbaru; (d) daftar harian buku yang dikembalikan, yang dicetak sekali menurut urutan pengembalian pada jam tutup.

7.17 Penyelesaian Soal Latihan

7.1 Sesudah setiap operasi list-nya adalah: `frontInsertion(5)`: 5. `backInsertion(8)`: 5, 8. `frontInsertion(2)`: 2, 5, 8. `backInsertion(6)`: 2, 5, 8, 6. `frontDeletion()`: 5, 8, 6. `backDeletion()`: 5, 8. `frontInsertion(9)`: 9, 5, 8. List akhirnya adalah 9, 5, 8.

7.2 (a) $\Theta(n)$: `aux` harus berjalan dari `head` ke simpul terakhir, $n-1$ langkah. (b) $O(1)$: `tail` langsung memberikan simpul terakhir dan dua penugasan pointer menempelkan simpul baru. (c) $\Theta(n)$: simpul sebelum `tail` harus menjadi simpul terakhir yang baru, dan simpul itu hanya dapat ditemukan dengan berjalan dari `head` (Subbab 7.8.2). (d) $O(1)$: simpul pertama adalah `last->next`, dan melepasnya dari rantai mengubah satu pointer. (e) Pemanggilan ke-i (dihitung dari $i=0$) menjumpai list berisi i simpul; ia tidak melangkah bila $i=0$ (cabang list kosong) dan melangkah $i-1$ kali selainnya, sehingga totalnya $0+0+1+2+\cdots+(n-2)=\frac{(n-2)(n-1)}{2}$ langkah, yaitu $\Theta(n^2)$.

7.3 Program penyelesaiannya, **Kode 7.17**, memuat `countKey`: penelusuran yang menambah pencacah setiap kali `h->data == key`. Fungsi itu harus mengunjungi setiap simpul, karena simpul mana pun dapat berisi kuncinya, sehingga ia berjalan dalam waktu $\Theta(n)$ pada setiap kasus.

Kode 7.17 `exercises.cpp`: penyelesaian Soal Latihan 7.3 sampai 7.9 dan 7.11.

```cpp
// exercises.cpp -- solutions to the programming exercises of Chapter 7.
#include <cstdio>

struct Node {
    int data;
    Node *next;
};

Node *fromArray(const int *a, int n) {        // builds a[0] -> a[1] -> ... -> NULL
    Node *head = nullptr;
    for (int i = n - 1; i >= 0; --i) head = new Node{a[i], head};
    return head;
}

void show(const char *label, const Node *h) {
    std::printf("%-24s", label);
    for (; h != nullptr; h = h->next) std::printf("%d -> ", h->data);
    std::printf("NULL\n");
}

void destroy(Node *&head) {
    while (head != nullptr) {
        Node *del = head;
        head = head->next;
        delete del;
    }
}

// Exercise 7.3: how many nodes hold key?
int countKey(const Node *h, int key) {
    int c = 0;
    for (; h != nullptr; h = h->next) if (h->data == key) ++c;
    return c;
}

// Exercise 7.4: the node at index k (0-based), or nullptr if the list is shorter.
Node *nth(Node *h, int k) {
    while (h != nullptr && k > 0) {
        h = h->next;
        --k;
    }
    return (k == 0) ? h : nullptr;
}
```

Kode 7.17 (Lanjutan) *exercises.cpp: penyelesaian Soal Latihan 7.3 sampai 7.9 dan 7.11.*

```
// Exercise 7.5: the middle node in one pass (the second middle if the length is even).
Node *middle(Node *h) {
    Node *slow = h, *fast = h;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;              // one step
        fast = fast->next->next;        // two steps
    }
    return slow;
}

// Exercise 7.6: does following next ever come back to a node already visited?
bool hasCycle(const Node *h) {
    const Node *slow = h, *fast = h;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) return true;  // the fast runner lapped the slow one
    }
    return false;                       // the fast runner fell off the end
}

// Exercise 7.7: merge two ascending lists by relinking; no node is created or freed.
Node *mergeSorted(Node *a, Node *b) {
    Node dummy{0, nullptr};             // a dummy node on the stack
    Node *tail = &dummy;
    while (a != nullptr && b != nullptr) {
        Node *&smaller = (a->data <= b->data) ? a : b;
        tail->next = smaller;           // append the smaller front node
        tail = smaller;
        smaller = smaller->next;        // advance a or b
    }
    tail->next = (a != nullptr) ? a : b;   // whatever is left is already sorted
    return dummy.next;
}

// Exercise 7.8: remove repeated values from an ascending list.
int dedupSorted(Node *h) {
    int removed = 0;
    while (h != nullptr && h->next != nullptr) {
        if (h->next->data == h->data) {
            Node *del = h->next;
            h->next = del->next;
            delete del;
            ++removed;
        } else {
            h = h->next;
```

Kode 7.17 (Lanjutan) *exercises.cpp: penyelesaian Soal Latihan 7.3 sampai 7.9 dan 7.11.*

```
        }
    }
    return removed;
}

// Exercise 7.9: insert x before the first node holding key; false if key is absent.
bool insertBefore(Node *&head, int key, int x) {
    Node **link = &head;
    while (*link != nullptr && (*link)->data != key) link = &(*link)->next;
    if (*link == nullptr) return false;
    *link = new Node{x, *link};
    return true;
}

// Exercise 7.11: Josephus with a circular list. n people, every k-th leaves.
int josephus(int n, int k) {
    Node *last = nullptr;
    for (int i = 1; i <= n; ++i) {      // build the circle 1, 2, ..., n
        Node *p = new Node{i, nullptr};
        if (last == nullptr) p->next = p;
        else { p->next = last->next; last->next = p; }
        last = p;
    }
    std::printf("Josephus n=%d k=%d, leaving order:", n, k);
    Node *prev = last;
    while (prev->next != prev) {        // more than one person left
        for (int s = 1; s < k; ++s) prev = prev->next;
        Node *del = prev->next;         // the k-th person
        std::printf(" %d", del->data);
        prev->next = del->next;
        delete del;
    }
    int survivor = prev->data;
    delete prev;
    std::printf("\n");
    return survivor;
}
```

Kode 7.17 (Lanjutan) *exercises.cpp: penyelesaian Soal Latihan 7.3 sampai 7.9 dan 7.11.*

```
int main() {
    int a[] = {4, 7, 4, 1, 4, 9};
    Node *h = fromArray(a, 6);
    show("list:", h);
    std::printf("countKey(4) = %d, countKey(5) = %d\n", countKey(h, 4), countKey(h, 5));
    Node *p = nth(h, 3);
    std::printf("nth(3) = %d, nth(6) is %s\n", p->data, nth(h, 6) ? "a node" : "nullptr");
    std::printf("middle = %d\n", middle(h)->data);

    Node *last = nth(h, 5);
    last->next = nth(h, 2);             // make a cycle on purpose
    std::printf("hasCycle after linking 9 back to 4: %s\n", hasCycle(h) ? "true" :
"false");
    last->next = nullptr;               // undo it before anything walks the list
    std::printf("hasCycle after undoing it:          %s\n", hasCycle(h) ? "true" :
"false");
    destroy(h);

    int x[] = {1, 4, 4, 9}, y[] = {2, 4, 10};
    Node *m = mergeSorted(fromArray(x, 4), fromArray(y, 3));
    show("mergeSorted:", m);
    int r = dedupSorted(m);
    std::printf("dedupSorted removed %d\n", r);
    show("after dedup:", m);
    insertBefore(m, 1, 0);              // before the first node
    insertBefore(m, 9, 7);              // in the middle
    bool ok = insertBefore(m, 99, 5);   // key absent
    show("insertBefore x3:", m);
    std::printf("insertBefore(99, 5) returned %s\n", ok ? "true" : "false");
    destroy(m);

    std::printf("survivor = %d\n", josephus(7, 3));
    return 0;
}
```

```
list:                  4 -> 7 -> 4 -> 1 -> 4 -> 9 -> NULL
countKey(4) = 3, countKey(5) = 0
nth(3) = 1, nth(6) is nullptr
middle = 1
hasCycle after linking 9 back to 4: true
hasCycle after undoing it:          false
mergeSorted:           1 -> 2 -> 4 -> 4 -> 4 -> 9 -> 10 -> NULL
dedupSorted removed 2
after dedup:           1 -> 2 -> 4 -> 9 -> 10 -> NULL
insertBefore x3:       0 -> 1 -> 2 -> 4 -> 7 -> 9 -> 10 -> NULL
insertBefore(99, 5) returned false
Josephus n=7 k=3, leaving order: 3 6 2 7 5 1
survivor = 4
```

Pada list 4, 7, 4, 1, 4, 9 keluarannya memastikan bahwa 4 muncul tiga kali dan 5 tidak pernah muncul.

7.4 `nth` di **Kode 7.17** memajukan `h` dan mengurangi `k` bersamaan sampai `k` mencapai 0 atau list berakhir. Bila `k` mencapai 0, `h` adalah simpul pada indeks yang diminta; ketika k sama dengan panjang list, "simpul" itu adalah NULL sesudah simpul terakhir, sehingga fungsi dengan benar mengembalikan `nullptr`. Bila list berakhir lebih dulu, `k` masih positif dan fungsi mengembalikan `nullptr`. Menurut Persamaan (7.2) fungsi itu melangkah $\min(k, n)$ kali, sehingga bernilai $O(k)$ dan $\Theta(n)$ pada kasus terburuk. Pada keluarannya, indeks 3 berisi 1 dan indeks 6 tidak ada dalam list berisi 6 simpul.

7.5 `middle` memajukan `slow` satu simpul dan `fast` dua simpul per iterasi, dan berhenti ketika `fast` tidak dapat maju dua langkah lagi. Ketika `fast` sudah melangkah $2t$ kali, `slow` sudah

melangkah t kali, sehingga ketika `fast` mencapai ujung, `slow` berada di tengah jalan. Untuk list enam simpul 4, 7, 4, 1, 4, 9 gelungnya berjalan tiga kali (`fast` mengunjungi indeks 2, 4, lalu NULL), sehingga `slow` berhenti pada indeks 3, yang berisi 1: yang kedua dari dua simpul tengah, sesuai permintaan. Algoritma ini melakukan sekitar $n/2$ iterasi dan memakai dua pointer, satu lintasan, dan memori $O(1)$.

7.6 `hasCycle` adalah algoritma "kura-kura dan kelinci" (*tortoise and hare*) karya Floyd, gagasan Soal Latihan 7.5 yang diterapkan pada list yang mungkin melingkar. Bila list tidak memiliki siklus, `fast` mencapai NULL dan fungsi mengembalikan `false`. Bila list memiliki siklus, kedua pointer akhirnya masuk ke dalamnya, dan di dalam siklus jarak dari `slow` ke `fast`, diukur searah tautan, menyusut tepat satu simpul per iterasi (`fast` mendahului `slow` satu langkah), sehingga jarak itu mencapai 0 dan kedua pointer bertemu sesudah paling banyak satu putaran siklus. Kerja totalnya $O(n)$, dan memori tambahannya dua pointer. Program sengaja membuat siklus dengan menautkan simpul terakhir (9) kembali ke simpul berindeks 2, memastikan bahwa `hasCycle` mengembalikan `true`, memulihkan tautan NULL sebelum ada penelusuran yang dapat berputar tanpa akhir, lalu memastikan `false`.

7.7 `mergeSorted` memakai simpul *dummy* di *stack* (sebuah variabel lokal, sehingga tidak memerlukan `delete`) sebagai awal hasil, dan pointer `tail` ke simpul terakhir hasil. Pada setiap langkah ia menempelkan yang lebih kecil dari kedua simpul depan dan maju pada list asalnya. Referensi `Node *&smaller` merujuk ke `a` atau ke `b` itu sendiri, sehingga `smaller = smaller->next` memajukan list yang benar. Ketika salah satu list habis, sisa list yang lain sudah terurut dan ditempelkan dengan satu penugasan. Setiap iterasi menangani satu simpul, sehingga waktunya $\Theta(m+n)$ pada kasus terburuk, dan tidak ada memori yang dialokasikan. Menggabungkan 1, 4, 4, 9 dengan 2, 4, 10 menghasilkan 1, 2, 4, 4, 4, 9, 10. Karena pembandingannya `<=`, nilai-nilai sama dari `a` mendahului nilai dari `b`, sehingga penggabungannya stabil. Inilah langkah penggabungan dalam *merge sort*, yang menjadi cara `std::forward_list::sort` mengurutkan dalam waktu $O(n\lg n)$ (Subbab 7.12.1).

7.8 Pada list terurut naik, nilai-nilai yang sama bersebelahan, sehingga `dedupSorted` membandingkan setiap simpul dengan penerusnya: bila sama, ia menghapus penerusnya (simpan, tautkan ulang, hapus) dan tetap di tempat, karena penerus yang baru mungkin juga sama; selainnya ia maju. List 1, 2, 4, 4, 4, 9, 10 kehilangan dua simpul dan menjadi 1, 2, 4, 9, 10, dalam waktu $\Theta(n)$.

7.9 Simpul sebelum titik penyisipan adalah simpul yang tautannya harus berubah, sehingga `insertBefore` menjalankan *pointer ke tautan*, teknik Subbab 7.7.3: `link` dimulai pada `&head` dan pindah ke `&(*link)->next` sampai `*link` adalah simpul yang berisi `key`. Lalu `*link = new Node{x, *link}` menyisipkan simpul baru di depannya, dan baris tunggal ini menugasi `head` ketika kuncinya berada di simpul pertama. Pada keluarannya, 0 disisipkan sebelum 1 (simpul pertama, sehingga `head` berubah), 7 sebelum 9, dan pemanggilan untuk kunci 99 yang tidak ada mengembalikan `false` tanpa mengubah apa pun. Dengan pointer pengekor `prev`, simpul pertama memerlukan cabang tersendiri; versi Java dan Python-nya (`Exercises.java`, `exercises.py`) memakai simpul *dummy* sementara sebagai gantinya.

7.10 Ketika list kosong, `head` bernilai `nullptr`. Cabang `else` menjalankan `newNode->next = head`, yang mengisi `newNode->next` dengan `nullptr`, dan `head = newNode`, yang menjadikan simpul baru simpul pertama. Hasilnya list satu simpul yang `next`-nya `nullptr`, persis yang dihasilkan cabang `if`, sehingga cabang itu dapat dihapus (bersama `newNode->next = nullptr` yang kini berlebihan). Itulah sebabnya **Kode 7.3** tidak memiliki `if` di dalam `frontInsertion`.

Penyederhanaan itu berhasil karena penyisipan depan tidak pernah *membaca* medan simpul yang sudah ada; ia hanya menyalin nilai `head`, yang terdefinisi dengan baik bahkan ketika bernilai nol. `backInsertion` harus menjalankan `aux->next` pada simpul terakhir, dan pada list kosong tidak ada simpul yang dapat ditunjuk `aux`, sehingga kasus kosong harus ditangani tersendiri (atau dihilangkan dengan simpul *dummy*, Subbab 7.7).

7.11 `josephus` di **Kode 7.17** membangun lingkaran 1 sampai 7 dengan penyisipan belakang lewat `last`, dan menjaga `prev` pada simpul sebelum orang berikutnya yang dihitung. Setiap putaran memajukan `prev` sebanyak $k-1$ simpul, menghapus `prev->next` (orang ke-k), dan merapatkan lingkaran. Untuk $n=7$, $k=3$ urutan keluarnya 3, 6, 2, 7, 5, 1 dan yang bertahan adalah 4, seperti diperlihatkan keluarannya. Periksalah putaran-putaran pertama dengan tangan: menghitung 1, 2, **3** mengeluarkan 3; menghitung 4, 5, **6** mengeluarkan 6; menghitung 7, 1, **2** mengeluarkan 2. Ada $n-1$ putaran masing-masing $k-1$ langkah, sehingga waktunya $\Theta(nk)$ ditambah $\Theta(n)$ untuk membangun lingkarannya.

7.12 Perhatikan `enqueue(A)`, `dequeue()`, `enqueue(B)`. Tanpa baris yang dihapus itu, `dequeue` meninggalkan `front_` bernilai `nullptr` tetapi `rear_` masih memegang alamat simpul A, yang baru saja dibebaskannya: sebuah pointer basi. Dengan `isEmpty` yang menguji `rear_`, *queue* kini mengaku *tidak* kosong, sehingga `enqueue(B)` mengambil cabang `else` dan menjalankan `rear_->next = n`, menulis ke simpul A yang sudah dibebaskan, dan `front_` tidak pernah diisi, sehingga B pun hilang. Inilah galat ke-2 di Subbab 7.11.1 (pemakaian sesudah pembebasan lewat pointer basi; pointernya berada di luar rantai, sehingga bukan tautan menggantung di dalam list). AddressSanitizer akan berhenti pada baris `else rear_->next = n;` dengan `heap-use-after-free` dan `WRITE of size 8` (medan `next` 8 bita, yang menyusul `Book` di dalam simpul), diikuti jejak tumpukan `delete` di dalam `dequeue` dan `new` di dalam `enqueue(A)`. Kelas aslinya selamat dari perubahan pertama saja hanya karena kebetulan: `isEmpty` dan `enqueue`-nya tidak pernah membaca `rear_` selama `front_` bernilai NULL. Tambahan apa pun di kemudian hari yang membacanya (pengakses `back()`, `size_` yang dihitung dengan cara lain, `isEmpty` yang ditulis ulang seperti di atas) akan mengubah pointer basi itu menjadi galat. Aturan yang aman adalah invarian *head-tail* di Subbab 7.8.1: setiap kali struktur menjadi kosong, *kedua* pointer menjadi NULL.

7.13 `reverseRec` melakukan satu pemanggilan bersarang per simpul, sehingga untuk $n=10^7$ simpul *call stack* mencapai kedalaman sepuluh juta bingkai. Dengan 48 bita per bingkai itu berarti 4.8×10^8 bita, sekitar 480 MB, jauh melampaui batas tumpukan lazim 8 MB di Linux (1 MB secara bawaan untuk *thread* utama di Windows), sehingga program akan runtuh karena luapan tumpukan jauh sebelum selesai. `reverseIter` memakai tiga pointer berapa pun n-nya. Keduanya memakan waktu $\Theta(n)$; hanya yang iteratif yang berjalan dengan memori $O(1)$.

7.14 (a) Larik: sesudah diurutkan, pencarian berdasarkan posisi (dan pencarian biner berdasarkan nomor buku) memerlukan akses acak, $O(1)$ per akses; list akan memerlukan $\Theta(n)$ langkah per pencarian. (b) *Queue* berantai (atau larik melingkar): pembaca bergabung di belakang dan pergi di depan, keduanya $O(1)$, dan banyaknya yang menunggu tidak dapat diduga. (c) *Stack*; *stack* berantai maupun `std::vector` yang dipakai sebagai *stack* sama-sama berfungsi, karena hanya puncaknya yang disentuh. Karena suntingan berupa rekaman kecil, *vector* lebih hemat, sedangkan *stack* berantai tidak pernah perlu tumbuh ulang. (d) Keduanya berfungsi, karena daftarnya hanya ditambah di belakang dan ditelusuri sekali; `std::vector` paling sederhana dan paling ramah *cache*, dan list *head-tail* sama benarnya dengan penambahan belakang $O(1)$.

Daftar Pustaka

[1] A. Newell and J. C. Shaw, “Programming the Logic Theory Machine,” in *Proc. Western Joint Computer Conference*, Los Angeles, CA, USA, Feb. 1957, pp. 230–240.

[2] A. Newell and F. M. Tonge, “An introduction to Information Processing Language V,” *Communications of the ACM*, vol. 3, no. 4, pp. 205–211, Apr. 1960.

[3] V. H. Yngve, “A programming language for mechanical translation,” *Mechanical Translation*, vol. 5, no. 1, pp. 25–41, Jul. 1958.

[4] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, Part I,” *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, Apr. 1960.

[5] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, sec. 2.2.

[6] U. Drepper, “What every programmer should know about memory,” Red Hat, Inc., Tech. Rep., Nov. 2007. [Online]. Available: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

[7] L. Torvalds, “The mind behind Linux,” interview by C. Anderson, TED2016, Vancouver, BC, Canada, Feb. 2016. [Online]. Available: https://www.ted.com/talks/linus_torvalds_the_mind_behind_linux

[8] Valgrind Developers, “Valgrind: Current releases.” [Online]. Available: https://valgrind.org/downloads/current.html. [Accessed: 24-Sep-2026].

[9] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in *Proc. 2012 USENIX Annual Technical Conference*, Boston, MA, USA, Jun. 2012, pp. 309–318.

[10] ISO/IEC 14882:2017, *Programming Languages — C++*, 5th ed. Geneva, Switzerland: ISO, 2017, sec. 26.3 (sequence containers `forward_list` and `list`).

[11] GNU Project, “GCC 16 release series.” [Online]. Available: https://gcc.gnu.org/gcc-16/. [Accessed: 24-Sep-2026].

[12] Oracle, “Class LinkedList,” *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/LinkedList.html. [Accessed: 24-Sep-2026].

[13] Python Software Foundation, “collections — Container datatypes: deque objects,” *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/collections.html. [Accessed: 24-Sep-2026].

[14] Python Software Foundation, “Modules/_collectionsmodule.c,” CPython source repository. [Online]. Available: https://github.com/python/cpython/blob/main/Modules/_collectionsmodule.c. [Accessed: 24-Sep-2026].

[15] B. Stroustrup, “C++11 style,” keynote, GoingNative 2012, Redmond, WA, USA, Feb. 2012.

[16] Profitina, “Profitina — AI Visibility (GEO) platform.” [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

Bab 8

Evaluasi dan Pembahasan II: Stack, Pointer, dan Linked List dalam Penerapan

Bab ini tidak menambahkan struktur baru. Bab ini mengambil tiga alat dari Bab 5, 6, dan 7, yaitu *stack* (tumpukan) dan *queue* (antrean), *pointer* (penunjuk) beserta fungsi yang menerimanya, serta *singly linked list*, lalu memakainya untuk dua belas soal. Setiap soal diselesaikan dari halaman kosong, selangkah demi selangkah, dalam C++, Java, dan Python. Lima soal membentuk satu rangkaian: melengkapi kelas stack, menelusuri list dua arah menurut petunjuk, mengurutkan bilangan menjadi genap dan ganjil hanya dengan linked list, membalik urutan kata sebuah kalimat, dan membuang nilai ganda dari sebuah list. Tujuh soal lainnya baru, dan masing-masing membidik satu gagasan yang sering licin bagi pemula: mencocokkan tanda kurung, queue yang berputar kembali ke awal lariknya, mengevaluasi ekspresi aritmetika, tiga cara sebuah fungsi menerima argumennya, membalik list di tempat, menggabungkan dua list terurut, dan queue yang dibangun dari simpul bertaut. Untuk setiap soal, Anda akan menggambar pointer-pointernya sebelum menulis kode, menelusuri program dengan tangan, membaca tiga program lengkap beserta keluaran yang benar-benar dicetaknya, mempelajari satu penyelesaian keliru yang menggoda dan melihat persis bagaimana ia gagal (untuk *bug* memori, kegagalan itu dilaporkan oleh pemeriksa memori, dan Anda akan membaca laporannya), lalu menutup dengan varian *uji diri* beserta jawabannya. Di akhir bab, Anda akan memiliki sebuah metode untuk soal apa pun yang berisi kotak-kotak yang menunjuk ke kotak lain, dan Anda sudah memakainya dua belas kali.

8.1 Titik Tolak Kita

Tiga bab sudah kita lalui, dan masing-masing memberi kita satu alat. Ada baiknya alat-alat itu kita sebut satu per satu sebelum dipakai, sebab separuh dari menyelesaikan soal adalah mengenali alat mana yang diminta soal itu.

Bab 5 memberi kita dua wadah yang memiliki aturan tentang ujung-ujungnya. *Stack* hanya mengizinkan penambahan dan pengambilan butir di satu ujung, yaitu *top* (puncak); butir yang terakhir masuk adalah yang pertama keluar, karena itu stack disebut LIFO (*last in, first out*). Operasinya adalah `push`, `pop`, `peek`, `isEmpty`, `isFull`, dan `clear`. Kita membangunnya di atas larik dengan satu bilangan bulat, `top`, yang mengingat letak butir teratas, dan kita memakainya untuk kalkulator ilmiah: mengubah ekspresi seperti `3+2*5` menjadi bentuk *postfiks*, `325*+`, lalu mengevaluasi bentuk postfiks itu. *Queue* menambah di satu ujung, *rear* (belakang), dan mengambil di ujung lain, *front* (depan); yang pertama masuk adalah yang pertama keluar, FIFO. Kita juga membangunnya di atas larik, bertemu dengan masalah "penuh semu" pada queue yang ujung belakangnya hanya bergerak ke kanan, lalu memperbaikinya dengan larik *sirkular* yang indeksnya berputar kembali ke awal.

Bab 6 menjelaskan apa itu *pointer*: variabel yang nilainya adalah alamat variabel lain. Kita berkenalan dengan operator alamat `&`, operator dereferensi `*` yang mengikuti sebuah alamat ke

variabel yang tinggal di sana, operator panah `->` yang melakukan hal yang sama untuk anggota sebuah `struct`, pointer nol `nullptr` yang tidak menunjuk ke mana pun, dan aritmetika pointer pada larik. Yang terpenting, Bab 6 membandingkan tiga cara sebuah fungsi C++ menerima argumen, yakni *by value*, *by reference*, dan *by pointer*, serta menunjukkan cara mana yang memungkinkan fungsi mengubah variabel milik pemanggilnya.

Bab 7 memakai pointer untuk membangun *singly linked list*: rantai *simpul*, masing-masing berisi data dan pointer `next` ke simpul berikutnya, dengan `next` pada simpul terakhir bernilai `nullptr`. Kita menulis list itu mula-mula hanya dengan pointer `head`, lalu dengan `head` dan `tail`, dan kita menulis operasinya satu per satu: `init`, `isEmpty`, `frontInsertion`, `backInsertion`, `show`, `frontDeletion`, `backDeletion`, dan `clear`. Setiap operasi disertai gambar kotak dan panah, dan setiap operasi memiliki kasus list kosong yang memerlukan baris kodenya sendiri.

Bab ini memakai ketiga alat itu, sering kali di dalam program yang sama. **Tabel 8.1** mendaftar kedua belas soal, subbab tempat masing-masing diselesaikan, dan bab yang menyumbangkan gagasan utamanya.

Tabel 8.1 Dua belas soal dalam bab ini, subbab yang menyelesaikan masing-masing, dan bab yang menyumbangkan gagasan utamanya.

Soal	Subbab	Yang diminta	Alat utama dari
8.1	8.3	Melengkapi kelas `Stack` berbasis larik	Bab 5
8.2	8.4	Menelusuri linked list dua arah menurut petunjuk yang tersimpan di simpulnya	Bab 6, 7 (dan sekilas Bab 9)
8.3	8.5	Mencetak bilangan genap secara naik, lalu bilangan ganjil secara turun, hanya dengan linked list	Bab 6, 7
8.4	8.6	Membalik urutan kata sebuah kalimat dengan linked list	Bab 5, 7
8.5	8.7	Membuang simpul yang nilainya sudah pernah muncul	Bab 7
8.6	8.8	Memeriksa apakah tanda kurung sebuah ekspresi berpasangan	Bab 5
8.7	8.8	Queue sirkular untuk permintaan peminjaman	Bab 5
8.8	8.8	Infiks ke postfiks, lalu evaluasi	Bab 5
8.9	8.9	By value, by reference, dan by pointer, ditelusuri	Bab 6
8.10	8.10	Membalik linked list di tempat	Bab 6, 7
8.11	8.10	Menggabungkan dua linked list terurut	Bab 7
8.12	8.10	Queue yang dibangun dari linked list dengan `head` dan `tail`	Bab 5, 7

Sepanjang bab ini kita tetap memakai contoh berjalan buku ini, yaitu katalog perpustakaan kecil *Pustaka Ceria*, yang bukunya memiliki kode seperti `B1001` dan yang anggotanya mengantre untuk judul-judul yang populer. Soal tentang tumpukan piring dan kalimat memakai datanya sendiri yang kecil, sebab data dalam pernyataan soal aslinya justru bagian dari apa yang membuatnya menjadi latihan yang baik.

> **WAWASAN KUNCI: SATU GAGASAN DI BALIK HAMPIR SETIAP SOAL DI SINI.** Hampir setiap soal dalam bab ini adalah aturan yang sama dengan pakaian berbeda: *sebelum mengubah sebuah tautan, pastikan Anda masih memiliki jalan menuju semua yang akan Anda perlukan sesudah perubahan itu.* Pada stack, bacalah butir teratas sebelum `top` digeser darinya. Pada penelusuran Soal 8.2, simpanlah banyaknya langkah sebelum pointer meninggalkan simpul yang memuatnya. Pada pembalikan list, simpanlah `cur->next` sebelum menimpanya. Pada penghapusan, tautkan pendahulu melompati simpul itu sebelum membebaskannya. Pada queue, setel ulang `tail` ketika simpul terakhir hilang. Bila sebuah program dalam bab ini

> keliru, hampir selalu penyebabnya adalah aturan ini dilanggar, dan penyelesaian-penyelesaian keliru di bawah melanggarnya dengan semua cara tersebut.

8.2 Cara Menghadapi Soal Pointer

Soal tentang stack, queue, atau linked list sesungguhnya adalah soal tentang *kotak-kotak yang menunjuk ke kotak lain*. Kodenya pendek. Kesulitannya terletak pada menjaga catatan kotak mana menunjuk ke mana pada setiap saat, dan kertas melakukannya lebih baik daripada ingatan. Subbab ini memberi sebuah metode dalam tujuh langkah. Pada soal kecil, ketujuhnya memakan lima menit; pada soal sulit, ketujuhnya membedakan program yang berjalan benar dari program yang hanya benar untuk contohnya.

Langkah 1 — Nyatakan ulang soal dengan kata-kata Anda sendiri, disertai satu contoh. Tuliskan apa yang masuk, apa yang keluar, dan satu masukan kecil beserta keluarannya yang dikerjakan dengan tangan. Jika Anda belum dapat mengerjakan contoh itu dengan tangan, Anda belum siap menulis kode, karena Anda belum tahu apa yang harus dikerjakan kode itu.

Langkah 2 — Pilih strukturnya, dan sebutkan alasannya. “Terakhir masuk, pertama keluar” berarti stack. “Siapa datang lebih dulu, dilayani lebih dulu” berarti queue. “Menyisipkan dan menghapus di tengah tanpa menggeser” berarti linked list. “Melompat maju dan mundur” berarti dua tautan per simpul. Tuliskan alasannya dalam satu kalimat; alasan itu akan memberi tahu operasi mana yang Anda perlukan.

Langkah 3 — Gambarlah kotak-kotaknya. Gambarlah setiap simpul sebagai persegi panjang yang terbagi menjadi bagian data dan bagian pointer, gambarlah setiap pointer sebagai panah yang berpangkal pada sebuah titik di dalam kotak tempat pointer itu tinggal, dan tuliskan setiap variabel pointer dalam program Anda (`head`, `tail`, `cur`, `prev`, `top`) sebagai label dengan panah ke kotak yang ditunjuknya. Gambarlah `nullptr` secara eksplisit. Inilah kebiasaan paling berguna dalam seluruh mata pelajaran ini [1], [2].

Langkah 4 — Tuliskan invariannya. *Invarian* adalah kalimat yang selalu benar setiap kali program melewati suatu titik tertentu. Untuk stack: “butir-butirnya adalah `a[0]` sampai `a[top]`, dan `top` bernilai -1 ketika stack kosong”. Untuk list dengan tail: “`tail` menunjuk ke simpul terakhir, atau bernilai `nullptr` ketika list kosong”. Setiap operasi harus membiarkan invarian tetap benar, dan memeriksa hal itu adalah cara Anda mengetahui bahwa operasinya benar.

Langkah 5 — Daftarlah kasus-kasus tepi sebelum menulis kode. Untuk struktur bertaut, daftarnya hampir selalu sama: struktur kosong, satu simpul, simpul pertama, simpul terakhir, dan dua nilai sama yang bersebelahan. Untuk larik: larik kosong, larik penuh, dan indeks yang berputar kembali ke awal.

Langkah 6 — Ubahlah pointer-pointer dalam urutan yang aman. Untuk setiap operasi, beri nomor pada setiap penugasan pointer di gambar Anda, dan sebelum masing-masing, periksa bahwa tidak ada yang masih Anda perlukan yang menjadi tak terjangkau. Kotak wawasan di Subbab 8.1 adalah langkah ini.

Langkah 7 — Ujilah, termasuk di bawah pemeriksa memori. Jalankan contoh Langkah 1, lalu setiap kasus tepi Langkah 5. Di C++, kompilasi sekali lagi dengan *sanitizer* dinyalakan dan jalankan kembali; *bug* memori yang kebetulan mencetak jawaban benar tetaplah *bug*, dan *sanitizer* akan mengatakannya.

Tabel 8.2 memadatkan metode ini ke dalam satu halaman. Setiap soal di bawah menerapkannya, dan label bercetak tebal di dalam setiap soal (**Dengan kata-kata kita sendiri**, **Dengan tangan**, **Penyelesaian C++**, dan seterusnya) mengikuti urutannya.

Tabel 8.2 Metode tujuh langkah untuk soal tentang stack, queue, dan linked list. Langkah 1 dan 2 memahami soal, Langkah 3 sampai 6 merancang pergerakan pointer, dan Langkah 7 memeriksanya.

Langkah	Tuliskan di kertas	Pertanyaan yang dijawabnya
1	Soal dengan kata-kata sendiri, dan satu contoh yang dikerjakan dengan tangan	Apakah saya tahu keluaran yang seharusnya?
2	Strukturnya, dan satu kalimat alasan	Stack, queue, larik, satu tautan atau dua?
3	Kotak, panah, dan setiap variabel pointer	Siapa menunjuk ke mana saat ini?
4	Invarian	Apa yang harus tetap benar sesudah setiap operasi?
5	Kasus-kasus tepi	Kosong, satu simpul, pertama, terakhir, tetangga yang sama?
6	Penugasan pointer, diberi nomor	Adakah yang hilang jika saya melakukannya dalam urutan ini?
7	Uji, lalu uji yang sama di bawah pemeriksa memori	Apakah ia bekerja, atau hanya *tampak* bekerja?

Bagaimana program-program dalam bab ini diperiksa. Setiap program di bawah lengkap: Anda dapat menyalinnya ke sebuah berkas, mengompilasinya, dan menjalankannya. Program C++ dikompilasi dengan GCC dalam mode C++17 dengan semua peringatan yang lazim dinyalakan, dan tidak satu pun menghasilkan peringatan. Masing-masing lalu dikompilasi untuk kedua kalinya dengan AddressSanitizer dan UndefinedBehaviorSanitizer [3], [4], dua pemeriksa bawaan GCC dan Clang yang mengawasi setiap akses memori selama program berjalan, dan dijalankan di bawah Valgrind [5], yang melakukan pekerjaan serupa tanpa kompilasi ulang; setiap jalannya bersih, tanpa akses yang tidak sah dan tanpa satu bait pun yang bocor. Perintahnya sebagai berikut (di Windows, berkas eksekusinya `p801.exe` dan dijalankan sebagai `p801`):

```
g++ -std=c++17 -O2 -Wall -Wextra -o p801 p801_stack.cpp        build biasa
./p801
g++ -std=c++17 -g -Wall -Wextra -fsanitize=address,undefined -o p801_asan p801_stack.cpp
./p801_asan                                                     keluaran sama, plus
laporan jika memori disalahgunakan
valgrind --leak-check=full ./p801                               Linux: pemeriksaan sama
tanpa kompilasi ulang
./p802 < ../input/p802.txt                                      program yang membaca
masukan, diumpan dari berkas
```

Program Java dikompilasi dengan `javac` dan dijalankan dengan `java`, dan program Python dengan `python3`; program-program itu ditulis untuk JDK 25 dan Python 3.14, tetapi tidak memakai apa pun yang lebih baru daripada JDK 21 dan Python 3.10, sehingga instalasi yang lebih lama pun dapat menjalankannya. Untuk setiap soal, ketiga program mencetak baris-baris yang persis sama, dan hal itu diperiksa dengan membandingkan keluarannya bait demi bait. Satu-satunya pengecualian adalah Soal 8.9, yang pokok bahasannya justru hal yang membedakan ketiga bahasa itu. Folder kode bab ini, yang dijelaskan dalam `README.md`-nya, memuat setiap program, setiap berkas masukan, dan penyelesaian-penyelesaian keliru yang dipakai untuk peragaan.

Satu simpul, tiga bahasa. C++ memperlihatkan pointer kepada kita; Java dan Python menyembunyikannya, tetapi kotak dan panahnya tetap ada. **Tabel 8.3** adalah kamus kecil yang memungkinkan Anda membaca ketiga versi setiap program secara berdampingan.

Tabel 8.3 Gagasan yang sama dalam tiga bahasa buku ini. Referensi Java dan nama Python berperilaku seperti pointer C++ yang hanya dapat diikuti, tidak pernah dinaikkan, dan yang sasarannya dibebaskan secara otomatis.

Gagasan	C++	Java	Python
Tipe simpul	`struct Node { int data; Node *next; };`	`class Node { int data; Node next; }`	`class Node:` dengan `self.data`, `self.next`
Variabel yang merujuk ke simpul	`Node *p` (pointer)	`Node p` (referensi)	`p` (nama)
"Tidak menunjuk ke mana pun"	`nullptr`	`null`	`None`
Membuat simpul	`new Node{5, nullptr}`	`new Node(5, null)`	`Node(5)`
Mengikuti tautan	`p->next`	`p.next`	`p.next`
Menghancurkan simpul	`delete p;` (tugas Anda)	otomatis (*garbage collector*)	otomatis (penghitungan referensi dan *collector*)
Membiarkan fungsi mengubah pointer milik pemanggil	`Node *&head` (referensi ke pointer)	kembalikan nilai barunya	kembalikan nilai barunya
Stack / queue pustaka	`std::stack`, `std::queue`	`ArrayDeque`	`list`, `collections.deque`

Dua baris **Tabel 8.3** layak mendapat satu kalimat masing-masing. Pertama, hanya C++ yang mewajibkan *Anda* menghancurkan simpul; karena itulah hanya program C++ yang dapat membocorkan memori atau memakai simpul sesudah membebaskannya, dan karena itu pula hanya program C++ yang dijalankan di bawah pemeriksa memori. Kedua, Java dan Python tidak menyediakan cara bagi sebuah fungsi untuk mengubah variabel milik pemanggil itu sendiri, sehingga di tempat versi C++ menerima `Node *&head`, versi Java dan Python *mengembalikan* simpul pertama yang baru dan pemanggil menulis `head = insertSorted(head, ...)`. Soal 8.9 menjelaskan perbedaan ini secara tuntas [6], [7], [8].

8.3 Soal 8.1 — Melengkapi Stack Larik

8.3.1 Soalnya

> **SOAL.** Complete the program below in C++. Implement the following functions: (a) `pop()`, which pops the uppermost item of the stack; (b) `push()`, which pushes an item onto the stack; (c) `peek()`, which shows the topmost item of the stack without removing it; (d) `isEmpty()`, which checks whether the stack is empty or not.
>
> *Dengan kata lain:* lengkapilah program C++ di bawah dengan empat fungsi: `pop()` mengambil butir teratas stack, `push()` menaruh sebuah butir di atas stack, `peek()` memperlihatkan butir teratas tanpa mengambilnya, dan `isEmpty()` memeriksa apakah stack kosong.

Program yang harus dilengkapi adalah sebagai berikut. Hanya keempat fungsi anggotanya yang belum ada; kelasnya, konstruktornya, dan *driver* pendek di dalam `main` sudah diberikan.

```
#include <bits/stdc++.h>
using namespace std;
#define MAX 1000

class Stack {
    int top;
public:
    int a[MAX]; // Maximum size of Stack
    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
};

int main()
{
    class Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.pop() << " popped from stack\n";
    return 0;
}
```

8.3.2 Dengan kata-kata kita sendiri

Kita memiliki sebuah kotak berisi 1000 sel bilangan bulat, `a[0]` sampai `a[999]`, dan satu bilangan bulat `top`. Konstruktor menyetel `top` menjadi -1. Kita harus menulis empat fungsi sehingga objek ini berperilaku seperti stack pada Bab 5: `push(x)` menaruh `x` di puncak dan melaporkan keberhasilannya dengan `true` atau `false`; `pop()` mengambil butir teratas dan mengembalikannya; `peek()` mengembalikan butir teratas dan membiarkannya di tempat; `isEmpty()` menjawab apakah ada butir sama sekali. Untuk *driver* yang diberikan, keluaran yang diharapkan adalah satu baris, `30 popped from stack`, karena 30 adalah nilai terakhir yang di-*push*.

Kerangka program sudah mengambil keputusan terpenting untuk kita, dan seluruh soal ini adalah soal membacanya dengan benar: **`top` dimulai dari** -1. Hal itu memberi tahu kita makna `top`. Ia bukan "banyaknya butir" dan bukan "sel kosong berikutnya"; ia adalah *indeks sel yang memuat butir teratas*, dan -1 adalah indeks untuk "tidak ada sel sama sekali". Begitu kalimat itu dituliskan, setiap fungsi mengikutinya.

> **DEFINISI.** *Invarian stack* kelas ini: butir-butir stack adalah tepat `a[0]`, `a[1]`, …, `a[top]`, dari dasar sampai puncak, dan $-1 \le$ `top` $\le$ `MAX` -1. Stack kosong ketika `top` bernilai -1 dan penuh ketika `top` bernilai `MAX` -1.

Dari invarian itu kita dapat membaca banyaknya butir. Sel `a[0]` sampai `a[top]` berjumlah top + 1 sel, sehingga

$$\text{ukuran} = \text{top} + 1, \qquad \text{kosong} \Leftrightarrow \text{top} = -1, \qquad \text{penuh} \Leftrightarrow \text{top} = \text{MAX} - 1. \tag{8.1}$$

Persamaan (8.1) layak diperiksa pada kasus-kasus terkecil. Tanpa butir, `top` bernilai -1 dan ukurannya $-1 + 1 = 0$. Dengan satu butir, butir itu tinggal di `a[0]`, `top` bernilai 0, dan ukurannya 1. Ketika stack penuh, `top` bernilai 999 dan ukurannya 1000, yaitu `MAX`. Rumusnya benar di kedua ujung.

8.3.3 Dengan tangan, selangkah demi selangkah

Gambar 8.1 menggambar empat sel pertama larik sesudah setiap operasi *driver*. Ikutilah gambar itu sambil mengingat invariannya.



Gambar 8.1 Stack larik Soal 8.1 sesudah setiap operasi *driver*: `top` adalah indeks butir teratas, dan `pop` hanya menggeser `top`; nilai 30 tetap berada di larik tetapi bukan lagi bagian dari stack.

1. **Awal.** `top` bernilai -1. Tidak ada sel yang menjadi bagian stack.
2. **`push(10)`.** Stack belum penuh ($-1 \neq 999$). Butir baru harus masuk ke sel *di atas* puncak saat ini, yaitu sel $-1 + 1 = 0$. Jadi kita naikkan dulu `top` menjadi 0, baru menyimpan: `a[0] = 10`. Invarian kembali berlaku: butir-butirnya adalah `a[0]..a[0]`.
3. **`push(20)`.** `top` menjadi 1, lalu `a[1] = 20`.
4. **`push(30)`.** `top` menjadi 2, lalu `a[2] = 30`. Butir-butirnya adalah `a[0]..a[2]` = 10, 20, 30.
5. **`pop()`.** Stack tidak kosong ($2 \neq -1$). Butir teratasnya adalah `a[top]` = `a[2]` = 30. Kita harus *membacanya lebih dulu*, *baru* menurunkan `top` menjadi 1. Fungsi mengembalikan 30, dan *driver* mencetak `30 popped from stack`. Perhatikan bahwa tidak ada yang dihapus: `a[2]` masih berisi 30. Nilai itu hanya tidak lagi berada di antara `a[0]` dan `a[top]`, sehingga menurut invarian ia bukan lagi butir. *Push* berikutnya akan menimpanya.

Urutan dua tindakan di dalam `push` dan `pop` adalah seluruh kesulitan soal ini, dan C++ memungkinkan kita menulis setiap pasangan dalam satu ekspresi. Pada `a[++top] = x`, *pre-increment* `++top` menaikkan `top` lebih dulu, lalu memakai nilai barunya sebagai indeks. Pada `return a[top--];`, *post-decrement* `top--` memakai nilai lamanya sebagai indeks dan menurunkan `top` sesudahnya. Bacalah keduanya dengan suara keras sebagai "naikkan, lalu simpan" dan "baca, lalu turunkan".

Apa yang terjadi di tepi. *Push* pada stack yang penuh tidak boleh menulis ke `a[1000]`, yang tidak ada; `push` kita mencetak pesan dan mengembalikan `false`, dan untuk itulah tipe kembaliannya `bool`. *Pop* pada stack kosong tidak memiliki apa pun untuk dikembalikan. Tipe kembalian di kerangka adalah `int`, dan mengembalikan bilangan khusus seperti 0 atau -1 adalah dusta, sebab 0 dan -1 adalah butir yang sah yang mungkin disimpan stack sungguhan. Karena itu `pop` dan `peek` kita *melempar eksepsi* `std::underflow_error`, yang dapat ditangkap pemanggil; pemanggil yang santun bertanya `isEmpty()` lebih dulu dan tidak pernah memicunya.

8.3.4 Penyelesaian C++

Kode 8.1 adalah program yang sudah dilengkapi. Program ini mempertahankan kerangkanya persis, menambahkan satu fungsi bantu kecil `size()` yang kita pakai dalam pengujian, dan memperluas `main` dengan pemeriksaan kasus-kasus tepi sesudah *driver* yang diberikan. Kami mengganti `<bits/stdc++.h>`, yang hanya ada di GCC, dengan dua *header* standar yang benar-benar diperlukan program, agar ia dapat dikompilasi oleh setiap kompilator C++.

Kode 8.1 `p801_stack.cpp`: stack larik Soal 8.1, sudah dilengkapi, dengan *driver* yang diberikan dan pemeriksaan kasus tepi kita sendiri.

```cpp
// p801_stack.cpp - Soal 8.1: melengkapi kelas Stack berbasis larik.
#include <iostream>
#include <stdexcept>
using namespace std;

#define MAX 1000   // kapasitas setiap Stack, dipertahankan dari kerangka yang diberikan

class Stack {
    int top;       // indeks butir teratas; -1 berarti "stack kosong"

public:
    int a[MAX];    // penyimpanan: butir-butir berada di a[0..top]

    Stack() {
        top = -1;
    }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
    int size();    // satu fungsi bantu tambahan, tidak ada di kerangka
};

bool Stack::push(int x) {
    if (top >= MAX - 1) {          // a[MAX-1] sudah terpakai: penuh
        cout << "Stack overflow: " << x << " not pushed\n";
        return false;
    }
    a[++top] = x;                  // PERTAMA naikkan top, BARU simpan
    return true;
}

int Stack::pop() {
    if (isEmpty()) {
        throw underflow_error("pop from an empty stack");
    }
    return a[top--];               // PERTAMA baca a[top], BARU turunkan top
}

int Stack::peek() {
    if (isEmpty()) {
        throw underflow_error("peek at an empty stack");
    }
    return a[top];                 // hanya membaca; top tidak bergerak
}
```

Kode 8.1 (Lanjutan) *p801_stack.cpp: stack larik Soal 8.1, sudah dilengkapi, dengan driver yang diberikan dan pemeriksaan kasus tepi kita sendiri.*

```
bool Stack::isEmpty() {
    return top < 0;
}

int Stack::size() {
    return top + 1;
}

int main() {
    Stack s;                                    // driver yang diberikan
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.pop() << " popped from stack\n";

    // Pemeriksaan tambahan kita sendiri.
    cout << "peek -> " << s.peek() << ", size " << s.size() << "\n";
    cout << "pop -> " << s.pop() << "\n";
    cout << "pop -> " << s.pop() << "\n";
    cout << boolalpha << "isEmpty -> " << s.isEmpty() << "\n";
    try {
        s.pop();
    } catch (const underflow_error& e) {
        cout << "caught: " << e.what() << "\n";
    }

    int pushed = 0;
    for (int v = 1; v <= MAX; v++) {
        if (s.push(v)) pushed++;
    }
    cout << "pushed " << pushed << " items, size " << s.size() << "\n";
    bool ok = s.push(1001);                     // mencetak pesan overflow
    cout << "push(1001) returned " << ok << "\n";
    cout << "peek -> " << s.peek() << "\n";
    return 0;
}
```

```
30 popped from stack
peek -> 20, size 2
pop -> 20
pop -> 10
isEmpty -> true
caught: pop from an empty stack
pushed 1000 items, size 1000
Stack overflow: 1001 not pushed
push(1001) returned false
peek -> 1000
```

Baris demi baris. Baris keluaran pertama berasal dari *driver* yang diberikan, dan isinya sesuai harapan: `30 popped from stack`. Baris-baris berikutnya memeriksa `peek` (mengembalikan 20 dan membiarkan ukurannya 2), mengosongkan stack dengan dua kali *pop* (20, lalu 10, dalam urutan terbalik dari *push*, sebagaimana dituntut LIFO), memastikan bahwa `isEmpty` kini bernilai benar, dan memperlihatkan bahwa *pop* ketiga ditolak dengan eksepsi, bukan dengan mengembalikan nilai karangan. Perulangan kemudian mem-*push* 1000 nilai dan menghitung berapa yang diterima: semua 1000. *Push* ke-1001 ditolak, pesannya berasal dari dalam `push`, dan `peek` memastikan bahwa puncaknya masih nilai 1000, jadi tidak ada yang tertimpa. `boolalpha` membuat `cout` mencetak `true` dan `false`, bukan 1 dan 0.

Mengapa `isEmpty` ditulis `top < 0`, bukan `top == -1`? Menurut invarian keduanya sama, jadi keduanya benar; `< 0` hanya sedikit lebih pemaaf seandainya kelak ada *bug* yang mendorong `top`

di bawah −1. Mengapa `pop` memanggil `isEmpty()` alih-alih mengulang pengujiannya? Karena dengan begitu makna "kosong" ditulis di satu tempat saja, dan perubahan padanya tidak mungkin meninggalkan satu fungsi di belakang.

Biaya. Setiap fungsi dari kelima fungsi itu mengerjakan sejumlah tetap langkah, berapa pun ukuran stacknya: satu perbandingan dan satu atau dua penugasan. Dalam notasi Bab 2, masing-masing berbiaya waktu $O(1)$. Seluruh larik dialokasikan sekali, ketika objek dibuat, sehingga stack memakai memori $\Theta(\text{MAX})$ bahkan ketika kosong; itulah harga stack larik, dan stack bertaut pada Soal 8.4 adalah jawabannya.

8.3.5 Penyelesaian Java dan Python

Kode 8.2 dan **Kode 8.3** adalah kelas yang sama dalam Java dan Python. Keduanya mencetak baris-baris yang persis sama dengan di atas. Java tidak memiliki `underflow_error`, jadi `pop` melempar `IllegalStateException`, eksepsi baku Java untuk "Anda memanggil metode ini pada saat ia tidak dapat bekerja". Python tidak memiliki larik berukuran tetap, jadi program membuatnya dengan `[0] * MAX`, sebuah list berisi 1000 nol yang tidak pernah dibiarkan tumbuh; program melempar `IndexError` pada stack kosong, eksepsi yang sama dengan yang dilempar `list.pop()` milik Python sendiri. Dalam program Python sungguhan, list biasa *adalah* stack: `append` untuk *push*, `pop()` untuk *pop*, dan `[-1]` untuk *peek*, semuanya dalam waktu $O(1)$ [8]. Di sini kita menulis versi larik agar ketiga program memiliki bentuk yang sama.

Kode 8.2 `P801Stack.java`: Soal 8.1 dalam Java.

```java
// P801Stack.java - Soal 8.1: melengkapi kelas Stack berbasis larik.
public class P801Stack {

    static final int MAX = 1000;       // kapasitas setiap Stack

    static class Stack {
        private int top = -1;          // indeks butir teratas; -1 berarti "stack kosong"
        private final int[] a = new int[MAX];   // butir-butir berada di a[0..top]

        boolean push(int x) {
            if (top >= MAX - 1) {      // a[MAX-1] sudah terpakai: penuh
                System.out.println("Stack overflow: " + x + " not pushed");
                return false;
            }
            a[++top] = x;              // PERTAMA naikkan top, BARU simpan
            return true;
        }

        int pop() {
            if (isEmpty()) {
                throw new IllegalStateException("pop from an empty stack");
            }
            return a[top--];           // PERTAMA baca a[top], BARU turunkan top
        }

        int peek() {
            if (isEmpty()) {
                throw new IllegalStateException("peek at an empty stack");
            }
            return a[top];
        }

        boolean isEmpty() {
            return top < 0;
        }

        int size() {
            return top + 1;
        }
    }
```

Kode 8.2 (Lanjutan) *P801Stack.java: Soal 8.1 dalam Java.*

```
    public static void main(String[] args) {
        Stack s = new Stack();                // driver yang diberikan
        s.push(10);
        s.push(20);
        s.push(30);
        System.out.println(s.pop() + " popped from stack");

        // Pemeriksaan tambahan kita sendiri.
        System.out.println("peek -> " + s.peek() + ", size " + s.size());
        System.out.println("pop -> " + s.pop());
        System.out.println("pop -> " + s.pop());
        System.out.println("isEmpty -> " + s.isEmpty());
        try {
            s.pop();
        } catch (IllegalStateException e) {
            System.out.println("caught: " + e.getMessage());
        }

        int pushed = 0;
        for (int v = 1; v <= MAX; v++) {
            if (s.push(v)) pushed++;
        }
        System.out.println("pushed " + pushed + " items, size " + s.size());
        boolean ok = s.push(1001);            // mencetak pesan overflow
        System.out.println("push(1001) returned " + ok);
        System.out.println("peek -> " + s.peek());
    }
}
```

Kode 8.3 `p801_stack.py`: Soal 8.1 dalam Python, dengan blok tetap berisi `MAX` sel agar logikanya sama dengan larik C++.

```python
# p801_stack.py - Soal 8.1: melengkapi kelas Stack berbasis larik.
MAX = 1000   # kapasitas setiap Stack


class Stack:
    def __init__(self):
        self._top = -1              # indeks butir teratas; -1 berarti "stack kosong"
        self.a = [0] * MAX          # blok tetap berisi MAX sel, seperti larik C++

    def push(self, x):
        if self._top >= MAX - 1:    # a[MAX-1] sudah terpakai: penuh
            print(f"Stack overflow: {x} not pushed")
            return False
        self._top += 1              # PERTAMA naikkan top,
        self.a[self._top] = x       # BARU simpan
        return True

    def pop(self):
        if self.is_empty():
            raise IndexError("pop from an empty stack")
        x = self.a[self._top]       # PERTAMA baca a[top],
        self._top -= 1              # BARU turunkan top
        return x

    def peek(self):
        if self.is_empty():
            raise IndexError("peek at an empty stack")
        return self.a[self._top]

    def is_empty(self):
        return self._top < 0

    def size(self):
        return self._top + 1


def main():
    s = Stack()                                 # driver yang diberikan
    s.push(10)
    s.push(20)
    s.push(30)
    print(s.pop(), "popped from stack")
```

Kode 8.3 (Lanjutan) *p801_stack.py: Soal 8.1 dalam Python, dengan blok tetap berisi MAX sel agar logikanya sama dengan larik C++.*

```python
    # Pemeriksaan tambahan kita sendiri.
    print(f"peek -> {s.peek()}, size {s.size()}")
    print("pop ->", s.pop())
    print("pop ->", s.pop())
    print("isEmpty ->", str(s.is_empty()).lower())
    try:
        s.pop()
    except IndexError as e:
        print("caught:", e)

    pushed = sum(1 for v in range(1, MAX + 1) if s.push(v))
    print(f"pushed {pushed} items, size {s.size()}")
    ok = s.push(1001)                           # mencetak pesan overflow
    print("push(1001) returned", str(ok).lower())
    print("peek ->", s.peek())


if __name__ == "__main__":
    main()
```

8.3.6 Penyelesaian keliru yang menggoda

Kesalahan paling umum pada soal ini adalah membalik urutan "geser" dan "baca" di dalam `pop`:

```
int pop() { return a[--top]; }   // KELIRU: menggeser top dulu, baru membaca
```

Kode itu terkompilasi tanpa peringatan dan tampak hampir sama dengan baris yang benar. Jalankan pada *driver* yang diberikan (program `wrong/p801_wrong.cpp` di folder kode melakukan persis itu, dan melakukan *pop* dua kali):

```
20 popped from stack
10 popped from stack
isEmpty -> false
```

Pop pertama menurunkan `top` dari 2 menjadi 1 *sebelum* membaca, sehingga mengembalikan `a[1]`, yaitu 20, butir *di bawah* puncak. Nilai 30 tidak pernah dikembalikan sama sekali. Sesudah dua kali *pop*, `top` bernilai 0, sehingga stack mengaku tidak kosong padahal kita mem-*push* tiga butir dan baru mengambil dua; *pop* ketiga akan menghitung `a[-1]`, sel yang tidak ada. Invarian langsung memberi tahu versi mana yang benar: butir teratas *adalah* `a[top]`, sehingga `pop` harus membaca `a[top]` selagi `top` masih bernilai lama. Kesalahan cerminannya pada `push`, `a[top++] = x`, lebih buruk lagi: dengan `top` bernilai -1, ia menulis ke `a[-1]`, di luar larik, yang dalam C++ merupakan perilaku tak terdefinisi (*undefined behavior*). Kesalahan semacam itulah yang menjadi alasan keberadaan AddressSanitizer.

8.3.7 Uji diri

UJI DIRI. Andaikan konstruktor diubah menjadi `top = 0`, sehingga `top` kini berarti "indeks sel kosong berikutnya". Tulis ulang satu baris penting pada `push`, `pop`, dan `peek`, serta pengujian di dalam `isEmpty`, lalu berikan versi baru Persamaan (8.1).

Jawaban. `push`: `a[top++] = x;` (simpan, lalu naikkan), dan pengujian penuh menjadi `top >= MAX`. `pop`: `return a[--top];` (turunkan, lalu baca). `peek`: `return a[top - 1];`. `isEmpty`: `return top == 0;`. Kini butir-butirnya adalah `a[0]..a[top-1]`, sehingga ukuran = `top`, kosong $\Leftrightarrow$ `top` = 0, penuh $\Leftrightarrow$ `top` = `MAX`. Kedua konvensi benar; yang tidak pernah benar adalah mencampurnya, dan itulah yang dilakukan penyelesaian keliru pada Subbab 8.3.6.

8.4 Soal 8.2 — Menelusuri List Dua Arah Menurut Petunjuk

8.4.1 Soalnya

SOAL. Complete the program below in C++. Each input line has the form *direction step data*: *direction* is the direction in which to traverse next, `b` for backward or `f` for forward; *step* is the number of nodes to move in that direction (for example, `b 5` means traverse 5 nodes backward); *data* is the character stored in the node. The input stops when the direction is `0`. Print the data taken from the traversal. For the sample input `f 3 s`, `f 3 k`, `f 4 r`, `b 1 t`, `f 3 d`, `f 3 t`, `b 5 u`, `b 2 a`, `0 0 0` the output is `strukdat`.

Dengan kata lain: setiap baris masukan berisi *arah langkah data*: arah `f` berarti maju dan `b` berarti mundur, langkah adalah banyaknya simpul yang dilalui ke arah itu, dan data adalah karakter yang disimpan di simpul. Masukan berhenti ketika arahnya `0`. Cetaklah data yang diambil selama penelusuran; untuk contoh masukan di atas keluarannya `strukdat`.

Kode yang diberikan, dengan badan kedua fungsinya masih kosong, adalah:

```
typedef struct snode_t {            // Node
    char data, direction;
    int step;
    struct snode_t *next;
    struct snode_t *prev;
} node;

typedef struct dlist_t {            // Deque
    unsigned _size;
    node *_head;
} linkedList;

bool slist_isEmpty(linkedList *list) { return (list->_head == NULL); }
void slist_pushBack(linkedList *list, char direction, char data, int step) { /* push back
*/ }
void transverse(linkedList *myList) { /* traverse the list */ }

int main() {
    linkedList *myList = new linkedList;
    myList->_head = NULL;
    char direction, data;
    int step;
    do {
        cin >> direction >> step >> data;
        slist_pushBack(myList, direction, data, step);
    } while (direction != '0');
    transverse(myList);
    return 0;
}
```

(Kode yang diberikan menamai fungsi kedua `transverse`; kata yang lazim adalah *traverse*, menelusuri, dan program kita memakai kata itu.)

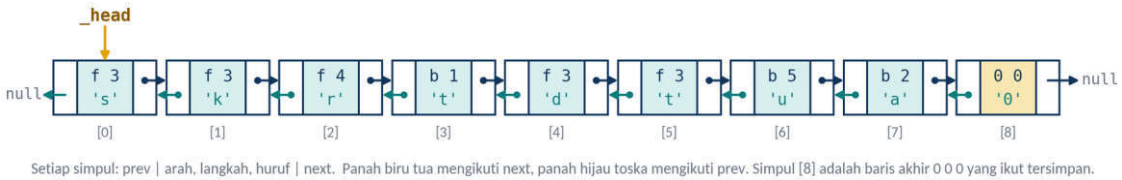
8.4.2 Secukupnya tentang simpul bertaut dua

Soal ini menjangkau selangkah melampaui Bab 7. Perhatikan tipe simpulnya: selain `next`, ia memiliki pointer kedua, `prev`. List yang simpul-simpulnya menunjuk ke penerus sekaligus ke pendahulunya disebut *doubly linked list*, dan Bab 9 dikhususkan untuknya: penyisipan dan penghapusan di kedua ujung, list dengan tail, versi sirkularnya, dan *deque* (*double-ended queue*, antrean berujung ganda) yang disebut dalam komentar kode yang diberikan. Untuk soal ini kita memerlukan jauh lebih sedikit daripada Bab 9, dan semuanya muat dalam tiga kalimat.

1. **Setiap simpul memiliki dua panah.** `next` menunjuk ke simpul sesudahnya dan `prev` ke simpul sebelumnya. `prev` pada simpul pertama bernilai `nullptr`, dan `next` pada simpul terakhir bernilai `nullptr`.
2. **Bergerak cukup dengan satu penugasan.** Bergerak satu simpul maju adalah `cur = cur->next`, persis seperti di Bab 7; bergerak satu simpul mundur adalah `cur = cur->prev`. Tidak ada hal baru selain itu.
3. **Menambah di belakang menyetel dua panah, bukan satu.** Ketika simpul baru ditambahkan sesudah simpul terakhir saat ini, `last`, kita harus menyetel `last->next = newNode` (panah maju, seperti di Bab 7) *dan* `newNode->prev = last` (panah mundur). Melupakan penugasan kedua menghasilkan list yang tampak sempurna bila ditelusuri maju dan rusak pada saat pertama seseorang menelusurinya mundur.

> **DEFINISI.** *Invarian list dua arah*: untuk setiap simpul `p` yang memiliki penerus, `p->next->prev == p`; `prev` pada simpul pertama bernilai `nullptr`; `next` pada simpul terakhir bernilai `nullptr`. Dengan kata lain: setiap panah maju memiliki panah mundur pasangannya.

Gambar 8.2 memperlihatkan list yang dibangun dari contoh masukan. Masih ada satu hal lagi yang perlu diperhatikan di dalamnya, dan hal itu adalah kunci seluruh soal.



Gambar 8.2 List dua arah yang dibangun dari contoh masukan Soal 8.2: sembilan simpul, masing-masing dengan pointer `prev` dan `next`. Simpul 8 memuat baris akhir `0 0 0`, yang juga ikut disimpan oleh perulangan pembacaan yang diberikan.

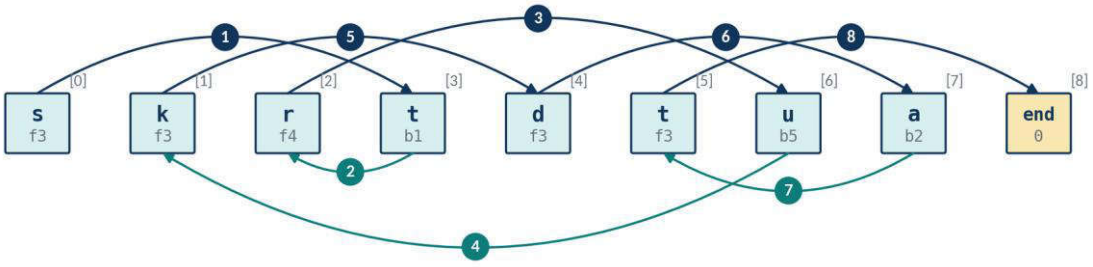
Baris akhir juga menjadi simpul. Bacalah `main` yang diberikan pelan-pelan. Ia adalah perulangan `do`–`while`: membaca satu baris, mem-*push*-nya, dan *baru kemudian* menguji apakah arahnya `'0'`. Jadi baris `0 0 0` di-*push* seperti baris lainnya, dan list berisi sembilan simpul, bukan delapan. Ini bukan *bug* yang harus diperbaiki; justru dengan cara inilah penelusuran tahu kapan berhenti. Penelusuran berakhir ketika ia *tiba* di simpul yang arahnya `'0'`. Contoh masukan disusun sedemikian sehingga lompatan kedelapan mendarat tepat di simpul itu.

8.4.3 Penelusuran, dengan tangan

Inilah aturan penelusurannya, dengan kata-kata kita sendiri. Mulailah di simpul pertama. Ulangi: jika arah simpul saat ini `'0'`, berhenti; jika tidak, cetak hurufnya, lalu bergeraklah sebanyak *langkah* simpul ke arahnya, satu simpul setiap kali. **Tabel 8.4** menjalankan aturan itu pada contoh masukan, dan **Gambar 8.3** menggambar kedelapan gerakan yang sama sebagai busur di atas dan di bawah list.

Tabel 8.4 Penelusuran Soal 8.2 pada contoh masukan. Setiap baris mencetak huruf simpul saat ini, lalu bergerak menurut petunjuk simpul itu sendiri.

Gerakan	Di simpul	Mencetak	Petunjuk	Mendarat di
1	0	`s`	`f 3`: maju 3	0 → 1 → 2 → 3
2	3	`t`	`b 1`: mundur 1	2
3	2	`r`	`f 4`: maju 4	6
4	6	`u`	`b 5`: mundur 5	1
5	1	`k`	`f 3`: maju 3	4
6	4	`d`	`f 3`: maju 3	7
7	7	`a`	`b 2`: mundur 2	5
8	5	`t`	`f 3`: maju 3	8
—	8	—	arah `'0'`: berhenti	—



Gambar 8.3 Delapan gerakan penelusuran: gerakan maju di atas list, gerakan mundur di bawahnya, diberi nomor menurut urutan terjadinya. Setiap simpul berhuruf dikunjungi tepat sekali.

Membaca kolom "Mencetak" dari atas ke bawah memberi `s t r u k d a t`, yaitu `strukdat` yang diharapkan, singkatan akrab untuk *struktur data* yang dieja oleh sebuah list yang menyembunyikan huruf-hurufnya secara acak. Perhatikan dari gambar bahwa penelusuran mengunjungi setiap simpul berhuruf tepat sekali. Itu adalah sifat masukan yang satu ini, bukan sifat setiap masukan, dan hal itu membawa kita ke pertanyaan berikutnya.

8.4.4 Apa yang dapat salah pada masukan

Program yang membaca masukan harus memutuskan apa yang dilakukan terhadap masukan yang tidak sesuai janji. Program ini menghadapi tiga bahaya, dan kode yang diberikan tidak menjaga satu pun.

Masukan berakhir tanpa baris `0 0 0`. Maka `cin >> direction` gagal, variabel-variabel mempertahankan nilai lamanya, dan perulangan `do`–`while` mem-*push* simpul yang sama berulang-ulang, selamanya. `main` kita menguji apakah pembacaan berhasil dan keluar dari perulangan jika tidak.

Sebuah gerakan melewati ujung. Petunjuk seperti `f 5` pada simpul kedua dari belakang akan mengikuti `next` melewati simpul terakhir sampai `nullptr`, dan `cur->next` berikutnya akan membuat program terhenti mendadak (*crash*). Penelusuran kita memeriksa `cur != nullptr` di dalam perulangan gerak dan melaporkan `[the walk left the list]`.

Petunjuknya berputar-putar. Dua simpul `f 1` dan `b 1` yang bersebelahan mengirim penelusuran bolak-balik di antara keduanya tanpa henti. Dapatkah program menyadarinya? Dapat, dengan argumen penghitungan. Ke mana penelusuran pergi berikutnya *hanya* bergantung pada simpul tempat ia berdiri. Jadi jika penelusuran pernah berdiri di simpul yang sama dua kali, semua yang terjadi sesudah kunjungan pertama akan terulang sesudah kunjungan kedua, dan lagi, selamanya. Dalam list berisi n simpul, penelusuran yang tidak pernah mengulang simpul paling banyak melakukan n kunjungan; jadi jika kita menghitung kunjungan dan hitungannya melampaui n, pasti ada simpul yang dikunjungi dua kali (bila kunjungan lebih banyak daripada simpul, dua kunjungan pasti berbagi simpul), dan penelusuran tidak akan pernah berakhir:

$$\begin{aligned} \text{kunjungan} > n \;&\Rightarrow\; \text{ada simpul yang dikunjungi dua kali} \\ &\Rightarrow\; \text{penelusuran tidak pernah mencapai akhir.} \end{aligned} \tag{8.2}$$

Persamaan (8.2) adalah *prinsip sarang merpati*: jika lebih dari n merpati menempati n sarang, ada sarang yang berisi dua merpati. Penelusuran kita menghitung kunjungannya dan berhenti dengan `[the directions go around in a circle]` begitu hitungannya melewati `_size`, banyaknya simpul. Pemeriksaan itu hanya memerlukan satu bilangan bulat dan satu perbandingan per kunjungan, dan ia mengubah program yang dapat macet menjadi program yang selalu berhenti.

8.4.5 Penyelesaian C++

Kode 8.4 mempertahankan tipe, nama, dan perulangan pembacaan yang diberikan, mengisi kedua fungsi, menambahkan pemeriksaan Subbab 8.4.4, dan membebaskan setiap simpul di akhir.

Kode 8.4 `p802_deque_walk.cpp`: Soal 8.2, list dua arah yang dibangun oleh `slist_pushBack` dan ditelusuri oleh `traverse`.

```cpp
// p802_deque_walk.cpp - Soal 8.2: membangun linked list dua arah, lalu menelusurinya
menurut petunjuk.
#include <iostream>
using namespace std;

typedef struct snode_t {       // satu simpul: sebuah huruf, arah, langkah, dan dua tautan
    char data, direction;
    int step;
    struct snode_t *next;      // simpul sesudah simpul ini (nullptr di ujung belakang)
    struct snode_t *prev;      // simpul sebelum simpul ini (nullptr di ujung depan)
} node;

typedef struct dlist_t {       // list itu sendiri: ukuran dan simpul pertama
    unsigned _size;
    node *_head;
} linkedList;

bool slist_isEmpty(linkedList *list) {
    return (list->_head == nullptr);
}

void slist_pushBack(linkedList *list, char direction, char data, int step) {
    node *newNode = new node;
    newNode->data = data;
    newNode->direction = direction;
    newNode->step = step;
    newNode->next = nullptr;              // simpul ini akan menjadi simpul terakhir
    if (slist_isEmpty(list)) {
        newNode->prev = nullptr;          // tidak ada apa-apa sebelum simpul pertama
        list->_head = newNode;
    } else {
        node *last = list->_head;
        while (last->next != nullptr) {   // berjalan ke simpul terakhir saat ini
            last = last->next;
        }
        last->next = newNode;             // tautan maju:  last -> newNode
        newNode->prev = last;             // tautan mundur: last <- newNode
    }
    list->_size++;
}
```

Kode 8.4 (Lanjutan) *p802_deque_walk.cpp: Soal 8.2, list dua arah yang dibangun oleh slist_pushBack dan ditelusuri oleh traverse.*

```
// Mencetak huruf yang dilewati penelusuran. Berhenti di simpul yang arahnya '0'.
void traverse(linkedList *myList) {
    node *cur = myList->_head;
    unsigned visits = 0;
    while (cur != nullptr && cur->direction != '0') {
        if (++visits > myList->_size) {   // sebuah simpul terkunjungi dua kali: putaran
tanpa akhir
            cout << " [the directions go around in a circle]";
            break;
        }
        cout << cur->data;
        int step = cur->step;             // SIMPAN petunjuk simpul ini
        char direction = cur->direction;  // sebelum cur beranjak darinya
        for (int k = 0; k < step && cur != nullptr; k++) {
            cur = (direction == 'b') ? cur->prev : cur->next;
        }
        if (cur == nullptr) {
            cout << " [the walk left the list]";
        }
    }
    cout << "\n";
}

void slist_destroy(linkedList *list) {
    node *cur = list->_head;
    while (cur != nullptr) {
        node *after = cur->next;          // simpan jalan ke depan sebelum menghapus
        delete cur;
        cur = after;
    }
    list->_head = nullptr;
    list->_size = 0;
}

int main() {
    linkedList *myList = new linkedList;
    myList->_size = 0;
    myList->_head = nullptr;
```

Kode 8.4 (Lanjutan) *p802_deque_walk.cpp: Soal 8.2, list dua arah yang dibangun oleh slist_pushBack dan ditelusuri oleh traverse.*

```
    char direction, data;
    int step;
    do {
        if (!(cin >> direction >> step >> data)) {
            break;                        // masukan berakhir tanpa baris 0 0 0
        }
        slist_pushBack(myList, direction, data, step);
    } while (direction != '0');

    traverse(myList);

    slist_destroy(myList);
    delete myList;
    return 0;
}
```

Berikut empat kali jalannya: contoh masukan, masukan kedua buatan kita sendiri, masukan yang petunjuknya berputar-putar, dan masukan yang gerakannya meninggalkan list. Berkas-berkas masukannya ada di folder `input` pada kode bab ini.

```
$ ./p802 < p802.txt
strukdat
$ ./p802 < p802_check.txt
list
$ ./p802 < p802_loop.txt
aba [the directions go around in a circle]
$ ./p802 < p802_off.txt
xy [the walk left the list]
```

Baris demi baris. `slist_pushBack` membuat simpul dengan `new`, mengisi ketiga medan datanya, dan menyetel `next` menjadi `nullptr`, karena simpul yang ditambahkan di belakang tidak memiliki penerus. Jika list kosong, simpul baru menjadi head dan `prev`-nya `nullptr`. Jika tidak, perulangan `while (last->next != nullptr) last = last->next;` berjalan ke simpul terakhir saat ini, sama dengan penelusuran pointer bantu pada `backInsertion` di Bab 7, dan dua penugasan `last->next = newNode;` dan `newNode->prev = last;` menyetel panah maju dan panah mundur. Penghitung ukuran bertambah satu pada kedua kasus.

`traverse` mengikuti aturan Subbab 8.4.3. Dua baris yang perlu dibaca dua kali adalah ini:

```
int step = cur->step;             // SIMPAN petunjuk simpul ini
char direction = cur->direction;  // sebelum cur beranjak darinya
```

Petunjuk itu milik simpul tempat gerakan *dimulai*. Begitu perulangan dalam mengeksekusi `cur = cur->next` yang pertama, `cur` sudah berdiri di simpul lain dengan petunjuk lain, sehingga banyaknya langkah dan arahnya harus disalin ke variabel lokal sebelum gerakan pertama. Kemudian perulangan dalam melakukan tepat `step` gerakan, mengikuti `prev` untuk `'b'` dan `next` untuk selainnya, dan berhenti lebih awal jika jatuh keluar dari list. `slist_destroy` membebaskan simpul satu per satu, menyimpan `cur->next` sebelum setiap `delete`, dan `main` menghapus kepala list itu sendiri, sehingga setiap `new` memiliki pasangan `delete`-nya. Valgrind dan AddressSanitizer tidak melaporkan kesalahan maupun kebocoran pada keempat masukan.

Biaya. Setiap `slist_pushBack` menelusuri seluruh list untuk menemukan simpul terakhir, sehingga membangun list berisi n simpul memerlukan $0 + 1 + \cdots + (n-1) = \frac{n(n-1)}{2}$ langkah, yaitu $\Theta(n^2)$. Tipe list yang diberikan memiliki `_head` tetapi tidak memiliki pointer tail, jadi biaya itu datang bersama soalnya; list pada Bab 9 menyimpan tail dan menambah di belakang dalam $O(1)$. Penelusurannya sendiri berbiaya satu langkah per simpul yang dilewati: $\sum$ nilai langkah simpul-simpul yang dikunjungi, yaitu 24 untuk contoh masukan.

8.4.6 Penyelesaian Java dan Python

Kode 8.5 dan **Kode 8.6** mencetak keempat hasil yang sama dengan program C++. Keduanya membaca seluruh masukan sekaligus dan memecahnya menjadi kata-kata, sehingga baris akhir yang hilang mudah dideteksi. Keduanya tidak memiliki fungsi penghancur: ketika `main` selesai, tidak ada lagi variabel yang merujuk ke simpul-simpulnya, dan *garbage collector* mengambilnya kembali.

Kode 8.5 `P802DequeWalk.java`: Soal 8.2 dalam Java.

```java
// P802DequeWalk.java - Soal 8.2: membangun linked list dua arah, lalu menelusurinya menurut petunjuk.
public class P802DequeWalk {

    static class Node {                // satu simpul: sebuah huruf, arah, langkah, dan dua tautan
        char data, direction;
        int step;
        Node next;                     // simpul sesudah simpul ini (null di ujung belakang)
        Node prev;                     // simpul sebelum simpul ini (null di ujung depan)
    }

    static class LinkedList {          // list itu sendiri: ukuran dan simpul pertama
        int size;
        Node head;
    }

    static boolean isEmpty(LinkedList list) {
        return list.head == null;
    }

    static void pushBack(LinkedList list, char direction, char data, int step) {
        Node newNode = new Node();
        newNode.data = data;
        newNode.direction = direction;
        newNode.step = step;
        newNode.next = null;                    // simpul ini akan menjadi simpul terakhir
        if (isEmpty(list)) {
            newNode.prev = null;                // tidak ada apa-apa sebelum simpul pertama
            list.head = newNode;
        } else {
            Node last = list.head;
            while (last.next != null) {         // berjalan ke simpul terakhir saat ini
                last = last.next;
            }
            last.next = newNode;                // tautan maju:  last -> newNode
            newNode.prev = last;                // tautan mundur: last <- newNode
        }
        list.size++;
    }
```

Kode 8.5 (Lanjutan) *P802DequeWalk.java: Soal 8.2 dalam Java.*

```
    // Mencetak huruf yang dilewati penelusuran. Berhenti di simpul yang arahnya '0'.
    static void traverse(LinkedList list) {
        StringBuilder out = new StringBuilder();
        Node cur = list.head;
        int visits = 0;
        while (cur != null && cur.direction != '0') {
            if (++visits > list.size) {        // sebuah simpul terkunjungi dua kali:
putaran tanpa akhir
                out.append(" [the directions go around in a circle]");
                break;
            }
            out.append(cur.data);
            int step = cur.step;               // SIMPAN petunjuk simpul ini
            char direction = cur.direction;    // sebelum cur beranjak darinya
            for (int k = 0; k < step && cur != null; k++) {
                cur = (direction == 'b') ? cur.prev : cur.next;
            }
            if (cur == null) {
                out.append(" [the walk left the list]");
            }
        }
        System.out.println(out);
    }

    public static void main(String[] args) throws java.io.IOException {
        String text = new String(System.in.readAllBytes()).trim();
        String[] tok = text.isEmpty() ? new String[0] : text.split("\\s+");
        LinkedList myList = new LinkedList();
        int i = 0;
        char direction;
        do {
            if (i + 2 >= tok.length) {
                break;                         // masukan berakhir tanpa baris 0 0 0
            }
            direction = tok[i].charAt(0);
            int step = Integer.parseInt(tok[i + 1]);
            char data = tok[i + 2].charAt(0);
            i += 3;
            pushBack(myList, direction, data, step);
        } while (direction != '0');
        traverse(myList);
        // Tanpa langkah penghancuran: garbage collector membebaskan simpul begitu tak ada
lagi yang merujuknya.
    }
}
```

Kode 8.6 `p802_deque_walk.py`: Soal 8.2 dalam Python. `__slots__` menetapkan medan-medan sebuah simpul, sehingga setiap simpul lebih kecil dan salah ketik pada nama medan tertangkap.

```python
# p802_deque_walk.py - Soal 8.2: membangun linked list dua arah, lalu menelusurinya menurut petunjuk.
import sys


class Node:
    """Satu simpul: sebuah huruf, arah, langkah, dan dua tautan."""
    __slots__ = ("data", "direction", "step", "next", "prev")

    def __init__(self, direction, step, data):
        self.data = data
        self.direction = direction
        self.step = step
        self.next = None            # simpul sesudah simpul ini (None di ujung belakang)
        self.prev = None            # simpul sebelum simpul ini (None di ujung depan)


class LinkedList:
    def __init__(self):
        self.size = 0
        self.head = None


def push_back(lst, direction, data, step):
    new_node = Node(direction, step, data)
    if lst.head is None:
        lst.head = new_node         # prev-nya tetap None: tidak ada apa-apa sebelum simpul pertama
    else:
        last = lst.head
        while last.next is not None:    # berjalan ke simpul terakhir saat ini
            last = last.next
        last.next = new_node            # tautan maju:  last -> new_node
        new_node.prev = last            # tautan mundur: last <- new_node
    lst.size += 1


def traverse(lst):
    """Mencetak huruf yang dilewati penelusuran. Berhenti di simpul yang arahnya '0'."""
    out = []
    cur = lst.head
    visits = 0
    while cur is not None and cur.direction != "0":
        visits += 1
        if visits > lst.size:           # sebuah simpul terkunjungi dua kali: putaran tanpa akhir
            out.append(" [the directions go around in a circle]")
            break
        out.append(cur.data)
```

Kode 8.6 (Lanjutan) *p802_deque_walk.py: Soal 8.2 dalam Python. __slots__ menetapkan medan-medan sebuah simpul, sehingga setiap simpul lebih kecil dan salah ketik pada nama medan tertangkap.*

```
        step = cur.step                 # SIMPAN petunjuk simpul ini
        direction = cur.direction       # sebelum cur beranjak darinya
        k = 0
        while k < step and cur is not None:
            cur = cur.prev if direction == "b" else cur.next
            k += 1
        if cur is None:
            out.append(" [the walk left the list]")
    print("".join(out))


def main():
    tok = sys.stdin.read().split()
    my_list = LinkedList()
    i = 0
    while i + 2 < len(tok):             # berhenti lebih awal jika baris 0 0 0 tidak ada
        direction, step, data = tok[i], int(tok[i + 1]), tok[i + 2]
        i += 3
        push_back(my_list, direction, data, step)
        if direction == "0":
            break
    traverse(my_list)


if __name__ == "__main__":
    main()
```

8.4.7 Penyelesaian keliru yang menggoda

Penelusuran yang tampak paling ringkas memakai medan simpul langsung di dalam syarat perulangan:

```
for (int k = 0; k < cur->step; k++)
    cur = (cur->direction == 'b') ? cur->prev : cur->next;   // KELIRU
```

Kalimatnya terdengar wajar: “ambil `step` langkah ke arah simpul itu”. Tetapi `cur` berubah di dalam perulangan, sehingga `cur->step` dan `cur->direction` dibaca dari *simpul yang berbeda* pada setiap putaran. Pada contoh masukan, penelusuran dimulai dengan benar di simpul 0 (`f 3`), melangkah satu kali ke simpul 1, lalu membandingkan `k` dengan langkah simpul 1, melangkah lagi ke simpul 2, membandingkan dengan langkah simpul 2 (yaitu 4), melangkah ketiga kalinya ke simpul 3, dan di sana membaca arah `b`. Dari simpul 3 ia mundur ke simpul 2, yang arahnya `f`, lalu maju lagi ke simpul 3. Ia terperangkap. Program `wrong/p802_wrong.cpp` melakukan penelusuran ini dan menghentikannya sesudah dua puluh huruf:

```
sttttttttttttttttttt
```

Perbaikannya adalah aturan Subbab 8.1 dalam bentuknya yang paling murni: *salinlah apa yang Anda perlukan dari sebuah simpul sebelum Anda meninggalkannya*.

8.4.8 Uji diri

> **UJI DIRI.** Apa yang dicetak program untuk masukan `f 2 l`, `f 2 s`, `b 1 i`, `f 1 t`, `0 0 0`? Gambarlah kelima simpul dan gerakan-gerakannya sebelum melihat jawabannya. Kemudian sebutkan satu perubahan pada baris kedua yang akan membuat penelusuran meninggalkan list.
>
> *Jawaban.* Penelusurannya berjalan simpul 0 `l` → maju 2 → simpul 2 `i` → mundur 1 → simpul 1 `s` → maju 2 → simpul 3 `t` → maju 1 → simpul 4, simpul akhir. Keluarannya `list`, seperti

diperlihatkan jalannya yang kedua pada **Kode 8.4**. Mengubah baris kedua menjadi `f 4 s` membuat gerakan dari simpul 1 maju empat simpul, ke simpul 5, yang tidak ada: penelusuran mencetak `lis` lalu `[the walk left the list]`.

8.5 Soal 8.3 — Genap Naik, Ganjil Turun, Hanya dengan Linked List

8.5.1 Soalnya

SOAL. Write a C++ program that sorts the numbers it reads: output the even numbers in ascending order, then the odd numbers in descending order. You must use a linked list for this problem; do not use arrays. Input: the first line is *N*, the number of values; each of the next *N* lines holds one value. Output: the sorted values, one per line. For the input 4, 32, 34, 543, 3456, 654, 567, 87, 6789, 98 the output is 4, 32, 34, 98, 654, 3456, 6789, 567, 543, 87.

Dengan kata lain: urutkan bilangan-bilangan masukan: cetak bilangan genap secara menaik, lalu bilangan ganjil secara menurun, satu per baris. Wajib memakai linked list; larik tidak boleh dipakai. Baris pertama masukan adalah banyaknya nilai, disusul baris-baris yang masing-masing berisi satu nilai.

8.5.2 Dengan kata-kata kita sendiri

Pisahkan bilangan-bilangan itu menjadi dua kelompok, yang genap dan yang ganjil. Cetak kelompok genap dari yang terkecil sampai yang terbesar, lalu kelompok ganjil dari yang terbesar sampai yang terkecil. Satu-satunya batasan ada pada *caranya*: tidak boleh ada larik yang menampung bilangan-bilangan itu, sehingga kita tidak dapat membacanya ke dalam larik lalu memanggil fungsi pengurutan. Bilangan-bilangan itu harus tinggal di linked list.

Batasan itu langsung menunjuk ke sebuah teknik dari Bab 7: jika setiap nilai baru disisipkan *di tempatnya yang benar* pada list yang sudah terurut, list itu tetap terurut sesudah setiap penyisipan, dan ketika masukan habis tidak ada lagi yang perlu diurutkan. Inilah *insertion sort* (pengurutan sisip, Bab 3) yang dikerjakan pada linked list, dan pada linked list cara ini terasa sangat wajar, karena menyisipkan di tengah hanya memerlukan dua penugasan pointer tanpa menggeser elemen apa pun. Kita memelihara dua list, `evens` yang terurut naik dan `odds` yang terurut turun, lalu mengirim setiap nilai ke list yang tepat.

Kedua list memerlukan dua urutan yang berbeda, tetapi penyisipannya sendiri adalah prosedur yang sama. Alih-alih menulisnya dua kali, kita menulisnya sekali dan memberinya sebuah *fungsi* yang menjawab satu pertanyaan: "haruskah `a` datang sebelum `b`?" Untuk `evens` jawabannya `a < b`; untuk `odds` jawabannya `a > b`. Di C++ sebuah fungsi dapat dikirim ke fungsi lain melalui *pointer ke fungsi*, gagasan Bab 6 yang kini kita pakai sungguh-sungguh. Parameternya ditulis

```
bool (*before)(int, int)
```

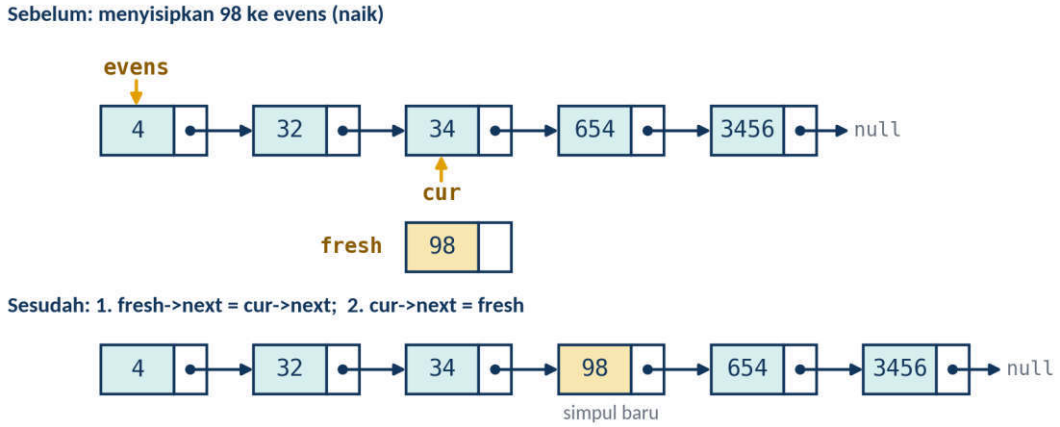
dan dibaca dari dalam ke luar: `before` adalah pointer (`*before`) ke sebuah fungsi yang menerima dua `int` dan mengembalikan `bool`. Di dalam `insertSorted` kita memanggilnya seperti fungsi biasa, `before(value, head->data)`, dan di tempat pemanggilan kita mengirim nama sebuah fungsi, `ascending` atau `descending`, tanpa tanda kurung.

8.5.3 Dengan tangan, selangkah demi selangkah

Apakah sebuah bilangan genap? Sebuah bilangan genap tepat ketika pembagiannya dengan 2 bersisa 0, sehingga pengujiannya `x % 2 == 0`. Pertahankan bentuk itu persis; Subbab 8.5.6 memperlihatkan mengapa `x % 2 == 1` untuk "ganjil", yang tampak lebih wajar, adalah jebakan.

Menyisipkan satu nilai. Ambil saat dalam contoh ketika 98 tiba dan `evens` sudah berisi 4, 32, 34, 654, 3456 (**Gambar 8.4**). Kita ingin simpul baru berada tepat sesudah *simpul terakhir yang harus tetap berada di depannya*. Ada dua kasus.

1. **Nilai baru harus berada sebelum simpul pertama** (atau list masih kosong). Maka simpul baru menjadi yang pertama: `fresh->next = head; head = fresh;`. Ini mengubah `head` itu sendiri, dan karena itulah fungsi menerima `head` sebagai referensi.
2. **Selain itu**, mulai `cur` di simpul pertama dan gerakkan maju selama simpul *sesudahnya* masih harus datang lebih dulu: `while (cur->next != nullptr && !before(value, cur->next->data)) cur = cur->next;`. Untuk 98: 98 < 32? Tidak, maju ke 32. 98 < 34? Tidak, maju ke 34. 98 < 654? Ya: berhenti. `cur` berada di 34. Sekarang tautkan simpul baru dalam urutan yang aman: pertama `fresh->next = cur->next;` (simpul baru menunjuk ke 654), baru kemudian `cur->next = fresh;` (34 menunjuk ke simpul baru).



Gambar 8.4 Menyisipkan 98 ke list naik `evens`: `cur` berhenti di simpul terakhir yang tetap di depan (34); simpul baru ditautkan ke 654 lebih dulu, dan baru sesudah itu 34 ditautkan ke simpul baru.

Mengapa kedua penugasan harus dalam urutan itu? Jika kita menulis `cur->next = fresh` lebih dulu, satu-satunya pointer ke 654 akan tertimpa, dan 654 serta 3456 akan hilang: tak terjangkau, dan di C++ bocor. Lagi-lagi aturan Subbab 8.1.

Mengapa perulangan membandingkan dengan `cur->next`, bukan dengan `cur`? Karena pada singly linked list kita tidak dapat mundur. Jika `cur` berjalan *sampai ke* 654 sebelum kita sadar bahwa 98 harus berada di depannya, kita tidak punya jalan kembali ke 34, yang `next`-nya adalah pointer yang harus kita ubah. Melihat satu simpul ke depan membuat `cur` tetap berada di pendahulu, tempat perubahan itu terjadi. Pola "berdiri di pendahulu" ini muncul di hampir setiap penyisipan dan penghapusan pada linked list.

Mengapa `!before(value, cur->next->data)` dan bukan `before(cur->next->data, value)`? Keduanya hanya berbeda untuk nilai yang *sama*. Dengan syarat kita, nilai yang sama dengan nilai yang sudah ada di list terus bergerak dan disisipkan *sesudah* nilai kembarnya. Hal itu menjaga nilai-nilai yang sama tetap dalam urutan kedatangannya, sehingga pengurutannya *stabil*; masukan kedua di bawah berisi dua angka 4 untuk mengujinya.

Tabel 8.5 menerapkan prosedur itu pada kesepuluh nilai contoh.

Tabel 8.5 Sepuluh penyisipan pada contoh masukan Soal 8.3. Mencetak `evens` lalu `odds` menghasilkan keluaran yang diharapkan.

Nilai	Genap?	Masuk ke	List sesudah penyisipan
4	ya	`evens`, list kosong	4
32	ya	`evens`, sesudah 4	4, 32
34	ya	`evens`, sesudah 32	4, 32, 34
543	tidak	`odds`, list kosong	543
3456	ya	`evens`, di ujung	4, 32, 34, 3456
654	ya	`evens`, sesudah 34	4, 32, 34, 654, 3456
567	tidak	`odds`, pertama yang baru (567 > 543)	567, 543
87	tidak	`odds`, di ujung	567, 543, 87
6789	tidak	`odds`, pertama yang baru	6789, 567, 543, 87
98	ya	`evens`, sesudah 34	4, 32, 34, 98, 654, 3456

Biaya. Menyisipkan ke list berisi k simpul paling banyak melihat seluruh k simpul itu, sehingga pada kasus terburuk (setiap nilai lebih besar daripada semua nilai sebelumnya, pada list naik) n penyisipan melihat $0 + 1 + 2 + \cdots + (n-1)$ simpul:

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2). \tag{8.3}$$

Untuk sepuluh nilai contoh, Persamaan (8.3) memberi paling banyak 45 langkah, yang tidak berarti apa-apa; untuk sejuta nilai, hasilnya setengah triliun, dan program yang harus mengurutkan sejuta bilangan memakai *merge sort* (Bab 3; Soal 8.11 di bawah memperlihatkan langkah intinya pada linked list). Pencetakan berbiaya satu langkah per simpul. Memorinya satu simpul per nilai dan tidak ada yang lain, persis seperti yang diminta soal.

8.5.4 Penyelesaian C++

Kode 8.7 `p803_evens_odds.cpp`: Soal 8.3 dengan dua linked list terurut dan satu fungsi penyisipan yang menerima urutannya sebagai pointer ke fungsi.

```cpp
// p803_evens_odds.cpp - Soal 8.3: genap naik, lalu ganjil turun, hanya dengan linked list.
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node *next;
};

bool ascending(int a, int b) { return a < b; }    // haruskah a datang sebelum b?
bool descending(int a, int b) { return a > b; }

// Menyisipkan value ke list yang simpul pertamanya head, dengan menjaga urutan
// yang ditentukan fungsi before(). head adalah REFERENSI ke pointer milik pemanggil,
// sehingga fungsi ini dapat mengubah simpul mana yang pertama.
void insertSorted(Node *&head, int value, bool (*before)(int, int)) {
    Node *fresh = new Node{value, nullptr};
    if (head == nullptr || before(value, head->data)) {
        fresh->next = head;                // simpul baru menjadi yang pertama
        head = fresh;
        return;
    }
    Node *cur = head;                      // cari simpul terakhir yang tetap di depan
    while (cur->next != nullptr && !before(value, cur->next->data)) {
        cur = cur->next;
    }
    fresh->next = cur->next;               // 1. simpul baru menunjuk ke penerusnya
    cur->next = fresh;                     // 2. pendahulunya menunjuk ke simpul baru
}

void printAndFree(Node *&head) {
    while (head != nullptr) {
        cout << head->data << "\n";
        Node *gone = head;
        head = head->next;                 // maju dulu ...
        delete gone;                       // ... baru bebaskan simpul pertama yang lama
    }
}
```

Kode 8.7 (Lanjutan) *p803_evens_odds.cpp: Soal 8.3 dengan dua linked list terurut dan satu fungsi penyisipan yang menerima urutannya sebagai pointer ke fungsi.*

```cpp
int main() {
    Node *evens = nullptr;                 // dua list kosong
    Node *odds = nullptr;
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        int x;
        if (!(cin >> x)) break;
        if (x % 2 == 0) {                  // genap, juga untuk bilangan negatif
            insertSorted(evens, x, ascending);
        } else {
            insertSorted(odds, x, descending);
        }
    }
    printAndFree(evens);
    printAndFree(odds);
    return 0;
}
```

Contoh masukan, lalu masukan kedua berisi tujuh nilai dengan bilangan negatif, nol, dan angka 4 yang berulang:

```
$ ./p803 < p803.txt
4
32
34
98
654
3456
6789
567
543
87
$ ./p803 < p803_check.txt
-2
0
4
4
9
-3
-7
```

Baris demi baris. `insertSorted` menerima `Node *&head`, sebuah *referensi ke pointer*. Bacalah tipenya dari kanan: `head` adalah referensi (`&`) ke sebuah `Node *`. Di dalam fungsi, `head` karenanya adalah nama lain bagi variabel `evens` atau `odds` milik pemanggil, sehingga `head = fresh;` mengubah pointer milik pemanggil. Dengan `Node *head` biasa, fungsi hanya mengubah salinannya sendiri, simpul pertama yang baru tidak pernah sampai ke `main`, dan setiap nilai yang lebih kecil daripada simpul pertama saat ini akan lenyap diam-diam (dan bocor). `printAndFree` mencetak list dan membebaskannya dalam satu kali penelusuran: ia memajukan `head` *sebelum* menghapus simpul yang baru saja ditinggalkannya, sehingga sisa list selalu terjangkau. Di akhir, `evens` dan `odds` sama-sama `nullptr` dan setiap simpul sudah dibebaskan. Pada jalannya yang kedua, -7, -3, dan 9 ganjil, sedangkan -2, 0, 4, 4 genap: nol genap, karena $0 \bmod 2 = 0$.

8.5.5 Penyelesaian Java dan Python

Java tidak memiliki pointer ke fungsi maupun referensi ke variabel, dan memang tidak memerlukan keduanya. Urutan dikirim sebagai objek yang memiliki satu metode: **Kode 8.8** mendeklarasikan *functional interface* kecil `Before` dengan metode `test(a, b)`, dan kedua urutannya adalah *ekspresi lambda*, `(a, b) -> a < b` dan `(a, b) -> a > b`. Karena Java tidak dapat mengubah variabel milik pemanggil, `insertSorted` mengembalikan simpul pertama (mungkin yang baru) dan `main` menulis `evens = insertSorted(evens, x, ASCENDING);`. Python mengirim fungsi sebagai nilai biasa, sehingga **Kode 8.9** cukup menyerahkan `ascending` atau `descending`. Ketiga program mencetak baris-baris yang sama.

Kode 8.8 `P803EvensOdds.java`: Soal 8.3 dalam Java, dengan urutan yang dikirim sebagai ekspresi lambda.

```java
// P803EvensOdds.java - Soal 8.3: genap naik, lalu ganjil turun, hanya dengan linked list.
public class P803EvensOdds {

    static class Node {
        int data;
        Node next;

        Node(int data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    // "Haruskah a datang sebelum b?" - sebuah fungsi yang dapat dikirim sebagai argumen.
    interface Before {
        boolean test(int a, int b);
    }

    static final Before ASCENDING = (a, b) -> a < b;
    static final Before DESCENDING = (a, b) -> a > b;

    // Java tidak dapat mengubah variabel pemanggil, maka simpul pertama (mungkin yang
baru) dikembalikan.
    static Node insertSorted(Node head, int value, Before before) {
        Node fresh = new Node(value, null);
        if (head == null || before.test(value, head.data)) {
            fresh.next = head;                  // simpul baru menjadi yang pertama
            return fresh;
        }
        Node cur = head;                        // cari simpul terakhir yang tetap di depan
        while (cur.next != null && !before.test(value, cur.next.data)) {
            cur = cur.next;
        }
        fresh.next = cur.next;                  // 1. simpul baru menunjuk ke penerusnya
        cur.next = fresh;                       // 2. pendahulunya menunjuk ke simpul baru
        return head;
    }

    static void print(Node head, StringBuilder out) {
        for (Node p = head; p != null; p = p.next) {
            out.append(p.data).append('\n');
        }
    }
```

Kode 8.8 (Lanjutan) *P803EvensOdds.java: Soal 8.3 dalam Java, dengan urutan yang dikirim sebagai ekspresi lambda.*

```
    public static void main(String[] args) throws java.io.IOException {
        String text = new String(System.in.readAllBytes()).trim();
        String[] tok = text.isEmpty() ? new String[0] : text.split("\\s+");
        Node evens = null;                    // dua list kosong
        Node odds = null;
        int n = tok.length > 0 ? Integer.parseInt(tok[0]) : 0;
        for (int i = 1; i <= n && i < tok.length; i++) {
            int x = Integer.parseInt(tok[i]);
            if (x % 2 == 0) {                 // genap, juga untuk bilangan negatif
                evens = insertSorted(evens, x, ASCENDING);
            } else {
                odds = insertSorted(odds, x, DESCENDING);
            }
        }
        StringBuilder out = new StringBuilder();
        print(evens, out);
        print(odds, out);
        System.out.print(out);
    }
}
```

Kode 8.9 `p803_evens_odds.py`: Soal 8.3 dalam Python, dengan urutan yang dikirim sebagai fungsi.

```python
# p803_evens_odds.py - Soal 8.3: genap naik, lalu ganjil turun, hanya dengan linked list.
import sys


class Node:
    __slots__ = ("data", "next")

    def __init__(self, data, next=None):
        self.data = data
        self.next = next


def ascending(a, b):        # haruskah a datang sebelum b?
    return a < b


def descending(a, b):
    return a > b


def insert_sorted(head, value, before):
    """Mengembalikan simpul pertama list (mungkin yang baru), terurut menurut before()."""
    fresh = Node(value)
    if head is None or before(value, head.data):
        fresh.next = head               # simpul baru menjadi yang pertama
        return fresh
    cur = head                          # cari simpul terakhir yang tetap di depan
    while cur.next is not None and not before(value, cur.next.data):
        cur = cur.next
    fresh.next = cur.next               # 1. simpul baru menunjuk ke penerusnya
    cur.next = fresh                    # 2. pendahulunya menunjuk ke simpul baru
    return head


def main():
    tok = sys.stdin.read().split()
    evens = None                        # dua list kosong
    odds = None
    n = int(tok[0]) if tok else 0
    for t in tok[1:n + 1]:
        x = int(t)
        if x % 2 == 0:                  # genap; di Python -7 % 2 bernilai 1, di C++ dan
Java -1
            evens = insert_sorted(evens, x, ascending)
        else:
            odds = insert_sorted(odds, x, descending)
    out = []
```

Kode 8.9 (Lanjutan) *p803_evens_odds.py: Soal 8.3 dalam Python, dengan urutan yang dikirim sebagai fungsi.*

```python
    for head in (evens, odds):
        p = head
        while p is not None:
            out.append(str(p.data))
            p = p.next
    if out:
        print("\n".join(out))


if __name__ == "__main__":
    main()
```

8.5.6 Penyelesaian keliru yang menggoda

Banyak orang lebih dulu memikirkan “ganjil” dan menulis pemisahannya seperti ini:

```
if (x % 2 == 1) insertSorted(odds, x, descending);    // KELIRU untuk x negatif
else            insertSorted(evens, x, ascending);
```

Pada contoh masukan, program ini mencetak jawaban yang benar, karena semua bilangan di sana positif. Pada masukan kedua, program `wrong/p803_wrong.cpp` mencetak

```
-7
-3
-2
0
4
4
9
```

Bilangan ganjil -7 dan -3 bergabung dengan bilangan genap, dan hanya 9 yang tersisa di antara bilangan ganjil. Penyebabnya adalah cara C++ membagi bilangan negatif. Sejak C++11, hasil bagi `/` *dibulatkan ke arah nol*, sehingga $-7/2$ bernilai -3, dan sisanya harus memenuhi $(-7/2)\cdot 2 + (-7 \,\%\, 2) = -7$, yang membuat `-7 % 2` bernilai -1, bukan 1 [9]. Java mendefinisikan `%` dengan cara yang sama. Python sebaliknya membulatkan hasil baginya *ke bawah*, sehingga di Python `-7 % 2` bernilai 1 dan pengujian yang keliru kebetulan berhasil, sebagaimana dicatat komentar di **Kode 8.9** [8]. Kebiasaan yang aman, di bahasa apa pun, adalah menguji kegenapan dengan `x % 2 == 0` dan membiarkan "ganjil" menjadi sisanya.

8.5.7 Uji diri

> **UJI DIRI.** Berikan keluaran **Kode 8.7** untuk $N = 6$ dan nilai-nilai 5, 0, -4, 15, -9, 8, lalu keluaran program keliru pada Subbab 8.5.6 untuk masukan yang sama. Berapa simpul yang dibuat program yang benar, dan berapa yang dibebaskannya?
>
> *Jawaban.* Benar: -4, 0, 8, lalu 15, 5, -9. Keliru: -9, -4, 0, 8, lalu 15, 5, karena `-9 % 2` bernilai -1 sehingga -9 dicatat sebagai genap. Keduanya membuat enam simpul, satu per nilai, dan program yang benar membebaskan keenamnya di `printAndFree`.

8.6 Soal 8.4 — Membalik Urutan Kata Sebuah Kalimat

8.6.1 Soalnya

> **SOAL.** Write a C++ program that reverses a string using a linked list. Sample input: `practice makes a man perfect`. Sample output: `perfect man a makes practice`.
>
> *Dengan kata lain:* tulislah program C++ yang membalik sebuah string dengan memakai linked list; contoh masukan dan keluarannya seperti di atas.

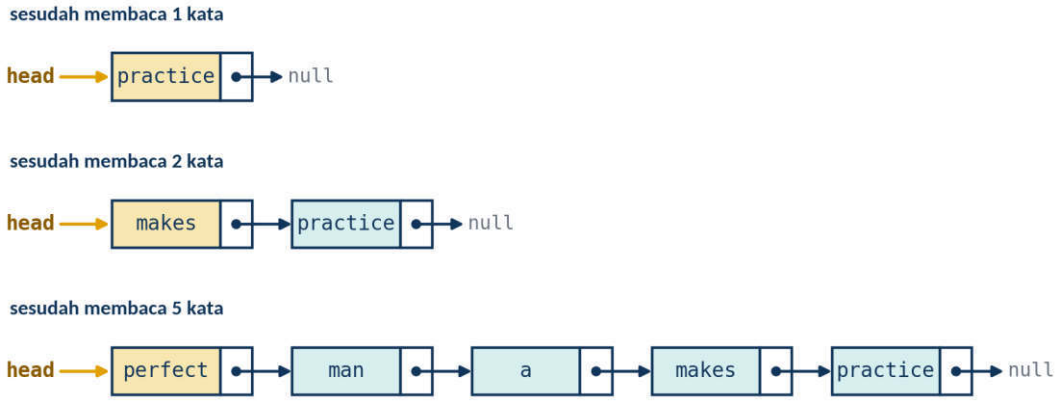
8.6.2 Dengan kata-kata kita sendiri

Contohnya memperlihatkan apa arti "membalik" di sini: *kata-kata* keluar dalam urutan terbalik, dan setiap kata mempertahankan ejaannya sendiri. Kita membaca satu baris, memotongnya menjadi kata-kata di setiap spasi, lalu mencetak kata-kata itu dari yang terakhir, dipisahkan oleh satu spasi.

Struktur apa yang dipakai? "Kata yang terakhir dibaca adalah kata yang pertama dicetak" berarti terakhir masuk, pertama keluar: stack. Dan stack mudah dibangun dari linked list, sebab `frontInsertion` pada Bab 7 *adalah* sebuah *push*: simpul baru ditaruh di depan simpul pertama saat ini, dan `head` berpindah ke simpul baru. Membaca list dari `head` sampai ujungnya lalu memberi kata-kata dalam urutan terbalik dari penyisipannya, persis yang kita inginkan. Tidak ada yang perlu dibalik sesudahnya; pembalikan terjadi selama pembangunan.

8.6.3 Dengan tangan, selangkah demi selangkah

Gambar 8.5 memperlihatkan list sesudah kata pertama, sesudah kata kedua, dan sesudah kelima kata.



Gambar 8.5 Membangun list Soal 8.4 dengan penyisipan depan: setiap kata baru menjadi simpul pertama, sehingga list membaca kalimatnya dari belakang.

1. List kosong: `head` bernilai `nullptr`.
2. Baca `practice`. Buat simpul yang `next`-nya adalah `head` saat ini (`nullptr`), lalu buat `head` menunjuk ke simpul itu. List berisi `practice`.
3. Baca `makes`. Buat simpul yang `next`-nya adalah `head` saat ini (simpul `practice`), lalu pindahkan `head` ke simpul itu. List berisi `makes`, `practice`.
4. Baca `a`, `man`, `perfect` dengan cara yang sama. List berisi `perfect`, `man`, `a`, `makes`, `practice`.
5. Telusuri dari `head` sampai `nullptr`, cetak setiap kata, dengan spasi *di antara* kata: `perfect man a makes practice`.

Di C++, langkah 2 dan 3 adalah satu baris, `head = new WordNode{w, head};`. Ruas kanan dievaluasi lebih dulu, sehingga simpul baru menerima nilai `head` yang *lama* sebagai `next`-nya, dan baru sesudah itu `head` diubah menjadi simpul baru. Sekali lagi, urutanlah yang membuatnya bekerja.

Memotong baris menjadi kata-kata. `getline(cin, line)` membaca seluruh baris, termasuk spasinya (`cin >> line` biasa akan berhenti di spasi pertama). Sebuah `istringstream` lalu memungkinkan kita membaca kata-kata dari string itu dengan `>>`, persis seperti membaca bilangan dari papan ketik, dan `>>` melewati berapa pun banyaknya spasi di antara kata. Jadi beberapa spasi di antara kata, atau spasi di awal dan akhir baris, tidak menghasilkan kata kosong. Masukan kedua di bawah mengujinya.

Biaya. Setiap kata di-*push* sekali dan dicetak sekali, dan setiap karakter disalin sejumlah tetap kali, sehingga program berjalan dalam waktu sebanding dengan panjang baris, $\Theta(L)$ untuk baris berisi L karakter. Memorinya satu simpul per kata ditambah karakter-karakternya sendiri.

8.6.4 Penyelesaian C++

Kode 8.10 `p804_reverse_words.cpp`: Soal 8.4 dengan linked list yang dipakai sebagai stack.

```cpp
// p804_reverse_words.cpp - Soal 8.4: membalik urutan kata sebuah kalimat dengan linked
list.
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

struct WordNode {
    string word;
    WordNode *next;
};

// Penyisipan depan: simpul baru ditaruh sebelum simpul pertama saat ini.
void pushFront(WordNode *&head, const string &w) {
    head = new WordNode{w, head};          // simpul baru menunjuk ke simpul pertama yang
lama
}

int main() {
    string line;
    getline(cin, line);                    // seluruh kalimat, termasuk spasi

    WordNode *head = nullptr;
    istringstream words(line);
    string w;
    while (words >> w) {                   // satu kata setiap kali, spasi dilewati
        pushFront(head, w);                // kata yang terakhir dibaca menjadi yang
pertama
    }

    bool first = true;
    while (head != nullptr) {
        if (!first) cout << ' ';           // spasi HANYA DI ANTARA kata, tidak pernah di
akhir
        cout << head->word;
        first = false;
        WordNode *gone = head;
        head = head->next;
        delete gone;
    }
    cout << "\n";
    return 0;
}
```

```
$ ./p804 < p804.txt
perfect man a makes practice
$ ./p804 < p804_check.txt
friends are queues and stacks
```

Baris demi baris. `pushFront` menerima `WordNode *&head`, referensi ke pointer milik pemanggil, dengan alasan yang sama seperti `insertSorted` pada Soal 8.3: ia harus mengubah simpul mana yang pertama. Perulangan pencetakan memakai penanda `first` untuk menaruh spasi sebelum setiap kata kecuali yang pertama, sehingga baris itu tidak berakhir dengan spasi; spasi di akhir tidak terlihat di layar, tetapi membuat keluaran berbeda dari keluaran yang diharapkan. Perulangan itu membebaskan setiap simpul tepat sesudah mencetaknya, dengan memajukan `head` sebelum `delete`. Masukan kedua, `stacks and queues are friends` dengan spasi berlebih di mana-mana, keluar sebagai `friends are queues and stacks`.

8.6.5 Penyelesaian Java dan Python

Kode 8.11 membaca baris dengan `BufferedReader.readLine()` dan memotongnya dengan `split("\\s+")`, ekspresi reguler yang berarti “satu atau lebih karakter spasi”, sesudah `trim()`

membuang spasi di kedua ujung. **Kode 8.12** memakai `line.split()`, yang tanpa argumen memotong di setiap deretan spasi dan mengabaikannya di kedua ujung. Pada keduanya, konstruktor simpul menerima `head` yang lama sebagai `next`-nya, yang tidak lain adalah penyisipan depan versi C++, dan di Python `" ".join(words)` menaruh spasi hanya di antara kata.

Kode 8.11 `P804ReverseWords.java`: Soal 8.4 dalam Java.

```
// P804ReverseWords.java - Soal 8.4: membalik urutan kata sebuah kalimat dengan linked
list.
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class P804ReverseWords {

    static class WordNode {
        String word;
        WordNode next;

        WordNode(String word, WordNode next) {
            this.word = word;
            this.next = next;
        }
    }

    public static void main(String[] args) throws java.io.IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line = in.readLine();              // seluruh kalimat, termasuk spasi
        if (line == null) line = "";

        WordNode head = null;
        for (String w : line.trim().split("\\s+")) {
            if (!w.isEmpty()) {
                head = new WordNode(w, head);     // penyisipan depan: kata yang terakhir
dibaca menjadi yang pertama
            }
        }

        StringBuilder out = new StringBuilder();
        for (WordNode p = head; p != null; p = p.next) {
            if (p != head) out.append(' ');       // spasi HANYA DI ANTARA kata, tidak
pernah di akhir
            out.append(p.word);
        }
        System.out.println(out);
    }
}
```

Kode 8.12 `p804_reverse_words.py`: Soal 8.4 dalam Python.

```python
# p804_reverse_words.py - Soal 8.4: membalik urutan kata sebuah kalimat dengan linked list.
import sys


class WordNode:
    __slots__ = ("word", "next")

    def __init__(self, word, next):
        self.word = word
        self.next = next


def main():
    line = sys.stdin.readline()             # seluruh kalimat, termasuk spasi
    head = None
    for w in line.split():                  # satu kata setiap kali, spasi dilewati
        head = WordNode(w, head)            # penyisipan depan: kata yang terakhir dibaca
menjadi yang pertama
    words = []
    p = head
    while p is not None:
        words.append(p.word)
        p = p.next
    print(" ".join(words))                  # spasi HANYA DI ANTARA kata, tidak pernah di
akhir


if __name__ == "__main__":
    main()
```

8.6.6 Penyelesaian keliru yang menggoda

Pemrogram yang lebih dulu belajar C sering membaca baris ke dalam larik karakter berukuran tetap lalu memotongnya dengan `strtok`:

```
char str[100];                              // KELIRU: mengira kalimatnya pendek
int i = 0, c;
while ((c = getchar()) != '\n' && c != EOF)
    str[i++] = (char)c;                     // tanpa memeriksa ukuran str
str[i] = '\0';
```

Pada contoh masukan, cara ini berhasil, kecuali satu hal kecil: versi di folder kode mencetak setiap kata diikuti spasi, sehingga baris keluarannya berakhir dengan spasi yang tidak ada pada keluaran yang diharapkan. Masalah sesungguhnya muncul pada kalimat sepanjang 100 karakter atau lebih. Maka `str[i++]` menulis melewati ujung larik, ke memori apa pun yang kebetulan berada sesudahnya. Program itu tidak wajib terhenti; ia mungkin mencetak omong kosong, atau mencetak jawaban benar sambil merusak variabel yang dipakai kemudian. Untuk itulah *sanitizer* pada Subbab 8.2 ada. Diberi kalimat sepanjang 104 karakter, program `wrong/p804_wrong.cpp` yang dibangun dengan `-fsanitize=address,undefined` berhenti pada operasi tulis keliru yang pertama; baris-baris awal laporannya adalah (alamat-alamatnya dipersingkat di sini, karena berubah setiap kali dijalankan):

```
wrong/p804_wrong.cpp:15:16: runtime error: index 100 out of bounds for type 'char [100]'
...
==...==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x...
WRITE of size 1 at 0x... thread T0
    #0 0x... in main wrong/p804_wrong.cpp:15
```

Laporan itu menyebut jenis kesalahannya (operasi tulis melewati ujung larik di *stack*), ukurannya (satu bait), dan baris tepatnya. Perbaikannya adalah membiarkan string tumbuh: `std::string`

dengan `getline` tidak memiliki batas tetap. Pelajaran Bab 6 tentang larik, bahwa C++ tidak pernah memeriksa indeks untuk Anda, adalah alasan keberadaan seluruh golongan *bug* ini.

8.6.7 Uji diri

> **UJI DIRI.** Apa yang dicetak **Kode 8.10** untuk baris berisi satu kata, `queue`, dan untuk baris kosong? Berapa simpul yang dibuat pada masing-masing kasus? Lalu: jika `pushFront` diganti dengan `backInsertion` dari Bab 7, apa yang akan dicetak untuk contoh masukan?
>
> *Jawaban.* `queue`, dari list berisi satu simpul; baris kosong tidak menghasilkan kata dan simpul apa pun, dan hanya mencetak akhir baris. Dengan `backInsertion`, kata-kata mempertahankan urutannya, sehingga program mencetak kalimatnya tanpa perubahan, `practice makes a man perfect`: pembalikan datang *hanya* dari penyisipan di depan.

8.7 Soal 8.5 — Membuang Nilai Ganda dari Sebuah List

8.7.1 Soalnya

> **SOAL.** Complete the following C++ code, where `remove_duplicate()` is a function that removes the nodes with the same data from the list. Sample input: 1, 2, 3, 4, 1, 5, 6, 3, 2, −1 (one value per line). Sample output: the list before the removal, `1 2 3 4 1 5 6 3 2 -1`, and after it, `1 2 3 4 5 6`.
>
> *Dengan kata lain:* lengkapilah kode C++ berikut, dengan `remove_duplicate()` sebagai fungsi yang membuang simpul-simpul berdata sama dari list; keluarannya adalah list sebelum dan sesudah pembuangan.

`main` yang diberikan adalah:

```
int main() {
    linkedList *myList = new linkedList;
    myList->_head = NULL;
    int data;
    do {
        cin >> data;
        slist_pushBack(myList, data);
    } while (data != -1);
    print_linkedList(myList);
    remove_duplicate(myList);
    print_linkedList(myList);
    return 0;
}
```

8.7.2 Dengan kata-kata kita sendiri

List dibangun dari masukan secara berurutan. Kita harus menghapus setiap simpul yang nilainya sudah muncul *lebih awal* di list, dan mempertahankan kemunculan pertama setiap nilai, sehingga yang tersisa tetap dalam urutan aslinya: 1, 2, 3, 4, 5, 6.

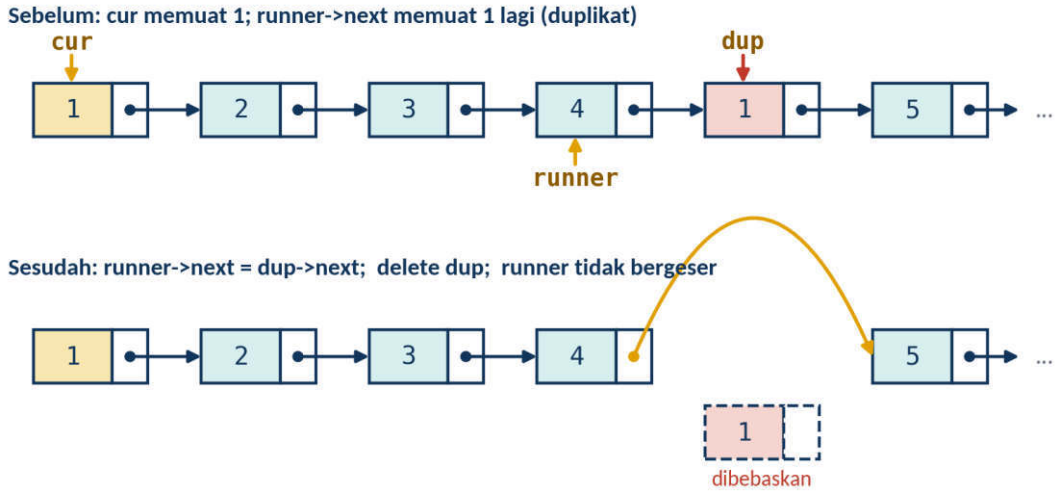
Satu rincian pada contoh itu layak diperhatikan dengan saksama, karena tampak seperti pertentangan. Seperti perulangan pada Soal 8.2, perulangan pembacaan yang diberikan adalah `do`–`while`, sehingga penanda akhir −1 di-*push* ke list sebelum perulangan menyadarinya; itulah sebabnya baris pertama yang dicetak berakhir dengan `-1`. Tetapi baris kedua tidak memuat −1, padahal −1 hanya muncul sekali sehingga bukan nilai ganda. Kedua baris itu hanya konsisten jika penanda akhir juga dibuang. Cara jujur untuk menyatakannya adalah: −1 *bukan data, melainkan isyarat bahwa data sudah berakhir*, dan ia tidak semestinya bertahan di list yang sudah dibersihkan. Program kita mempertahankan perulangan pembacaan yang diberikan, agar baris

pertama cocok, dan sesudah `remove_duplicate` ia membuang simpul terakhir, yaitu penanda itu, dengan `slist_popBack` kecil (`backDeletion` dari Bab 7). `remove_duplicate` sendiri hanya melakukan apa yang disebut namanya. Rancangan yang lebih bersih tidak akan pernah menyimpan penanda itu sejak awal; Soal Latihan 8.12 meminta versi tersebut.

8.7.3 Dengan tangan, selangkah demi selangkah

Kita memakai dua pointer. `cur` berjalan menyusuri list, satu simpul setiap kali; ia menandai nilai yang sedang kita cari. Untuk setiap posisi `cur`, pointer kedua, `runner`, mulai dari `cur` dan berjalan sampai ujung list, setiap kali melihat `runner->next`. Ketika `runner->next` memuat nilai yang sama dengan `cur`, simpul itu adalah salinan yang datang belakangan, dan kita menghapusnya.

Mengapa `runner` melihat `runner->next` dan bukan dirinya sendiri? Dengan alasan yang kita temui pada Soal 8.3: untuk menghapus simpul dari singly linked list, kita harus mengubah pointer `next` milik simpul *sebelumnya*, sehingga pointer yang bekerja harus berdiri di pendahulu. **Gambar 8.6** memperlihatkan saat `cur` berada di angka 1 yang pertama dan `runner` berada di 4, yang penerusnya adalah angka 1 yang kedua.



Gambar 8.6 Menghapus salinan yang datang belakangan: `runner` berdiri di pendahulu duplikat, menautkan melompatinya, dan baru sesudah itu membebaskannya; `runner` tidak bergeser, karena penerusnya yang baru belum diperiksa.

Penghapusan terdiri atas tiga pernyataan, dan urutannya penting:

1. `SListNode *dup = runner->next;` memberi nama pada simpul yang akan dihapus, agar kita masih dapat menjangkaunya.
2. `runner->next = dup->next;` menautkan melompatinya: 4 kini menunjuk ke 5. List tidak lagi memuat `dup`.
3. `delete dup;` membebaskan memorinya. Melakukannya *sebelum* langkah 2 membuat langkah 2 membaca `dup->next` dari memori yang sudah dibebaskan, sebuah *use after free*, kesalahan pada penyelesaian keliru Soal 8.12.

Sesudah penghapusan, `runner` harus **tetap di tempatnya**, karena `next`-nya kini adalah simpul yang belum diperiksa. Sesudah perbandingan yang tidak menghapus apa pun, `runner` maju satu simpul. Pada contoh masukan, seluruh prosesnya singkat:

- `cur` = 1: `runner` menemukan 1 di posisi kelima dan menghapusnya. List menjadi 1 2 3 4 5 6 3 2 −1.
- `cur` = 2: menghapus 2 di dekat ujung. List menjadi 1 2 3 4 5 6 3 −1.

- `cur` = 3: menghapus 3 yang terakhir. List menjadi 1 2 3 4 5 6 −1.
- `cur` = 4, 5, 6, −1: tidak ada yang dihapus.
- `slist_popBack` menghapus penanda akhir. List menjadi 1 2 3 4 5 6.

Biaya. Untuk setiap posisi `cur`, `runner` melihat setiap simpul sesudahnya. Dengan n simpul dan tanpa duplikat sama sekali (kasus terburuk, karena tidak ada yang memendek), banyaknya perbandingan adalah $(n-1)+(n-2)+\cdots+1+0$:

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2) \text{ perbandingan, dan } O(1) \text{ memori tambahan.} \tag{8.4}$$

Jumlahnya sama dengan Persamaan (8.3), dengan alasan yang sama: setiap elemen dibandingkan dengan setiap elemen di satu sisinya. Ada cara yang lebih cepat, yang menukar memori dengan waktu. Telusuri list sekali dan simpan sebuah *himpunan* nilai yang sudah terlihat (`std::unordered_set` di C++, `HashSet` di Java, `set` di Python); simpul yang nilainya sudah ada di himpunan itu dihapus. Setiap pencarian di himpunan memakan waktu rata-rata $O(1)$, sehingga penelusuran berbiaya waktu $O(n)$ dan memori tambahan $O(n)$. Bagaimana *hash set* mencapai $O(1)$ adalah pokok bahasan Bab 15; untuk list berisi sepuluh nilai, metode kuadratik sudah sangat memadai, dan ia tidak memerlukan apa pun di luar Bab 7.

8.7.4 Penyelesaian C++

Kode 8.13 mempertahankan nama-nama dan `main` yang diberikan, dan hanya menambahkan pemanggilan `slist_popBack`, satu baris terakhir berisi banyaknya simpul, dan pembersihan memori.

Kode 8.13 `p805_remove_duplicates.cpp`: Soal 8.5 dengan pembuangan dua pointer, ditambah `slist_popBack` yang membuang penanda akhir.

```cpp
// p805_remove_duplicates.cpp - Soal 8.5: membuang simpul yang nilainya sudah muncul sebelumnya.
#include <iostream>
using namespace std;

typedef struct snode_t {
    int data;
    struct snode_t *next;
} SListNode;

typedef struct slist_t {
    unsigned _size;
    SListNode *_head;
} linkedList;

bool slist_isEmpty(linkedList *myList) {
    return (myList->_head == nullptr);
}

void slist_pushBack(linkedList *myList, int value) {
    SListNode *newNode = new SListNode{value, nullptr};
    if (slist_isEmpty(myList)) {
        myList->_head = newNode;
    } else {
        SListNode *last = myList->_head;
        while (last->next != nullptr) {
            last = last->next;
        }
        last->next = newNode;
    }
    myList->_size++;
}

// Membuang simpul terakhir (dipakai di bawah untuk penanda akhir -1).
void slist_popBack(linkedList *myList) {
    if (slist_isEmpty(myList)) return;
    if (myList->_head->next == nullptr) {  // hanya satu simpul
        delete myList->_head;
        myList->_head = nullptr;
    } else {
        SListNode *aux = myList->_head;    // berhenti di simpul SEBELUM yang terakhir
        while (aux->next->next != nullptr) {
            aux = aux->next;
        }
        delete aux->next;                  // bebaskan simpul terakhir
        aux->next = nullptr;               // aux kini simpul terakhir
    }
```

Kode 8.13 (Lanjutan) *p805_remove_duplicates.cpp: Soal 8.5 dengan pembuangan dua pointer, ditambah slist_popBack yang membuang penanda akhir.*

```
    myList->_size--;
}

void print_linkedList(linkedList *myList) {
    for (SListNode *p = myList->_head; p != nullptr; p = p->next) {
        cout << p->data << (p->next != nullptr ? " " : "");
    }
    cout << "\n";
}

// Mempertahankan simpul PERTAMA setiap nilai dan menghapus setiap simpul berikutnya yang
nilainya sama.
void remove_duplicate(linkedList *myList) {
    for (SListNode *cur = myList->_head; cur != nullptr; cur = cur->next) {
        SListNode *runner = cur;           // runner->next adalah simpul yang sedang
diperiksa
        while (runner->next != nullptr) {
            if (runner->next->data == cur->data) {
                SListNode *dup = runner->next;
                runner->next = dup->next;  // 1. lepaskan: lompati duplikatnya
                delete dup;                // 2. baru kemudian bebaskan
                myList->_size--;           // runner tetap: next-nya yang BARU belum
diperiksa
            } else {
                runner = runner->next;     // tidak ada yang dibuang: maju
            }
        }
    }
}

void slist_destroy(linkedList *myList) {
    while (!slist_isEmpty(myList)) {
        SListNode *gone = myList->_head;
        myList->_head = gone->next;
        delete gone;
    }
    myList->_size = 0;
}
```

Kode 8.13 (Lanjutan) *p805_remove_duplicates.cpp: Soal 8.5 dengan pembuangan dua pointer, ditambah slist_popBack yang membuang penanda akhir.*

```
int main() {
    linkedList *myList = new linkedList;
    myList->_size = 0;
    myList->_head = nullptr;
    int data;
    do {
        if (!(cin >> data)) break;         // masukan berakhir tanpa -1
        slist_pushBack(myList, data);      // catatan: penanda akhir -1 ikut tersimpan
    } while (data != -1);
    print_linkedList(myList);
    remove_duplicate(myList);
    slist_popBack(myList);                 // buang penanda akhir, karena ia bukan data
    print_linkedList(myList);
    cout << "nodes left: " << myList->_size << "\n";
    slist_destroy(myList);
    delete myList;
    return 0;
}
```

```
$ ./p805 < p805.txt
1 2 3 4 1 5 6 3 2 -1
1 2 3 4 5 6
nodes left: 6
$ ./p805 < p805_check.txt
7 7 7 2 7 2 -1
7 2
nodes left: 2
```

Baris demi baris. `slist_pushBack` adalah `backInsertion` dari Bab 7 untuk list yang hanya menyimpan head: list kosong menerima simpul baru sebagai head-nya; jika tidak, pointer bantu berjalan ke simpul terakhir dan menautkan simpul baru sesudahnya. `slist_popBack` memiliki dua kasus `backDeletion` dari Bab 7: list satu simpul menjadi kosong, dan list yang lebih panjang ditelusuri sampai `aux` berdiri di simpul *sebelum* yang terakhir (`aux->next->next == nullptr`), yang penerusnya dibebaskan dan `next`-nya menjadi `nullptr`. `print_linkedList` mencetak spasi hanya di antara nilai. `remove_duplicate` adalah prosedur Subbab 8.7.3; perhatikan bahwa cabang `else` adalah satu-satunya tempat `runner` bergerak. Terakhir, `main` juga mencetak ukurannya, sebagai pemeriksaan silang bahwa penghitung `_size` tetap mutakhir melalui setiap penghapusan, dan membebaskan semuanya dengan `slist_destroy`. Masukan kedua, `7 7 7 2 7 2 -1`, memuat tiga salinan berderet, kasus yang menjebak penyelesaian keliru di bawah.

8.7.5 Penyelesaian Java dan Python

Di Java dan Python, menghapus simpul hanyalah langkah 2, `runner.next = runner.next.next`: begitu tidak ada lagi yang merujuk ke simpul yang dilepas, *garbage collector* mengambilnya kembali, sehingga tidak ada langkah 3 dan tidak ada cara untuk memakai simpul sesudah dibebaskan. Selebihnya algoritmenya identik, dan **Kode 8.14** serta **Kode 8.15** mencetak baris-baris yang sama.

Kode 8.14 `P805RemoveDuplicates.java`: Soal 8.5 dalam Java.

```java
// P805RemoveDuplicates.java - Soal 8.5: membuang simpul yang nilainya sudah muncul sebelumnya.
public class P805RemoveDuplicates {

    static class Node {
        int data;
        Node next;

        Node(int data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    static class LinkedList {
        int size;
        Node head;
    }

    static boolean isEmpty(LinkedList list) {
        return list.head == null;
    }

    static void pushBack(LinkedList list, int value) {
        Node newNode = new Node(value, null);
        if (isEmpty(list)) {
            list.head = newNode;
        } else {
            Node last = list.head;
            while (last.next != null) {
                last = last.next;
            }
            last.next = newNode;
        }
        list.size++;
    }
```

Kode 8.14 (Lanjutan) *P805RemoveDuplicates.java: Soal 8.5 dalam Java.*

```
    // Membuang simpul terakhir (dipakai di bawah untuk penanda akhir -1).
    static void popBack(LinkedList list) {
        if (isEmpty(list)) return;
        if (list.head.next == null) {          // hanya satu simpul
            list.head = null;
        } else {
            Node aux = list.head;              // berhenti di simpul SEBELUM yang terakhir
            while (aux.next.next != null) {
                aux = aux.next;
            }
            aux.next = null;                   // aux kini simpul terakhir
        }
        list.size--;
    }

    static void print(LinkedList list) {
        StringBuilder out = new StringBuilder();
        for (Node p = list.head; p != null; p = p.next) {
            out.append(p.data).append(p.next != null ? " " : "");
        }
        System.out.println(out);
    }

    // Mempertahankan simpul PERTAMA setiap nilai dan melepas setiap simpul berikutnya yang
nilainya sama.
    static void removeDuplicate(LinkedList list) {
        for (Node cur = list.head; cur != null; cur = cur.next) {
            Node runner = cur;                 // runner.next adalah simpul yang sedang
diperiksa
            while (runner.next != null) {
                if (runner.next.data == cur.data) {
                    runner.next = runner.next.next;   // lompati duplikatnya
                    list.size--;                       // runner tetap: next-nya yang BARU
belum diperiksa
                } else {
                    runner = runner.next;              // tidak ada yang dibuang: maju
                }
            }
        }
    }
```

Kode 8.14 (Lanjutan) *P805RemoveDuplicates.java: Soal 8.5 dalam Java.*

```
    public static void main(String[] args) throws java.io.IOException {
        String text = new String(System.in.readAllBytes()).trim();
        String[] tok = text.isEmpty() ? new String[0] : text.split("\\s+");
        LinkedList myList = new LinkedList();
        int i = 0;
        int data;
        do {
            if (i >= tok.length) break;        // masukan berakhir tanpa -1
            data = Integer.parseInt(tok[i++]);
            pushBack(myList, data);            // catatan: penanda akhir -1 ikut tersimpan
        } while (data != -1);
        print(myList);
        removeDuplicate(myList);
        popBack(myList);                       // buang penanda akhir, karena ia bukan data
        print(myList);
        System.out.println("nodes left: " + myList.size);
    }
}
```

Kode 8.15 `p805_remove_duplicates.py`: Soal 8.5 dalam Python.

```python
# p805_remove_duplicates.py - Soal 8.5: membuang simpul yang nilainya sudah muncul sebelumnya.
import sys


class Node:
    __slots__ = ("data", "next")

    def __init__(self, data, next=None):
        self.data = data
        self.next = next


class LinkedList:
    def __init__(self):
        self.size = 0
        self.head = None


def push_back(lst, value):
    new_node = Node(value)
    if lst.head is None:
        lst.head = new_node
    else:
        last = lst.head
        while last.next is not None:
            last = last.next
        last.next = new_node
    lst.size += 1


def pop_back(lst):
    """Membuang simpul terakhir (dipakai di bawah untuk penanda akhir -1)."""
    if lst.head is None:
        return
    if lst.head.next is None:           # hanya satu simpul
        lst.head = None
    else:
        aux = lst.head                  # berhenti di simpul SEBELUM yang terakhir
        while aux.next.next is not None:
            aux = aux.next
        aux.next = None                 # aux kini simpul terakhir
    lst.size -= 1
```

Kode 8.15 (Lanjutan) *p805_remove_duplicates.py: Soal 8.5 dalam Python.*

```
def print_list(lst):
    values = []
    p = lst.head
    while p is not None:
        values.append(str(p.data))
        p = p.next
    print(" ".join(values))


def remove_duplicate(lst):
    """Mempertahankan simpul PERTAMA setiap nilai dan melepas setiap simpul berikutnya yang
nilainya sama."""
    cur = lst.head
    while cur is not None:
        runner = cur                    # runner.next adalah simpul yang sedang diperiksa
        while runner.next is not None:
            if runner.next.data == cur.data:
                runner.next = runner.next.next  # lompati duplikatnya
                lst.size -= 1                   # runner tetap: next-nya yang BARU belum
diperiksa
            else:
                runner = runner.next            # tidak ada yang dibuang: maju
        cur = cur.next


def main():
    my_list = LinkedList()
    for t in sys.stdin.read().split():
        data = int(t)
        push_back(my_list, data)        # catatan: penanda akhir -1 ikut tersimpan
        if data == -1:
            break
    print_list(my_list)
    remove_duplicate(my_list)
    pop_back(my_list)                   # buang penanda akhir, karena ia bukan data
    print_list(my_list)
    print("nodes left:", my_list.size)


if __name__ == "__main__":
    main()
```

8.7.6 Penyelesaian keliru yang menggoda

Cara yang wajar untuk menulis penelusuran adalah “lihat simpul berikutnya, lalu maju”, setiap kali:

```
  if (runner->next->data == cur->data) {
      SListNode *dup = runner->next;
      runner->next = dup->next;
      delete dup;
  }
  runner = runner->next;          // KELIRU: tetap maju sesudah penghapusan
```

Cara ini lolos pada contoh masukan, karena di sana tidak ada dua salinan sebuah nilai yang bersebelahan. Pada `7 7 7 2 7 2 -1`, program `wrong/p805_wrong.cpp` mencetak

```
7 7 7 2 7 2 -1
7 7 2
nodes left: 3
```

Ikutilah jalannya: `cur` berada di angka 7 yang pertama, demikian pula `runner`. Angka 7 yang kedua dihapus, dan `runner->next` kini adalah angka 7 yang ketiga, tetapi baris yang keliru

memajukan `runner` ke atasnya tanpa membandingkannya dengan `cur`. Angka 7 yang ketiga lolos. Aturannya sederhana begitu diucapkan: *sesudah penghapusan, simpul di balik `runner->next` adalah simpul baru, jadi periksalah sebelum maju*. Kasus tepi yang membongkar kesalahan ini adalah "dua salinan bersebelahan", yang tercantum dalam daftar Langkah 5 metode kita justru karena alasan ini.

8.7.7 Uji diri

> **UJI DIRI.** Berapa kali perbandingan `runner->next->data == cur->data` dilakukan `remove_duplicate` pada list contoh berisi sepuluh simpul (termasuk -1)? Hitunglah dengan tangan.
>
> *Jawaban.* Dengan `cur` di simpul pertama, `runner` memeriksa kesembilan simpul sesudahnya, dan satu di antaranya dihapus, sehingga tersisa 9 simpul. Dengan `cur` di simpul kedua, ada 7 simpul sesudahnya (satu dihapus, tersisa 8); di simpul ketiga, 5 simpul sesudahnya (satu dihapus, tersisa 7); lalu simpul keempat, kelima, keenam, dan ketujuh memiliki 3, 2, 1, dan 0 simpul sesudahnya. Totalnya $9+7+5+3+2+1+0=27$, kurang dari $\frac{10\cdot 9}{2}=45$ pada Persamaan (8.4), karena setiap penghapusan memperpendek setiap pemindaian berikutnya.

8.8 Soal Baru yang Dikerjakan: Stack dan Queue

Kelima soal sejauh ini datang sebagai satu rangkaian. Tujuh soal berikutnya baru, dan masing-masing membidik satu gagasan Bab 5 sampai Bab 7 yang hanya disentuh ringan oleh rangkaian itu. Soal-soal ini mengikuti metode yang sama, dengan langkah yang sedikit lebih cepat, memakai label-label Subbab 8.2 yang dicetak tebal.

8.8.1 Soal 8.6 — Mencocokkan tanda kurung

> **SOAL.** Tulislah fungsi yang memutuskan apakah tanda kurung `()`, `[]`, dan `{}` dalam sebaris teks seimbang: setiap kurung buka ditutup oleh kurung tutup yang sejenis, dan pasangan-pasangannya bersarang dengan benar. Jika tidak seimbang, sebutkan apa yang salah dan di posisi mana (karakter dihitung mulai dari 1). Ujilah pada `(a+b)*[c-d]`, `{[()()]}`, `([)]`, `((x)x)]`, dan baris kosong.

Dengan kata-kata kita sendiri. Membaca dari kiri ke kanan, setiap kurung tutup harus cocok dengan kurung buka *terbaru* yang masih terbuka. "Terbaru yang masih terbuka" berarti terakhir masuk, pertama keluar: stack. *Push* setiap kurung buka (beserta posisinya, untuk pesan kesalahan); pada setiap kurung tutup, puncak stack harus pasangannya, lalu kita *pop*. Ada tepat tiga cara untuk gagal: kurung tutup datang ketika stack kosong (ia tidak punya pasangan), kurung tutup tidak cocok dengan puncak (pasangan-pasangannya bersilangan), atau teks berakhir dengan stack yang tidak kosong (ada yang tidak pernah ditutup).

Dengan tangan. Untuk `([)]`: posisi 1 `(` di-*push*; posisi 2 `[` di-*push*, dan stack berisi `(`, `[` dengan `[` di puncak; posisi 3 adalah `)`, tetapi puncaknya `[`, sehingga `)` tidak dapat menutupnya, dan kita melaporkan `')' at 3 cannot close '[' from 2`. Untuk `{[()()]}`, stack tumbuh sampai tiga kurung lalu menyusut kembali menjadi kosong, setiap kurung tutup bertemu pasangannya di puncak, sehingga baris itu seimbang. Untuk `((x)`, `)` di posisi 4 menutup `(` dari posisi 2, dan di akhir `(` dari posisi 1 masih berada di stack.

Rancangan. Stack-nya adalah stack larik seperti pada Soal 8.1, tetapi kapasitasnya dipilih ketika barisnya sudah diketahui: baris berisi L karakter tidak mungkin memuat lebih dari L kurung buka. C++ mengalokasikan larik semacam itu dengan `new char[capacity]` dan wajib

melepaskannya dengan `delete[]` (dengan kurung siku, karena yang dilepas adalah larik). Kita menyerahkan pelepasan itu kepada *destruktor* stack, sehingga ia terjadi secara otomatis ketika `check` selesai, pada setiap jalur, termasuk ketiga `return` dini pada **Kode 8.16**. Karena objek itu memiliki lariknya, menyalinnya akan menghasilkan dua objek yang sama-sama merasa memiliki larik yang sama dan sama-sama menghapusnya; dua baris `= delete` melarang penyalinan, sehingga kesalahan itu bahkan tidak dapat terkompilasi.

Kode 8.16 `p806_brackets.cpp`: Soal 8.6, pencocokan tanda kurung dengan stack larik yang kapasitasnya dipilih saat program berjalan.

```
// p806_brackets.cpp - Soal 8.6: memeriksa tanda kurung sebuah ekspresi dengan stack.
#include <iostream>
#include <string>
using namespace std;

// Stack larik yang kapasitasnya ditentukan saat program berjalan (new[] / delete[]).
struct BracketStack {
    char *symbol;     // tanda kurung buka
    int *where;       // posisinya dalam teks (mulai dari 1)
    int top;          // -1 berarti kosong

    explicit BracketStack(int capacity)
        : symbol(new char[capacity]), where(new int[capacity]), top(-1) {}
    ~BracketStack() {
        delete[] symbol;
        delete[] where;
    }
    BracketStack(const BracketStack &) = delete;             // tidak boleh disalin:
    BracketStack &operator=(const BracketStack &) = delete;  // satu pemilik untuk setiap
larik

    bool isEmpty() const { return top < 0; }
    void push(char c, int pos) {
        ++top;
        symbol[top] = c;
        where[top] = pos;
    }
    void pop() { --top; }
};

char partnerOf(char close) {
    if (close == ')') return '(';
    if (close == ']') return '[';
    return '{';
}

// Mengembalikan "balanced" atau kalimat yang menyebut apa yang salah, dan di mana.
string check(const string &text) {
    BracketStack st(static_cast<int>(text.size()) + 1);  // tak pernah ada kurung buka
lebih banyak daripada karakter
    for (size_t i = 0; i < text.size(); i++) {
        char c = text[i];
        int pos = static_cast<int>(i) + 1;
        if (c == '(' || c == '[' || c == '{') {
            st.push(c, pos);
        } else if (c == ')' || c == ']' || c == '}') {
            if (st.isEmpty()) {
                return string("'") + c + "' at " + to_string(pos) + " has no opening
partner";
```

Kode 8.16 (Lanjutan) *p806_brackets.cpp: Soal 8.6, pencocokan tanda kurung dengan stack larik yang kapasitasnya dipilih saat program berjalan.*

```
            }
            if (st.symbol[st.top] != partnerOf(c)) {
                return string("'") + c + "' at " + to_string(pos) + " cannot close '" +
                       st.symbol[st.top] + "' from " + to_string(st.where[st.top]);
            }
            st.pop();
        }
    }
    if (!st.isEmpty()) {
        return string("'") + st.symbol[st.top] + "' at " + to_string(st.where[st.top]) +
               " is never closed";
    }
    return "balanced";
}

int main() {
    const string tests[] = {"(a+b)*[c-d]", "{[()()]}", "([)]", "((x)", "x)]", ""};
    for (const string &t : tests) {
        cout << "\"" << t << "\" -> " << check(t) << "\n";
    }
    return 0;
}
```

```
"(a+b)*[c-d]" -> balanced
"{[()()]}" -> balanced
"([)]" -> ')' at 3 cannot close '[' from 2
"((x)" -> '(' at 1 is never closed
"x)]" -> ')' at 2 has no opening partner
"" -> balanced
```

Program Java dan Python, **Kode 8.17** dan **Kode 8.18**, mencetak keenam baris yang sama. Larik Java juga dibuat dengan `new char[capacity]`, tetapi dibebaskan oleh *garbage collector*; Python memakai list berpanjang tetap dan kamus `PARTNER` yang memetakan setiap kurung tutup ke kurung buka pasangannya.

Kode 8.17 `P806Brackets.java`: Soal 8.6 dalam Java.

```java
// P806Brackets.java - Soal 8.6: memeriksa tanda kurung sebuah ekspresi dengan stack.
public class P806Brackets {

    // Stack larik yang kapasitasnya ditentukan saat program berjalan.
    static class BracketStack {
        final char[] symbol;       // tanda kurung buka
        final int[] where;         // posisinya dalam teks (mulai dari 1)
        int top = -1;              // -1 berarti kosong

        BracketStack(int capacity) {
            symbol = new char[capacity];
            where = new int[capacity];
        }

        boolean isEmpty() { return top < 0; }

        void push(char c, int pos) {
            ++top;
            symbol[top] = c;
            where[top] = pos;
        }

        void pop() { --top; }
    }

    static char partnerOf(char close) {
        if (close == ')') return '(';
        if (close == ']') return '[';
        return '{';
    }

    // Mengembalikan "balanced" atau kalimat yang menyebut apa yang salah, dan di mana.
    static String check(String text) {
        BracketStack st = new BracketStack(text.length() + 1);
        for (int i = 0; i < text.length(); i++) {
            char c = text.charAt(i);
            int pos = i + 1;
            if (c == '(' || c == '[' || c == '{') {
                st.push(c, pos);
            } else if (c == ')' || c == ']' || c == '}') {
                if (st.isEmpty()) {
                    return "'" + c + "' at " + pos + " has no opening partner";
                }
                if (st.symbol[st.top] != partnerOf(c)) {
                    return "'" + c + "' at " + pos + " cannot close '"
                           + st.symbol[st.top] + "' from " + st.where[st.top];
```

Kode 8.17 (Lanjutan) *P806Brackets.java: Soal 8.6 dalam Java.*

```
                }
                st.pop();
            }
        }
        if (!st.isEmpty()) {
            return "'" + st.symbol[st.top] + "' at " + st.where[st.top] + " is never
closed";
        }
        return "balanced";
    }

    public static void main(String[] args) {
        String[] tests = {"(a+b)*[c-d]", "{[()()]}", "([)]", "((x)", "x)]", ""};
        for (String t : tests) {
            System.out.println("\"" + t + "\" -> " + check(t));
        }
    }
}
```

Kode 8.18 `p806_brackets.py`: Soal 8.6 dalam Python.

```python
# p806_brackets.py - Soal 8.6: memeriksa tanda kurung sebuah ekspresi dengan stack.
PARTNER = {")": "(", "]": "[", "}": "{"}


class BracketStack:
    """Stack larik yang kapasitasnya ditentukan saat program berjalan."""

    def __init__(self, capacity):
        self.symbol = [""] * capacity   # tanda kurung buka
        self.where = [0] * capacity     # posisinya dalam teks (mulai dari 1)
        self.top = -1                   # -1 berarti kosong

    def is_empty(self):
        return self.top < 0

    def push(self, c, pos):
        self.top += 1
        self.symbol[self.top] = c
        self.where[self.top] = pos

    def pop(self):
        self.top -= 1


def check(text):
    """Mengembalikan "balanced" atau kalimat yang menyebut apa yang salah, dan di mana."""
    st = BracketStack(len(text) + 1)
    for i, c in enumerate(text):
        pos = i + 1
        if c in "([{":
            st.push(c, pos)
        elif c in ")]}":
            if st.is_empty():
                return f"'{c}' at {pos} has no opening partner"
            if st.symbol[st.top] != PARTNER[c]:
                return f"'{c}' at {pos} cannot close '{st.symbol[st.top]}' from
{st.where[st.top]}"
            st.pop()
    if not st.is_empty():
        return f"'{st.symbol[st.top]}' at {st.where[st.top]} is never closed"
    return "balanced"


def main():
    for t in ["(a+b)*[c-d]", "{[()()]}", "([)]", "((x)", "x)]", ""]:
        print(f'"{t}" -> {check(t)}')
```

Kode 8.18 (Lanjutan) *p806_brackets.py: Soal 8.6 dalam Python.*

```python

if __name__ == "__main__":
    main()
```

Biaya. Setiap karakter menyebabkan paling banyak satu *push* atau satu *pop*, sehingga pemeriksaannya berbiaya waktu $\Theta(L)$ dan memori $O(L)$ untuk baris berisi L karakter.

Penyelesaian keliru yang menggoda. Menghitung tampaknya sudah cukup: tambah satu untuk kurung buka, kurangi satu untuk kurung tutup, gagal jika hitungannya menjadi negatif, dan berhasil jika berakhir di nol. Program `wrong/p806_wrong.cpp` melakukan hal itu, dan mencetak

```
"(a+b)*[c-d]" -> balanced
"{[()()]}" -> balanced
"([)]" -> balanced
"((x)" -> not balanced
"x)]" -> not balanced
"(]" -> balanced
```

Sebuah penghitung tahu *berapa banyak* kurung yang terbuka, tetapi tidak tahu *yang mana*, sehingga ia menerima `([)]` dan bahkan `(]`. Dengan satu jenis kurung saja, penghitung sudah cukup; dengan beberapa jenis, urutan kurung yang terbuka menjadi penting, dan mengingat urutan adalah pekerjaan stack.

> **UJI DIRI.** Apa yang dikembalikan `check` untuk `{(a)[b]}}` dan untuk `[(])`? Telusurilah stack-nya.
>
> *Jawaban.* `{(a)[b]}}`: stack menjadi `{`, lalu `{(`, kembali ke `{`, lalu `{[`, kembali ke `{`, lalu kosong di posisi 8; `}` di posisi 9 mendapati stack kosong: `'}' at 9 has no opening partner`. `[(])`: `[` dan `(` di-*push*; `]` di posisi 3 bertemu `(` di puncak: `']' at 3 cannot close '(' from 2`.

8.8.2 Soal 8.7 — Queue sirkular untuk permintaan peminjaman

> **SOAL.** Pustaka Ceria menyimpan permintaan peminjaman ruang bacanya dalam sebuah queue berkapasitas empat permintaan, yang disimpan dalam larik. Implementasikan `enqueue`, `dequeue`, `isEmpty`, dan `isFull` sehingga slot yang dikosongkan oleh `dequeue` dapat dipakai lagi, dan perlihatkan keadaan larik sesudah setiap kelompok operasi untuk urutan berikut: *enqueue* `B1001` sampai `B1005`; *dequeue* dua kali; *enqueue* `B1005` dan `B1006`; *dequeue* semuanya.

Dengan kata-kata kita sendiri. Depan queue adalah tempat permintaan keluar, belakangnya tempat permintaan datang. Bab 5 memperlihatkan kesulitan queue larik yang paling sederhana, yang kedua ujungnya hanya bergerak ke kanan: sesudah beberapa operasi, ujung belakang mencapai sel terakhir, dan queue melaporkan "penuh" padahal sel-sel di depan kosong. Perbaikannya adalah membiarkan indeks-indeksnya *berputar kembali*: sesudah sel terakhir datang lagi sel 0, seakan-akan lariknya dibengkokkan menjadi cincin.

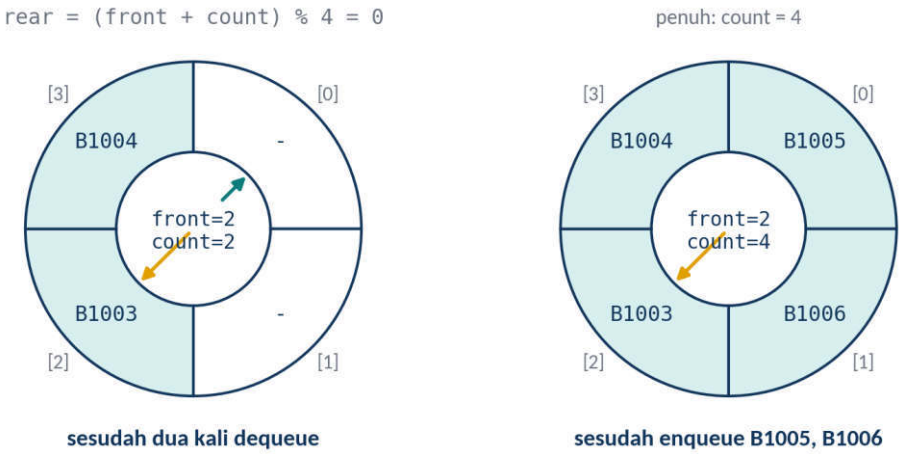
Rancangan. Kita menyimpan dua bilangan: `front`, indeks permintaan yang paling lama, dan `count`, banyaknya permintaan yang menunggu. Semua yang lain mengikuti dari keduanya. Slot kosong pertama berada `count` tempat sesudah `front`, berputar pada cincin berisi `CAPACITY` sel, dan itulah yang dikerjakan operator sisa bagi:

$$\begin{aligned} \text{rear} &= (\text{front} + \text{count}) \bmod \text{CAPACITY}, \qquad \text{front} \\ &\leftarrow (\text{front} + 1) \bmod \text{CAPACITY} \text{ sesudah dequeue.} \end{aligned} \tag{8.5}$$

Dengan `count` tersimpan, "kosong" adalah `count == 0` dan "penuh" adalah `count == CAPACITY`, dan keduanya tidak mungkin tertukar. (Queue yang hanya menyimpan `front` dan `rear` tidak dapat membedakan cincin penuh dari cincin kosong, karena pada kedua keadaan itu kedua indeksnya sama; queue semacam itu selalu membiarkan satu sel kosong untuk membedakannya. Menyimpan `count` hanya memerlukan satu bilangan bulat dan menghindarkan teka-teki itu.)

Dengan tangan. Empat kali *enqueue* mengisi sel 0 sampai 3 (rear = 0,1,2,3) dan `count` menjadi 4; yang kelima ditolak. Dua kali *dequeue* mengembalikan `B1001` dan `B1002` dan menggeser `front` ke 1 lalu 2, dengan `count` = 2. Kini Persamaan (8.5) memberi slot kosong berikutnya $(2 + 2) \bmod 4 = 0$: `B1005` masuk ke sel 0, sel yang ditinggalkan `B1001`, dan `B1006` ke $(2 + 3) \bmod 4 = 1$. Larik kini berbunyi `B1005 B1006 B1003 B1004`, yang tampak tidak berurutan, tetapi urutan queue dimulai

dari `front` = 2 dan berputar: `B1003`, `B1004`, `B1005`, `B1006`. **Gambar 8.7** menggambar kedua saat itu sebagai cincin, dan **Kode 8.19** menjalankan seluruh urutannya.



Gambar 8.7 Queue sirkular Soal 8.7 yang digambar sebagai cincin. Panah jingga adalah `front`, panah hijau toska adalah slot kosong berikutnya, `rear`. Sesudah dua kali *dequeue* (kiri), permintaan berikutnya masuk ke sel 0; sesudah dua kali *enqueue* lagi (kanan), cincin penuh, dan urutan queue dimulai di sel 2 lalu berputar.

Kode 8.19 `p807_circular_queue.cpp`: Soal 8.7, queue larik sirkular dengan `front` dan `count`.

```cpp
// p807_circular_queue.cpp - Soal 8.7: queue larik sirkular untuk permintaan peminjaman.
#include <iostream>
#include <string>
using namespace std;

const int CAPACITY = 4;

struct CircularQueue {
    string slot[CAPACITY];
    int front = 0;     // indeks permintaan yang paling lama
    int count = 0;     // banyaknya permintaan yang menunggu

    bool isEmpty() const { return count == 0; }
    bool isFull() const { return count == CAPACITY; }

    bool enqueue(const string &id) {
        if (isFull()) return false;
        int rear = (front + count) % CAPACITY;   // slot kosong pertama, berputar kembali
ke awal
        slot[rear] = id;
        count++;
        return true;
    }

    string dequeue() {                           // prasyarat: !isEmpty()
        string id = slot[front];
        slot[front] = "-";                       // hanya agar gambarnya lebih jelas
        front = (front + 1) % CAPACITY;          // yang paling lama berikutnya, berputar
kembali ke awal
        count--;
        return id;
    }

    void show() const {
        cout << "  slots [";
        for (int i = 0; i < CAPACITY; i++) {
            cout << (i ? " " : "") << (slot[i].empty() ? "-" : slot[i]);
        }
        cout << "]  front=" << front << " count=" << count << "\n";
    }
};
```

Kode 8.19 (Lanjutan) *p807_circular_queue.cpp: Soal 8.7, queue larik sirkular dengan front dan count.*

```
int main() {
    CircularQueue q;
    const string requests[] = {"B1001", "B1002", "B1003", "B1004", "B1005"};
    for (const string &r : requests) {
        bool ok = q.enqueue(r);
        cout << "enqueue " << r << (ok ? "" : " -> refused, the queue is full") << "\n";
    }
    q.show();
    cout << "dequeue -> " << q.dequeue() << "\n";
    cout << "dequeue -> " << q.dequeue() << "\n";
    q.show();
    cout << "enqueue B1005 -> " << (q.enqueue("B1005") ? "ok" : "refused") << "\n";
    cout << "enqueue B1006 -> " << (q.enqueue("B1006") ? "ok" : "refused") << "\n";
    q.show();
    cout << "served in order:";
    while (!q.isEmpty()) {
        cout << " " << q.dequeue();
    }
    cout << "\n";
    q.show();
    return 0;
}
```

```
enqueue B1001
enqueue B1002
enqueue B1003
enqueue B1004
enqueue B1005 -> refused, the queue is full
  slots [B1001 B1002 B1003 B1004]  front=0 count=4
dequeue -> B1001
dequeue -> B1002
  slots [- - B1003 B1004]  front=2 count=2
enqueue B1005 -> ok
enqueue B1006 -> ok
  slots [B1005 B1006 B1003 B1004]  front=2 count=4
served in order: B1003 B1004 B1005 B1006
  slots [- - - -]  front=2 count=0
```

`show` yang terakhir memastikan invarian sesudah queue kosong: `count` bernilai 0 dan `front` berhenti begitu saja di 2, dan itu tidak masalah, karena selama `count` bernilai 0 nilai `front` tidak berpengaruh. **Kode 8.20** dan **Kode 8.21** mencetak baris-baris yang sama.

Kode 8.20 `P807CircularQueue.java`: Soal 8.7 dalam Java.

```java
// P807CircularQueue.java - Soal 8.7: queue larik sirkular untuk permintaan peminjaman.
public class P807CircularQueue {

    static final int CAPACITY = 4;

    static class CircularQueue {
        final String[] slot = new String[CAPACITY];
        int front = 0;     // indeks permintaan yang paling lama
        int count = 0;     // banyaknya permintaan yang menunggu

        boolean isEmpty() { return count == 0; }
        boolean isFull() { return count == CAPACITY; }

        boolean enqueue(String id) {
            if (isFull()) return false;
            int rear = (front + count) % CAPACITY;   // slot kosong pertama, berputar
kembali ke awal
            slot[rear] = id;
            count++;
            return true;
        }

        String dequeue() {                           // prasyarat: !isEmpty()
            String id = slot[front];
            slot[front] = "-";                       // hanya agar gambarnya lebih jelas
            front = (front + 1) % CAPACITY;          // yang paling lama berikutnya,
berputar kembali ke awal
            count--;
            return id;
        }

        void show() {
            StringBuilder b = new StringBuilder("  slots [");
            for (int i = 0; i < CAPACITY; i++) {
                b.append(i > 0 ? " " : "").append(slot[i] == null ? "-" : slot[i]);
            }
            b.append("]  front=").append(front).append(" count=").append(count);
            System.out.println(b);
        }
    }
```

Kode 8.20 (Lanjutan) *P807CircularQueue.java: Soal 8.7 dalam Java.*

```
    public static void main(String[] args) {
        CircularQueue q = new CircularQueue();
        String[] requests = {"B1001", "B1002", "B1003", "B1004", "B1005"};
        for (String r : requests) {
            boolean ok = q.enqueue(r);
            System.out.println("enqueue " + r + (ok ? "" : " -> refused, the queue is
full"));
        }
        q.show();
        System.out.println("dequeue -> " + q.dequeue());
        System.out.println("dequeue -> " + q.dequeue());
        q.show();
        System.out.println("enqueue B1005 -> " + (q.enqueue("B1005") ? "ok" : "refused"));
        System.out.println("enqueue B1006 -> " + (q.enqueue("B1006") ? "ok" : "refused"));
        q.show();
        StringBuilder served = new StringBuilder("served in order:");
        while (!q.isEmpty()) {
            served.append(" ").append(q.dequeue());
        }
        System.out.println(served);
        q.show();
    }
}
```

Kode 8.21 `p807_circular_queue.py`: Soal 8.7 dalam Python.

```python
# p807_circular_queue.py - Soal 8.7: queue larik sirkular untuk permintaan peminjaman.
CAPACITY = 4


class CircularQueue:
    def __init__(self):
        self.slot = ["-"] * CAPACITY
        self.front = 0      # indeks permintaan yang paling lama
        self.count = 0      # banyaknya permintaan yang menunggu

    def is_empty(self):
        return self.count == 0

    def is_full(self):
        return self.count == CAPACITY

    def enqueue(self, request_id):
        if self.is_full():
            return False
        rear = (self.front + self.count) % CAPACITY    # slot kosong pertama, berputar
kembali ke awal
        self.slot[rear] = request_id
        self.count += 1
        return True

    def dequeue(self):                                 # prasyarat: not is_empty()
        request_id = self.slot[self.front]
        self.slot[self.front] = "-"                    # hanya agar gambarnya lebih jelas
        self.front = (self.front + 1) % CAPACITY       # yang paling lama berikutnya,
berputar kembali ke awal
        self.count -= 1
        return request_id

    def show(self):
        print(f"  slots [{' '.join(self.slot)}]  front={self.front} count={self.count}")


def main():
    q = CircularQueue()
    for r in ["B1001", "B1002", "B1003", "B1004", "B1005"]:
        ok = q.enqueue(r)
        print("enqueue", r + ("" if ok else " -> refused, the queue is full"))
    q.show()
    print("dequeue ->", q.dequeue())
    print("dequeue ->", q.dequeue())
    q.show()
    print("enqueue B1005 ->", "ok" if q.enqueue("B1005") else "refused")
    print("enqueue B1006 ->", "ok" if q.enqueue("B1006") else "refused")
```

Kode 8.21 (Lanjutan) *p807_circular_queue.py: Soal 8.7 dalam Python.*

```python
    q.show()
    served = []
    while not q.is_empty():
        served.append(q.dequeue())
    print("served in order:", " ".join(served))
    q.show()


if __name__ == "__main__":
    main()
```

Biaya. Setiap operasi hanya beberapa langkah aritmetika: $O(1)$, berapa pun kapasitasnya. Itulah inti cincin. Perbaikan lain dari Bab 5, yaitu menggeser semua butir yang menunggu satu sel ke kiri sesudah setiap *dequeue*, juga berhasil tetapi berbiaya $O(n)$ per *dequeue*.

Penyelesaian keliru yang menggoda. Queue linear pada Bab 5 menyimpan `front` dan `rear` dan menyatakan dirinya penuh ketika `rear` mencapai sel terakhir. Program `wrong/p807_wrong.cpp` mengisinya, melakukan *dequeue* dua kali, lalu mencoba lagi:

```
dequeue -> B1001
dequeue -> B1002
waiting: 2 of 4
enqueue B1005 -> refused
```

Dua dari empat sel kosong, dan queue menolak sebuah permintaan: keadaan "penuh semu". Pengujiannya, `rear == CAPACITY - 1`, menanyakan di mana ujung belakang *berada*, padahal pertanyaan yang benar adalah berapa butir yang *menunggu*.

> **UJI DIRI.** Dengan `CAPACITY` = 4, mulailah dari queue kosong dan lakukan: *enqueue* A, B, C; *dequeue*; *dequeue*; *enqueue* D, E, F; *dequeue*. Berikan `front`, `count`, dan keempat slot di akhir, dengan "-" untuk slot yang nilainya sudah di-*dequeue*.
>
> *Jawaban.* Sesudah A, B, C: slot `A B C -`, `front` 0, `count` 3. Dua kali *dequeue*: `- - C -`, `front` 2, `count` 1. D masuk ke $(2+1) \bmod 4 = 3$, E ke $(2+2) \bmod 4 = 0$, F ke $(2+3) \bmod 4 = 1$: `E F C D`, `count` 4. Satu kali *dequeue* mengembalikan C: slot `E F - D`, `front` 3, `count` 3. Urutan queue-nya D, E, F.

8.8.3 Soal 8.8 — Dari infiks ke postfiks, dan nilai sebuah ekspresi

> **SOAL.** Tulislah dua fungsi. `toPostfix` mengubah ekspresi aritmetika dalam notasi biasa (*infiks*), dengan operand berupa satu huruf atau satu angka, operator `+ - * /`, dan tanda kurung, menjadi notasi *postfiks*, yang setiap operatornya menyusul kedua operandnya. `evaluate` menghitung nilai ekspresi postfiks yang operandnya berupa angka. Ujilah keduanya pada `3+2*5`, `a+b*c-d`, `8-3-2`, `(1+2)*(7-3)/2`, dan `9-4/2*3`.

Dengan kata-kata kita sendiri. Inilah kalkulator ilmiah Bab 5, dilengkapi dengan tanda kurung. Dalam notasi postfiks, `3+2*5` menjadi `325*+`: tidak ada tanda kurung dan tidak ada aturan presedensi, karena posisi setiap operator menentukan kapan ia diterapkan. Hal itu membuat postfiks mudah dievaluasi mesin, dan pemikirannya tercurah pada pengubahan. Kedua langkah memakai stack; kali ini kita memakai stack pustaka alih-alih menulisnya sendiri: `std::stack` di C++, `ArrayDeque` yang dipakai sebagai stack di Java, dan list biasa di Python [9], [7], [8].

Pengubahan, dalam aturan. Bacalah ekspresi infiks dari kiri ke kanan. *Derajat* (presedensi) `*` dan `/` adalah 2, `+` dan `-` adalah 1, dan `(` di stack dihitung 0.

1. Operand langsung masuk ke keluaran.
2. `(` di-*push*.
3. `)` mengeluarkan operator ke keluaran sampai `(` pasangannya, yang lalu di-*pop* dan dibuang.
4. Operator lebih dulu mengeluarkan ke keluaran setiap operator di stack yang derajatnya **lebih besar atau sama dengan** derajatnya sendiri, lalu dirinya di-*push*.
5. Di akhir, semua yang tersisa di stack di-*pop* ke keluaran.

Prosedur ini adalah *algoritme shunting-yard* Dijkstra, dinamai menurut langsiran kereta api tempat gerbong-gerbong menunggu di jalur samping sampai dapat diberangkatkan dalam urutan yang benar [10].

Mengapa "lebih besar atau sama dengan" pada Aturan 4? Karena `-` dan `/` tidak asosiatif: `8-3-2` berarti $(8-3)-2=3$, bukan $8-(3-2)=7$. Ketika `-` kedua datang, `-` pertama harus meninggalkan stack *sebelum*-nya, agar diterapkan lebih dulu; karena itu operator dengan derajat

sama harus dikeluarkan. Dengan "lebih besar" saja, `-` pertama akan tetap di stack, keluarannya menjadi `832--`, dan nilainya 7.

Dengan tangan. **Tabel 8.6** mengubah `a+b*c-d`, contoh dari Bab 5.

Tabel 8.6 Mengubah `a+b*c-d` ke postfiks dengan aturan-aturan Soal 8.8. Stack menampung operator yang menunggu operand kanannya lengkap.

Dibaca	Aturan	Stack (puncak di kanan)	Keluaran sejauh ini
`a`	1		`a`
`+`	4: stack kosong, *push*	`+`	`a`
`b`	1	`+`	`ab`
`*`	4: derajat `+` (1) < 2, *push*	`+ *`	`ab`
`c`	1	`+ *`	`abc`
`-`	4: *pop* `*` ($2 \geq 1$), *pop* `+` ($1 \geq 1$), *push* `-`	`-`	`abc*+`
`d`	1	`-`	`abc*+d`
akhir	5: *pop* semuanya		`abc*+d-`

Evaluasi. Bacalah string postfiks dari kiri ke kanan dengan stack bilangan yang kosong. Angka di-*push*. Operator mem-*pop* dua bilangan, menerapkan dirinya, lalu mem-*push* hasilnya. Bilangan pertama yang di-*pop* adalah operand **kanan**, karena ia di-*push* paling akhir. Untuk `83-2-`: *push* 8, *push* 3; `-` mem-*pop* $y = 3$, lalu $x = 8$, dan mem-*push* $x - y = 5$; *push* 2; `-` mem-*pop* $y = 2$, $x = 5$ dan mem-*push* 3. Di akhir harus tersisa tepat satu bilangan, dan itulah nilainya. Kurang dari dua bilangan ketika bertemu operator, atau lebih dari satu di akhir, berarti string postfiksnya cacat, dan `evaluate` melempar eksepsi dengan sebuah pesan. **Kode 8.22** mengimplementasikan kedua fungsi dan mengujinya.

Kode 8.22 `p808_postfix.cpp`: Soal 8.8, pengubahan *shunting-yard* dan evaluasi postfiks dengan `std::stack`.

```cpp
// p808_postfix.cpp - Soal 8.8: infiks ke postfiks dengan stack, lalu evaluasi postfiksnya.
#include <cctype>
#include <iostream>
#include <stack>
#include <stdexcept>
#include <string>
using namespace std;

int degree(char op) {                       // "derajat" (presedensi) sebuah operator
    if (op == '*' || op == '/') return 2;
    if (op == '+' || op == '-') return 1;
    return 0;                               // '(' : lebih rendah daripada semua operator
}

// Operand berupa satu huruf atau satu angka; operatornya + - * / dan tanda kurung.
string toPostfix(const string &infix) {
    string postfix;
    stack<char> ops;
    for (char c : infix) {
        if (isalnum(static_cast<unsigned char>(c))) {
            postfix += c;                   // operand langsung masuk ke keluaran
        } else if (c == '(') {
            ops.push(c);
        } else if (c == ')') {
            while (ops.top() != '(') {      // keluarkan sampai '(' pasangannya
                postfix += ops.top();
                ops.pop();
            }
            ops.pop();                      // buang '(' itu sendiri
        } else if (degree(c) > 0) {
            while (!ops.empty() && degree(ops.top()) >= degree(c)) {
                postfix += ops.top();       // >= : derajat yang sama keluar lebih dulu,
                ops.pop();                  //      sehingga 8-3-2 berarti (8-3)-2
            }
            ops.push(c);
        }                                   // selain itu (spasi) dilewati
    }
    while (!ops.empty()) {
        postfix += ops.top();
        ops.pop();
    }
    return postfix;
}
```

Kode 8.22 (Lanjutan) *p808_postfix.cpp: Soal 8.8, pengubahan shunting-yard dan evaluasi postfiks dengan std::stack.*

```
long long evaluate(const string &postfix) {
    stack<long long> values;
    for (char c : postfix) {
        if (isdigit(static_cast<unsigned char>(c))) {
            values.push(c - '0');
        } else {
            if (values.size() < 2) throw invalid_argument("missing operand");
            long long y = values.top();     // pop PERTAMA adalah operand KANAN
            values.pop();
            long long x = values.top();     // pop kedua adalah operand kiri
            values.pop();
            if (c == '+') values.push(x + y);
            else if (c == '-') values.push(x - y);
            else if (c == '*') values.push(x * y);
            else {
                if (y == 0) throw invalid_argument("division by zero");
                values.push(x / y);
            }
        }
    }
    if (values.size() != 1) throw invalid_argument("too many operands");
    return values.top();
}

int main() {
    const string tests[] = {"3+2*5", "a+b*c-d", "8-3-2", "(1+2)*(7-3)/2", "9-4/2*3"};
    for (const string &t : tests) {
        string p = toPostfix(t);
        cout << t << "  ->  " << p;
        bool numeric = true;
        for (char c : p) {
            if (isalpha(static_cast<unsigned char>(c))) numeric = false;
        }
        if (numeric) cout << "  =  " << evaluate(p);
        cout << "\n";
    }
    try {
        evaluate("3+");
    } catch (const invalid_argument &e) {
        cout << "evaluate(\"3+\"): " << e.what() << "\n";
    }
    return 0;
}
```

```
3+2*5  ->  325*+  =  13
a+b*c-d  ->  abc*+d-
8-3-2  ->  83-2-  =  3
(1+2)*(7-3)/2  ->  12+73-*2/  =  6
9-4/2*3  ->  942/3*-  =  3
evaluate("3+"): missing operand
```

Kode 8.23 dan **Kode 8.24** mencetak baris-baris yang sama. Dua rincian layak diperhatikan. Di Java, `ArrayDeque` adalah kelas stack yang dianjurkan; `push`, `pop`, dan `peek`-nya bekerja di bagian depan *deque* [7]. Di Python, pembagian bilangan bulat `//` membulatkan *ke bawah*, sedangkan C++ dan Java membulatkan *ke arah nol*; keduanya sepakat untuk bilangan positif tetapi tidak untuk $-7/2$, sehingga program Python menghitung hasil bagi dari nilai mutlaknya lalu memulihkan tandanya, dan ketiga program memberi jawaban yang sama untuk setiap masukan.

Kode 8.23 `P808Postfix.java`: Soal 8.8 dalam Java, dengan `ArrayDeque` sebagai stack.

```java
// P808Postfix.java - Soal 8.8: infiks ke postfiks dengan stack, lalu evaluasi postfiksnya.
import java.util.ArrayDeque;

public class P808Postfix {

    static int degree(char op) {               // "derajat" (presedensi) sebuah operator
        if (op == '*' || op == '/') return 2;
        if (op == '+' || op == '-') return 1;
        return 0;                              // '(' : lebih rendah daripada semua operator
    }

    // Operand berupa satu huruf atau satu angka; operatornya + - * / dan tanda kurung.
    static String toPostfix(String infix) {
        StringBuilder postfix = new StringBuilder();
        ArrayDeque<Character> ops = new ArrayDeque<>();   // dipakai sebagai stack: push, peek, pop
        for (char c : infix.toCharArray()) {
            if (Character.isLetterOrDigit(c)) {
                postfix.append(c);             // operand langsung masuk ke keluaran
            } else if (c == '(') {
                ops.push(c);
            } else if (c == ')') {
                while (ops.peek() != '(') {    // keluarkan sampai '(' pasangannya
                    postfix.append(ops.pop());
                }
                ops.pop();                     // buang '(' itu sendiri
            } else if (degree(c) > 0) {
                while (!ops.isEmpty() && degree(ops.peek()) >= degree(c)) {
                    postfix.append(ops.pop()); // >= : derajat yang sama keluar lebih dulu
                }
                ops.push(c);
            }
        }
        while (!ops.isEmpty()) {
            postfix.append(ops.pop());
        }
        return postfix.toString();
    }
```

Kode 8.23 (Lanjutan) *P808Postfix.java: Soal 8.8 dalam Java, dengan ArrayDeque sebagai stack.*

```
    static long evaluate(String postfix) {
        ArrayDeque<Long> values = new ArrayDeque<>();
        for (char c : postfix.toCharArray()) {
            if (Character.isDigit(c)) {
                values.push((long) (c - '0'));
            } else {
                if (values.size() < 2) throw new IllegalArgumentException("missing
operand");
                long y = values.pop();         // pop PERTAMA adalah operand KANAN
                long x = values.pop();         // pop kedua adalah operand kiri
                switch (c) {
                    case '+' -> values.push(x + y);
                    case '-' -> values.push(x - y);
                    case '*' -> values.push(x * y);
                    default -> {
                        if (y == 0) throw new IllegalArgumentException("division by zero");
                        values.push(x / y);
                    }
                }
            }
        }
        if (values.size() != 1) throw new IllegalArgumentException("too many operands");
        return values.pop();
    }

    public static void main(String[] args) {
        String[] tests = {"3+2*5", "a+b*c-d", "8-3-2", "(1+2)*(7-3)/2", "9-4/2*3"};
        for (String t : tests) {
            String p = toPostfix(t);
            boolean numeric = p.chars().noneMatch(Character::isLetter);
            System.out.println(t + "  ->  " + p + (numeric ? "  =  " + evaluate(p) : ""));
        }
        try {
            evaluate("3+");
        } catch (IllegalArgumentException e) {
            System.out.println("evaluate(\"3+\"): " + e.getMessage());
        }
    }
}
```

Kode 8.24 `p808_postfix.py`: Soal 8.8 dalam Python, dengan list sebagai stack.

```
# p808_postfix.py - Soal 8.8: infiks ke postfiks dengan stack, lalu evaluasi postfiksnya.


def degree(op):                         # "derajat" (presedensi) sebuah operator
    if op in "*/":
        return 2
    if op in "+-":
        return 1
    return 0                            # '(' : lebih rendah daripada semua operator


def to_postfix(infix):
    """Operand berupa satu huruf atau satu angka; operatornya + - * / dan tanda kurung."""
    postfix = []
    ops = []                            # list Python dipakai sebagai stack: append, [-1],
pop
    for c in infix:
        if c.isalnum():
            postfix.append(c)           # operand langsung masuk ke keluaran
        elif c == "(":
            ops.append(c)
        elif c == ")":
            while ops[-1] != "(":       # keluarkan sampai '(' pasangannya
                postfix.append(ops.pop())
            ops.pop()                   # buang '(' itu sendiri
        elif degree(c) > 0:
            while ops and degree(ops[-1]) >= degree(c):
                postfix.append(ops.pop())   # >= : derajat yang sama keluar lebih dulu
            ops.append(c)
    while ops:
        postfix.append(ops.pop())
    return "".join(postfix)


def evaluate(postfix):
    values = []
    for c in postfix:
        if c.isdigit():
            values.append(int(c))
        else:
            if len(values) < 2:
                raise ValueError("missing operand")
            y = values.pop()            # pop PERTAMA adalah operand KANAN
            x = values.pop()            # pop kedua adalah operand kiri
            if c == "+":
                values.append(x + y)
            elif c == "-":
```

Kode 8.24 (Lanjutan) *p808_postfix.py: Soal 8.8 dalam Python, dengan list sebagai stack.*

```
                values.append(x - y)
            elif c == "*":
                values.append(x * y)
            else:
                if y == 0:
                    raise ValueError("division by zero")
                q = abs(x) // abs(y)    # dibulatkan ke arah nol, seperti C++ dan Java
                values.append(q if (x >= 0) == (y >= 0) else -q)
    if len(values) != 1:
        raise ValueError("too many operands")
    return values[0]


def main():
    for t in ["3+2*5", "a+b*c-d", "8-3-2", "(1+2)*(7-3)/2", "9-4/2*3"]:
        p = to_postfix(t)
        numeric = not any(c.isalpha() for c in p)
        print(f"{t}  ->  {p}" + (f"  =  {evaluate(p)}" if numeric else ""))
    try:
        evaluate("3+")
    except ValueError as e:
        print(f'evaluate("3+"): {e}')


if __name__ == "__main__":
    main()
```

Biaya. Setiap karakter di-*push* dan di-*pop* paling banyak sekali pada setiap fungsi, sehingga keduanya berjalan dalam waktu $\Theta(L)$ untuk ekspresi berisi L karakter.

Penyelesaian keliru yang menggoda. Di dalam `evaluate`, wajar sekali menamai bilangan pertama yang di-*pop* `x` dan yang kedua `y`, lalu menghitung `x - y`. Program `wrong/p808_wrong.cpp` melakukan persis itu:

```
325*+  =  13
83-2-  =  7
12+73-*2/  =  0
942/3*-  =  -9
```

Penjumlahan dan perkalian tidak peduli urutan, sehingga `325*+` tetap menghasilkan 13, dan pengujian yang hanya memakai ekspresi itu akan lolos. Pengurangan dan pembagian peduli. Stack mengembalikan butir dalam urutan terbalik dari *push*-nya, sehingga operand yang di-*push terakhir*, yaitu operand kanan, keluar *pertama*.

> **UJI DIRI.** Ubahlah `(a+b)*(c-d)/e` ke postfiks dengan aturan di atas, lalu evaluasi `72-3*` dan `92/`.
>
> *Jawaban.* `ab+cd-*e/`. Untuk `72-3*`: $7 - 2 = 5$, lalu $5 \cdot 3 = 15$. Untuk `92/`: $9/2 = 4$ dalam pembagian bilangan bulat, dibulatkan ke arah nol.

8.9 Soal Baru yang Dikerjakan: Pointer dan Fungsi

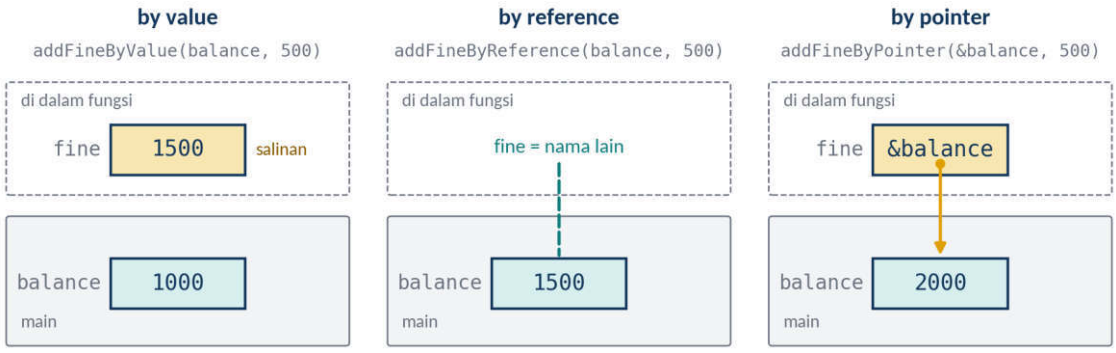
8.9.1 Soal 8.9 — By value, by reference, by pointer

> **SOAL.** Seorang anggota Pustaka Ceria menanggung tagihan keterlambatan 1000 rupiah, yang disimpan dalam `int balance`. Tulislah tiga fungsi yang masing-masing menambahkan 500 pada sebuah tagihan, satu menerima tagihan itu *by value*, satu *by reference*, dan satu *by pointer*; panggil masing-masing pada `balance`, dan cetak `balance` sesudah setiap pemanggilan. Di dalam setiap fungsi, cetaklah apakah parameternya adalah kotak memori yang sama

dengan `balance`. Kemudian tulislah dua fungsi `swap`, by value dan by pointer, serta fungsi `larger` yang mengembalikan alamat yang lebih besar dari dua `int`, dan pakailah untuk menambahkan 10 pada yang lebih besar. Terakhir, perlihatkan apa yang dilakukan Java dan Python sebagai ganti referensi dan pointer.

Dengan kata-kata kita sendiri. Bab 6 membandingkan ketiga cara itu dengan kata-kata; soal ini membuat sebuah program *memperlihatkannya*. Pertanyaannya sama pada setiap kasus: ketika fungsi menulis ke parameternya, kotak memori mana yang berubah?

Dengan tangan. **Gambar 8.8** menggambar memori selama setiap pemanggilan.



Gambar 8.8 Tiga cara mengirim `balance`, digambar pada akhir setiap pemanggilan. By value, fungsi bekerja pada salinan yang lenyap ketika fungsi selesai. By reference, `fine` adalah nama lain untuk `balance` itu sendiri. By pointer, `fine` menyimpan alamat `balance`, dan `*fine` mengikutinya.

- **By value** (`int fine`). Ketika fungsi dipanggil, sebuah kotak *baru* bernama `fine` dibuat dan nilai 1000 disalin ke dalamnya. `fine += amount` membuat kotak itu bernilai 1500, dan ketika fungsi selesai kotak itu dihancurkan. `balance` tidak pernah tersentuh: nilainya tetap 1000.
- **By reference** (`int &fine`). Tidak ada kotak baru. `fine` adalah nama kedua bagi kotak yang di `main` bernama `balance`; tanda *ampersand* di daftar parameter berarti persis itu. `fine += amount` mengubah `balance` menjadi 1500. Pemanggilannya tampak polos, `addFineByReference(balance, 500)`, dan itulah satu-satunya bahaya referensi: tidak ada apa pun di tempat pemanggilan yang memperingatkan bahwa `balance` mungkin berubah.
- **By pointer** (`int *fine`). Sebuah kotak baru dibuat, tetapi yang disalin ke dalamnya adalah *alamat* `balance`, yang harus diberikan pemanggil secara eksplisit dengan `&balance`. Fungsi mengikuti alamat itu dengan `*fine` dan mengubah apa yang ditemukannya di sana: `balance` menjadi 2000. Tanda `&` di tempat pemanggilan adalah peringatan yang tidak dimiliki referensi. Pointer juga dapat bernilai `nullptr`, sehingga fungsi yang cermat memeriksanya sebelum mengikutinya.

Penalaran yang sama menjelaskan `swap`. `swap` by value menukar dua salinan dan membiarkan `x` dan `y` seperti semula; `swap` by pointer menukar kotak-kotak yang dituju pointernya. Dan `larger(&x, &y)` mengembalikan sebuah *alamat*, sehingga `*larger(&x, &y) += 10` menambahkan 10 pada variabel mana pun yang lebih besar: pemanggilan itu adalah alamat, dan `*` di depannya membuka kotaknya.

Penyelesaian C++. Untuk mencetak "kotak yang sama dengan `balance`?", program membandingkan alamat: `&fine == callerBox` menanyakan apakah parameter itu tinggal di alamat yang sama dengan `balance`. (Kita mencetak `true` atau `false`, bukan alamatnya sendiri, karena alamat berubah dari satu jalannya ke jalannya yang lain dan akan membuat keluarannya mustahil diulang.) Seluruh programnya adalah **Kode 8.25**.

Kode 8.25 `p809_three_ways.cpp`: Soal 8.9, tiga cara mengirim argumen di C++, dengan pemeriksaan memori mana yang dipakai setiap parameter.

```cpp
// p809_three_ways.cpp - Soal 8.9: by value, by reference, dan by pointer, ditelusuri.
#include <iostream>
using namespace std;

int *callerBox = nullptr;   // mengingat letak balance milik main, untuk pemeriksaan di bawah

void addFineByValue(int fine, int amount) {       // fine adalah kotak BARU berisi salinan
    cout << "  inside by value    : same box as balance? " << (&fine == callerBox) << "\n";
    fine += amount;                                // hanya mengubah salinannya
}

void addFineByReference(int &fine, int amount) {  // fine adalah NAMA lain untuk balance
    cout << "  inside by reference: same box as balance? " << (&fine == callerBox) << "\n";
    fine += amount;
}

void addFineByPointer(int *fine, int amount) {    // fine menyimpan ALAMAT balance
    cout << "  inside by pointer  : same box as balance? " << (fine == callerBox) << "\n";
    if (fine != nullptr) {
        *fine += amount;                           // ikuti alamatnya, ubah isi di sana
    }
}

void swapByValue(int a, int b) {
    int t = a;
    a = b;
    b = t;
}

void swapByPointer(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

// Mengembalikan alamat tagihan yang lebih besar, agar pemanggil dapat mengubahnya.
int *larger(int *a, int *b) {
    return (*a >= *b) ? a : b;
}

int main() {
    cout << boolalpha;
    int balance = 1000;
    callerBox = &balance;
    cout << "start: balance = " << balance << "\n";
```

Kode 8.25 (Lanjutan) *p809_three_ways.cpp: Soal 8.9, tiga cara mengirim argumen di C++, dengan pemeriksaan memori mana yang dipakai setiap parameter.*

```
    addFineByValue(balance, 500);
    cout << "after by value    : balance = " << balance << "\n";
    addFineByReference(balance, 500);
    cout << "after by reference: balance = " << balance << "\n";
    addFineByPointer(&balance, 500);
    cout << "after by pointer  : balance = " << balance << "\n";

    int x = 1, y = 2;
    swapByValue(x, y);
    cout << "swapByValue(x, y)     : x = " << x << ", y = " << y << "\n";
    swapByPointer(&x, &y);
    cout << "swapByPointer(&x, &y) : x = " << x << ", y = " << y << "\n";

    *larger(&x, &y) += 10;                          // pemanggilan ini adalah alamat; *
membukanya
    cout << "*larger(&x, &y) += 10 : x = " << x << ", y = " << y << "\n";
    return 0;
}
```

```
start: balance = 1000
  inside by value    : same box as balance? false
after by value    : balance = 1000
  inside by reference: same box as balance? true
after by reference: balance = 1500
  inside by pointer  : same box as balance? true
after by pointer  : balance = 2000
swapByValue(x, y)     : x = 1, y = 2
swapByPointer(&x, &y) : x = 2, y = 1
*larger(&x, &y) += 10 : x = 12, y = 1
```

Java dan Python. Di sinilah ketiga bahasa berpisah jalan, dan inilah satu-satunya soal yang ketiga programnya mencetak baris berbeda. Java memiliki satu cara mengirim argumen: **by value**, selalu [6], [7]. Tetapi nilai sebuah variabel bertipe kelas *adalah referensi* ke sebuah objek, sehingga mengirim objek berarti menyalin referensinya, dan metode dapat mengikutinya lalu mengubah objek itu. Metode tidak dapat membuat variabel milik pemanggil merujuk ke objek lain. **Kode 8.26** memperlihatkan ketiga fakta itu: `addFine(int, ...)` tidak dapat mengubah `balance`; `addFine(Account, ...)` mengubah objeknya; `replace` mengarahkan ulang salinan referensinya sendiri dan pemanggil tidak pernah melihatnya. `swap(int, int)` di Java mustahil; jalan keluar yang lazim adalah menaruh kedua nilai di satu objek, di sini sebuah larik, lalu menukarnya di dalam objek itu.

Kode 8.26 `P809ThreeWays.java`: Soal 8.9 dalam Java, yang mengirim setiap argumen by value dan yang variabel objeknya menyimpan referensi.

```java
// P809ThreeWays.java - Soal 8.9: apa yang dilakukan Java sebagai ganti by reference dan by pointer.
public class P809ThreeWays {

    static class Account {             // saldo tagihan seorang anggota, disimpan di dalam sebuah objek
        int balance;

        Account(int balance) {
            this.balance = balance;
        }
    }

    static void addFine(int fine, int amount) {        // fine adalah salinan int-nya
        fine += amount;                                 // hanya mengubah salinannya
    }

    static void addFine(Account acc, int amount) {     // acc adalah salinan REFERENSI-nya
        acc.balance += amount;                          // mengikutinya: objek milik pemanggil berubah
    }

    static void replace(Account acc, int amount) {
        acc = new Account(amount);                      // hanya mengarahkan ulang salinannya;
    }                                                   // objek baru itu dibuang begitu saja

    static void swap(int a, int b) {                    // tidak mungkin berhasil di Java
        int t = a;
        a = b;
        b = t;
    }

    static void swap(int[] box, int i, int j) {         // berhasil: kedua int berada di satu objek
        int t = box[i];
        box[i] = box[j];
        box[j] = t;
    }

    static Account larger(Account a, Account b) {       // mengembalikan referensi ke salah satunya
        return (a.balance >= b.balance) ? a : b;
    }

    public static void main(String[] args) {
        int balance = 1000;
        System.out.println("start: balance = " + balance);
        addFine(balance, 500);
        System.out.println("after addFine(int, 500)    : balance = " + balance);
```

Kode 8.26 (Lanjutan) *P809ThreeWays.java: Soal 8.9 dalam Java, yang mengirim setiap argumen by value dan yang variabel objeknya menyimpan referensi.*

```
        Account acc = new Account(1000);
        addFine(acc, 500);
        System.out.println("after addFine(Account, 500): acc.balance = " + acc.balance);
        replace(acc, 9999);
        System.out.println("after replace(acc, 9999)   : acc.balance = " + acc.balance);

        int x = 1, y = 2;
        swap(x, y);
        System.out.println("swap(x, y)          : x = " + x + ", y = " + y);
        int[] box = {1, 2};
        swap(box, 0, 1);
        System.out.println("swap(box, 0, 1)     : box = [" + box[0] + ", " + box[1] + "]");

        Account p = new Account(2), q = new Account(1);
        larger(p, q).balance += 10;
        System.out.println("larger(p, q) += 10  : p = " + p.balance + ", q = " +
q.balance);
    }
}
```

```
start: balance = 1000
after addFine(int, 500)    : balance = 1000
after addFine(Account, 500): acc.balance = 1500
after replace(acc, 9999)   : acc.balance = 1500
swap(x, y)          : x = 1, y = 2
swap(box, 0, 1)     : box = [2, 1]
larger(p, q) += 10  : p = 12, q = 1
```

Python berperilaku seperti Java, dengan satu kekhasan: *segala sesuatu* adalah objek, termasuk bilangan bulat, dan parameter hanyalah nama baru bagi objek argumennya [8]. Jadi `add_fine_int` memang menerima objek `int` yang sama persis dengan `balance`; tetapi bilangan bulat tidak pernah dapat berubah, dan `fine += amount` membuat nama lokal `fine` merujuk ke objek bilangan bulat *baru*, 1500, dan membiarkan `balance`. Objek yang atributnya dapat berubah, seperti `Account`, berubah bagi semua yang merujuk kepadanya. Baris terakhir **Kode 8.27** memakai `is`, pengujian "objek yang sama?" milik Python.

Kode 8.27 `p809_three_ways.py`: Soal 8.9 dalam Python, yang parameternya adalah nama baru bagi objek yang sama.

```python
# p809_three_ways.py - Soal 8.9: apa yang dilakukan Python sebagai ganti by reference dan by pointer.


class Account:                      # saldo tagihan seorang anggota, disimpan di dalam sebuah objek
    def __init__(self, balance):
        self.balance = balance


def add_fine_int(fine, amount):     # fine adalah nama baru untuk objek int yang sama ...
    fine += amount                  # ... tetapi int tak dapat berubah: ini mengikat ulang namanya


def add_fine_account(acc, amount):  # acc adalah nama baru untuk objek milik pemanggil
    acc.balance += amount           # mengubah objek itu sendiri: pemanggil melihatnya


def replace(acc, amount):
    acc = Account(amount)           # hanya mengikat ulang nama lokal; objek baru itu dibuang


def swap(a, b):                     # tidak mungkin berhasil: hanya mengikat ulang nama-nama lokal
    a, b = b, a


def swap_in(box, i, j):             # berhasil: kedua nilai berada di satu objek list
    box[i], box[j] = box[j], box[i]


def larger(a, b):                   # mengembalikan salah satu dari kedua objek itu sendiri
    return a if a.balance >= b.balance else b


def main():
    balance = 1000
    print("start: balance =", balance)
    add_fine_int(balance, 500)
    print("after add_fine_int(balance, 500)    : balance =", balance)

    acc = Account(1000)
    add_fine_account(acc, 500)
    print("after add_fine_account(acc, 500)    : acc.balance =", acc.balance)
    replace(acc, 9999)
    print("after replace(acc, 9999)            : acc.balance =", acc.balance)
```

Kode 8.27 (Lanjutan) *p809_three_ways.py: Soal 8.9 dalam Python, yang parameternya adalah nama baru bagi objek yang sama.*

```
    x, y = 1, 2
    swap(x, y)
    print(f"swap(x, y)          : x = {x}, y = {y}")
    box = [1, 2]
    swap_in(box, 0, 1)
    print(f"swap_in(box, 0, 1)  : box = {box}")

    p, q = Account(2), Account(1)
    larger(p, q).balance += 10
    print(f"larger(p, q) += 10  : p = {p.balance}, q = {q.balance}")
    print("same object?", larger(p, q) is p)


if __name__ == "__main__":
    main()
```

```
start: balance = 1000
after add_fine_int(balance, 500)    : balance = 1000
after add_fine_account(acc, 500)    : acc.balance = 1500
after replace(acc, 9999)            : acc.balance = 1500
swap(x, y)          : x = 1, y = 2
swap_in(box, 0, 1)  : box = [2, 1]
larger(p, q) += 10  : p = 12, q = 1
same object? True
```

Tabel 8.7 menjajarkan ketiga bahasa.

Tabel 8.7 Apa yang dapat diubah sebuah fungsi di setiap bahasa. Inilah sebabnya versi Java dan Python pada Soal 8.3 sampai 8.5 mengembalikan simpul pertama yang baru, sedangkan versi C++ menerima `Node *&head`.

Pertanyaan	C++	Java	Python
Dapatkah fungsi mengubah variabel `int` milik pemanggil?	ya, by reference atau by pointer	tidak	tidak
Dapatkah ia mengubah objek yang dirujuk pemanggil?	ya	ya, melalui referensinya	ya, jika objeknya dapat berubah
Dapatkah ia membuat variabel pemanggil merujuk ke objek lain?	ya, dengan `T *&` atau `T **`	tidak	tidak
Cara menyerahkan simpul pertama yang baru	referensi ke pointer, atau mengembalikannya	mengembalikannya	mengembalikannya

Penyelesaian keliru yang menggoda. Andaikan sebuah fungsi harus membuat catatan tagihan baru dan memberi pemanggil pointer ke catatan itu:

```
int *makeFine(int amount) {
    int fine = amount;      // kotak lokal: lenyap ketika fungsi selesai
    return &fine;           // KELIRU: pemanggil menerima alamat kotak yang sudah mati
}
```

Variabel lokal `fine` hanya hidup selama `makeFine` berjalan; kotaknya dipakai ulang begitu fungsi selesai, sehingga pemanggil memegang *dangling pointer* (pointer menggantung). GCC menyadari kasus khusus ini dan memberi peringatan ketika program `wrong/p809_wrong.cpp` dikompilasi:

```
wrong/p809_wrong.cpp:7:12: warning: address of local variable 'fine' returned [-Wreturn-
local-addr]
```

dan, dengan optimasi dinyalakan, GCC 13 bahkan mengganti alamat mati itu dengan pointer nol, sehingga program mencetak `null`. Kompilator lain, atau fungsi yang sedikit lebih rumit yang menyembunyikan alamat lokal itu dari kompilator, tidak memberi peringatan, dan program

membaca sampah atau terhenti mendadak. Pilihan yang benar adalah mengembalikan nilainya sendiri (`int makeFine(int)`), atau membuat kotaknya di *heap* dengan `new` dan menjelaskan siapa yang wajib melakukan `delete`. Peringatan seperti inilah alasan setiap program dalam buku ini dikompilasi dengan `-Wall -Wextra`, dan alasan setiap peringatan diperlakukan sebagai kesalahan yang harus diperbaiki.

> **UJI DIRI.** Di C++, `int a = 5, b = 7; int *p = &a; int &r = b; *p = *p + r; r = 1; p = &b; *p += a;` Berapakah `a` dan `b` di akhir?
>
> *Jawaban.* `*p = *p + r` menyetel `a` menjadi $5 + 7 = 12$. `r = 1` menyetel `b` menjadi 1, karena `r` adalah nama lain untuk `b`. `p = &b` mengarahkan ulang `p` ke `b`. `*p += a` menambahkan 12 pada `b`. Jadi `a` = 12 dan `b` = 13.

8.10 Soal Baru yang Dikerjakan: Singly Linked List

8.10.1 Soal 8.10 — Membalik list di tempat

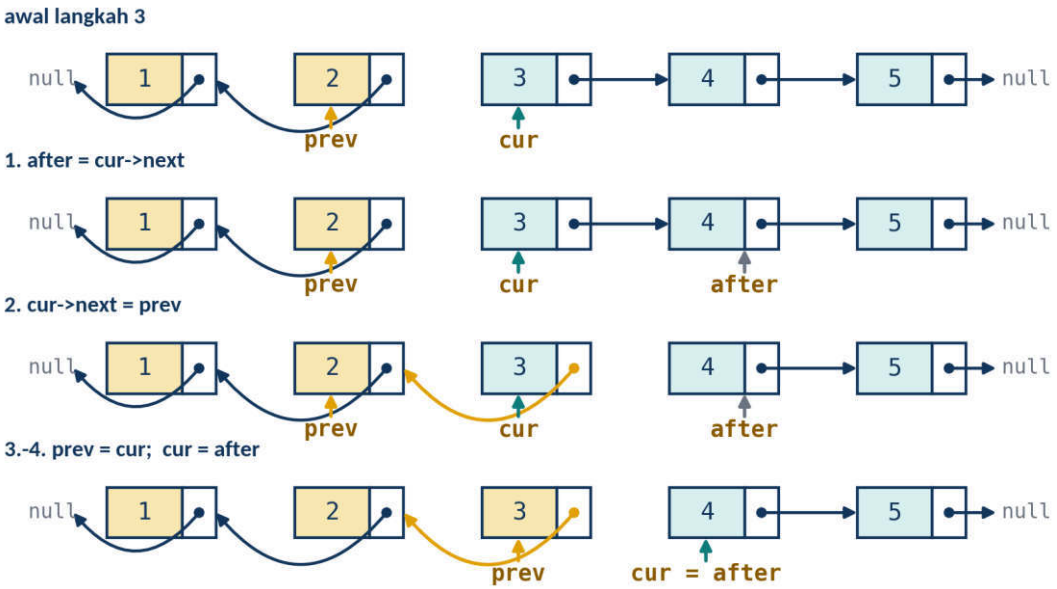
> **SOAL.** Tulislah fungsi `reverse` yang membalik singly linked list *di tempat*: ia tidak membuat simpul dan tidak menghapus simpul, ia hanya membalik arah pointer-pointer `next`, lalu mengembalikan simpul pertama yang baru. Telusurilah pada list 1, 2, 3, 4, 5, dengan mencetak bagian yang sudah dibalik dan bagian yang belum tersentuh sesudah setiap langkah, lalu ujilah pada list satu simpul dan list kosong.

Dengan kata-kata kita sendiri. Soal 8.4 membalik kalimat dengan membangun list *baru* melalui penyisipan depan. Di sini simpul-simpulnya sudah ada, dan kita harus memakainya ulang: sesudah pemanggilan, simpul yang berisi 5 menjadi yang pertama, `next`-nya menuju 4, dan seterusnya sampai 1, yang `next`-nya `nullptr`. Setiap panah harus berbalik arah.

Rancangan. Bayangkan list itu terpotong menjadi dua bagian: bagian *done*, yang sudah dibalik, dengan simpul pertama `prev`, dan bagian *rest*, yang belum tersentuh, dengan simpul pertama `cur`. Pada awalnya `done` kosong (`prev = nullptr`) dan `rest` adalah seluruh list (`cur = head`). Satu langkah memindahkan simpul pertama `rest` ke depan `done`, dalam empat penugasan:

1. `after = cur->next;` **simpan** jalan menuju sisa list;
2. `cur->next = prev;` balik panah `cur`, sehingga ia bergabung dengan bagian `done`;
3. `prev = cur;` `cur` kini simpul pertama `done`;
4. `cur = after;` sisa list dimulai dari simpul yang disimpan.

Invariannya: *`prev` adalah simpul pertama bagian depan yang sudah dibalik, dan `cur` adalah simpul pertama sisa yang belum tersentuh*. Ketika `cur` menjadi `nullptr`, sisanya kosong, sehingga `prev` adalah simpul pertama seluruh list yang sudah dibalik, dan fungsi mengembalikannya. **Gambar 8.9** memperlihatkan satu langkah penuh pada contohnya, dan **Kode 8.28** mencetak keadaan sesudah setiap langkah.



Gambar 8.9 Satu langkah pembalikan di tempat, yaitu langkah ketiga: simpan `cur->next` di `after`, balik panah `cur` ke `prev`, lalu majukan `prev` dan `cur` satu simpul. Simpul berwarna jingga adalah bagian yang sudah dibalik.

Kode 8.28 `p810_reverse_list.cpp`: Soal 8.10, pembalikan di tempat dengan tiga pointer, ditelusuri langkah demi langkah.

```cpp
// p810_reverse_list.cpp - Soal 8.10: membalik singly linked list di tempat dengan tiga pointer.
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node *next;
};

void printChain(const char *label, const Node *p) {
    cout << label << "[";
    for (; p != nullptr; p = p->next) {
        cout << p->data << (p->next != nullptr ? " " : "");
    }
    cout << "]";
}

// Membalik setiap panah. Mengembalikan simpul pertama yang baru (simpul terakhir yang lama).
Node *reverse(Node *head, bool trace) {
    Node *prev = nullptr;    // bagian yang sudah dibalik
    Node *cur = head;        // bagian yang belum disentuh
    int step = 0;
    while (cur != nullptr) {
        Node *after = cur->next;   // 1. SIMPAN jalan ke depan
        cur->next = prev;          // 2. balik panah milik cur
        prev = cur;                // 3. bagian yang dibalik bertambah satu
        cur = after;               // 4. sisanya berkurang satu
        if (trace) {
            cout << "after step " << ++step << ": ";
            printChain("done = ", prev);
            printChain("   rest = ", cur);
            cout << "\n";
        }
    }
    return prev;
}

Node *build(int from, int to) {            // list from, from+1, ..., to
    Node *head = nullptr;
    for (int v = to; v >= from; v--) {
        head = new Node{v, head};           // penyisipan depan, yang terbesar dulu
    }
    return head;
}
```

Kode 8.28 (Lanjutan) *p810_reverse_list.cpp: Soal 8.10, pembalikan di tempat dengan tiga pointer, ditelusuri langkah demi langkah.*

```
void destroy(Node *head) {
    while (head != nullptr) {
        Node *after = head->next;
        delete head;
        head = after;
    }
}

int main() {
    Node *list = build(1, 5);
    printChain("start       : done = ", nullptr);
    printChain("   rest = ", list);
    cout << "\n";
    list = reverse(list, true);
    printChain("result: ", list);
    cout << "\n";
    destroy(list);

    Node *one = reverse(build(7, 7), false);
    printChain("one node  -> ", one);
    cout << "\n";
    destroy(one);
    Node *none = reverse(nullptr, false);
    printChain("empty list -> ", none);
    cout << "\n";
    return 0;
}
```

```
start       : done = []   rest = [1 2 3 4 5]
after step 1: done = [1]   rest = [2 3 4 5]
after step 2: done = [2 1]   rest = [3 4 5]
after step 3: done = [3 2 1]   rest = [4 5]
after step 4: done = [4 3 2 1]   rest = [5]
after step 5: done = [5 4 3 2 1]   rest = []
result: [5 4 3 2 1]
one node  -> [7]
empty list -> []
```

Jejak itu memperlihatkan invarian yang sedang bekerja: sesudah langkah ke-k, `done` memuat k nilai pertama dalam urutan terbalik dan `rest` memuat sisanya dalam urutan aslinya. List satu simpul dan list kosong tidak memerlukan kode khusus: untuk satu simpul, perulangan berjalan sekali dan mengembalikan simpul itu; untuk list kosong, perulangan tidak pernah berjalan dan mengembalikan `nullptr`. **Kode 8.29** dan **Kode 8.30** mencetak baris-baris yang sama.

Kode 8.29 `P810ReverseList.java`: Soal 8.10 dalam Java.

```java
// P810ReverseList.java - Soal 8.10: membalik singly linked list di tempat dengan tiga referensi.
public class P810ReverseList {

    static class Node {
        int data;
        Node next;

        Node(int data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    static String chain(Node p) {
        StringBuilder b = new StringBuilder("[");
        for (; p != null; p = p.next) {
            b.append(p.data).append(p.next != null ? " " : "");
        }
        return b.append("]").toString();
    }

    // Membalik setiap panah. Mengembalikan simpul pertama yang baru (simpul terakhir yang lama).
    static Node reverse(Node head, boolean trace) {
        Node prev = null;          // bagian yang sudah dibalik
        Node cur = head;           // bagian yang belum disentuh
        int step = 0;
        while (cur != null) {
            Node after = cur.next; // 1. SIMPAN jalan ke depan
            cur.next = prev;       // 2. balik panah milik cur
            prev = cur;            // 3. bagian yang dibalik bertambah satu
            cur = after;           // 4. sisanya berkurang satu
            if (trace) {
                System.out.println("after step " + (++step) + ": done = " + chain(prev)
                        + "   rest = " + chain(cur));
            }
        }
        return prev;
    }
```

Kode 8.29 (Lanjutan) *P810ReverseList.java: Soal 8.10 dalam Java.*

```java
    static Node build(int from, int to) {      // list from, from+1, ..., to
        Node head = null;
        for (int v = to; v >= from; v--) {
            head = new Node(v, head);
        }
        return head;
    }

    public static void main(String[] args) {
        Node list = build(1, 5);
        System.out.println("start       : done = []   rest = " + chain(list));
        list = reverse(list, true);
        System.out.println("result: " + chain(list));
        System.out.println("one node  -> " + chain(reverse(build(7, 7), false)));
        System.out.println("empty list -> " + chain(reverse(null, false)));
    }
}
```

Kode 8.30 `p810_reverse_list.py`: Soal 8.10 dalam Python.

```python
# p810_reverse_list.py - Soal 8.10: membalik singly linked list di tempat dengan tiga nama.


class Node:
    __slots__ = ("data", "next")

    def __init__(self, data, next=None):
        self.data = data
        self.next = next


def chain(p):
    values = []
    while p is not None:
        values.append(str(p.data))
        p = p.next
    return "[" + " ".join(values) + "]"


def reverse(head, trace=False):
    """Membalik setiap panah. Mengembalikan simpul pertama yang baru (simpul terakhir yang
lama)."""
    prev = None                 # bagian yang sudah dibalik
    cur = head                  # bagian yang belum disentuh
    step = 0
    while cur is not None:
        after = cur.next        # 1. SIMPAN jalan ke depan
        cur.next = prev         # 2. balik panah milik cur
        prev = cur              # 3. bagian yang dibalik bertambah satu
        cur = after             # 4. sisanya berkurang satu
        if trace:
            step += 1
            print(f"after step {step}: done = {chain(prev)}   rest = {chain(cur)}")
    return prev


def build(first, last):
    head = None
    for v in range(last, first - 1, -1):
        head = Node(v, head)
    return head
```

Kode 8.30 (Lanjutan) *p810_reverse_list.py: Soal 8.10 dalam Python.*

```python
def main():
    lst = build(1, 5)
    print(f"start       : done = []   rest = {chain(lst)}")
    lst = reverse(lst, trace=True)
    print("result:", chain(lst))
    print("one node  ->", chain(reverse(build(7, 7))))
    print("empty list ->", chain(reverse(None)))


if __name__ == "__main__":
    main()
```

Biaya. Satu kali penelusuran, empat penugasan per simpul: waktu $\Theta(n)$ dan memori tambahan $O(1)$, karena hanya tiga variabel pointer yang dipakai, berapa pun panjang list-nya.

Penyelesaian keliru yang menggoda. Tinggalkan langkah 1 dan tulis ketiga langkah lainnya dalam urutan yang tampak jelas:

```
    cur->next = prev;      // KELIRU: satu-satunya jalan ke simpul 2 tertimpa di sini ...
    prev = cur;
    cur = cur->next;       // ... sehingga ini kini KEMBALI ke prev (nullptr pada awalnya)
```

Sesudah penugasan pertama, `next` milik simpul 1 bernilai `nullptr`, dan tidak ada apa pun lagi di program yang menunjuk ke simpul 2. Baris ketiga lalu mengikuti pointer yang baru saja ditulis dan memindahkan `cur` ke `nullptr`. Perulangan berakhir sesudah satu langkah, list "terbalik" hanyalah simpul 1 seorang diri, dan simpul 2 sampai 5 tak terjangkau. Di Java atau Python simpul-simpul itu akan diambil diam-diam; di C++ mereka bocor. Program `wrong/p810_wrong.cpp` mencetak hasilnya dan, bila dikompilasi dengan AddressSanitizer, pemeriksa kebocorannya melaporkan apa yang hilang ketika program berakhir:

```
1
==...==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 16 byte(s) in 1 object(s) allocated from:
Indirect leak of 48 byte(s) in 3 object(s) allocated from:
SUMMARY: AddressSanitizer: 64 byte(s) leaked in 4 allocation(s).
```

(Baris-baris yang menunjukkan *di mana* setiap blok dialokasikan dihilangkan.) Empat simpul masing-masing 16 bait hilang: simpul 2 secara langsung, karena tidak ada pointer di mana pun yang menuju kepadanya, dan simpul 3 sampai 5 secara tidak langsung, karena hanya memori yang hilang yang menunjuk ke mereka.

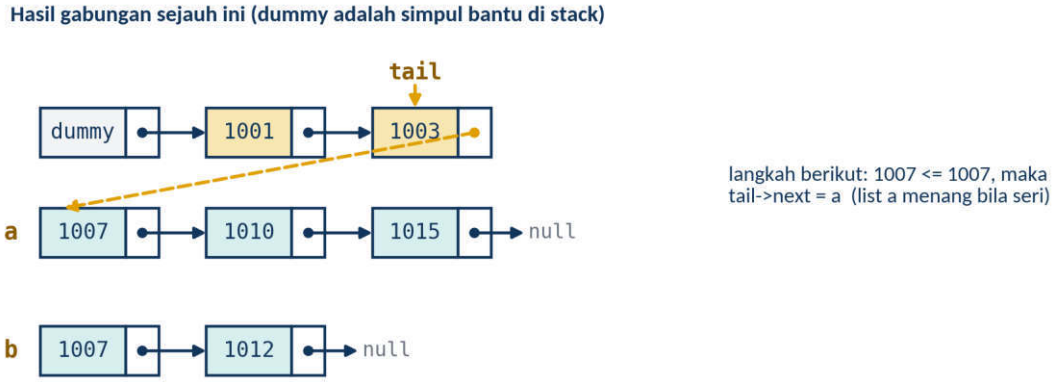
> **UJI DIRI.** Apa yang dikembalikan `reverse`, dan bagaimana pointer-pointer `next`-nya, untuk list dua simpul 8 → 9? Berapa kali masing-masing dari keempat penugasan dieksekusi?
>
> *Jawaban.* Fungsi mengembalikan simpul 9. Sesudahnya `next` milik 9 adalah 8 dan `next` milik 8 adalah `nullptr`. Perulangan berjalan dua kali, sehingga masing-masing dari keempat penugasan dieksekusi dua kali.

8.10.2 Soal 8.11 — Menggabungkan dua list terurut

> **SOAL.** Kedua cabang Pustaka Ceria masing-masing menyimpan linked list kode buku dalam urutan naik: cabang A memiliki 1003, 1007, 1010, 1015 dan cabang B memiliki 1001, 1007, 1012. Tulislah `merge(a, b)`, yang menggabungkan kedua list menjadi satu list naik *dengan menautkan ulang simpul-simpulnya*, tanpa membuat simpul baru. Hitunglah banyaknya perbandingan, dan ujilah kasus satu list kosong, dua list kosong, dan dua list yang sama sekali tidak bertumpang tindih.

Dengan kata-kata kita sendiri. Pada setiap saat, kode terkecil yang tersisa berada di depan list `a` atau di depan list `b`, karena kedua list terurut. Maka kita berulang kali membandingkan kedua simpul depan, melepas yang lebih kecil, dan menyambungkannya ke ujung hasil. Ketika satu list habis, simpul-simpul sisa list lainnya semuanya lebih besar daripada semua yang ada di hasil, dan sudah terurut, sehingga dapat disambungkan dengan satu penugasan. Inilah langkah *merge* pada merge sort (Bab 3), dan pada linked list ia tidak memerlukan larik tambahan [1], [11].

Rancangan. Menyambung ke "ujung hasil" memerlukan pointer `tail` ke simpul terakhir hasil. Pada awalnya hasil kosong dan tidak ada simpul terakhir, sehingga simpul pertama memerlukan kasus khusus. Trik bakunya adalah *simpul dummy*: simpul tambahan, bukan bagian dari jawaban, tempat hasil bermula. `tail` dimulai di dummy, setiap penyambungan adalah `tail->next = node;` `tail = node;`, dan hasil sebenarnya adalah `dummy.next`. Di C++, dummy dapat berupa variabel lokal biasa, `Node dummy{0, nullptr};`, yang tinggal di *stack*; `tail = &dummy` adalah pointer ke variabel itu, pemakaian `&` dari Bab 6. Dummy lenyap ketika `merge` selesai, dan pada saat itu tidak ada lagi yang menunjuk kepadanya. **Gambar 8.10** memperlihatkan keadaan sesudah dua simpul tersambung.



Gambar 8.10 Menggabungkan list kedua cabang: hasilnya bergantung pada simpul dummy, `tail` menandai simpul terakhirnya, dan perbandingan berikutnya adalah antara simpul depan `a` dan `b`.

Dengan tangan. Bandingkan 1003 dengan 1001: ambil 1001 dari `b`. Bandingkan 1003 dengan 1007: ambil 1003 dari `a`. Bandingkan 1007 dengan 1007: sama; kita mengambil yang dari `a` lebih dulu, yang membuat penggabungan *stabil* (kunci yang sama mempertahankan urutan list asalnya). Lalu 1010 dengan 1007: ambil 1007 dari `b`. 1010 dengan 1012: ambil 1010. 1015 dengan 1012: ambil 1012. Kini `b` kosong sesudah enam perbandingan; sambungkan sisa `a`, yaitu simpul tunggal 1015. Hasilnya 1001, 1003, 1007, 1007, 1010, 1012, 1015.

Setiap perbandingan menempatkan tepat satu simpul, dan simpul terakhir selalu ditempatkan tanpa perbandingan (ia bagian dari sisa), sehingga untuk list sepanjang m dan n

$$\begin{aligned} &\text{perbandingan} \le m + n - 1, && \text{waktu} \\ &\qquad = \Theta(m + n), && \text{memori tambahan} = O(1). \end{aligned} \tag{8.6}$$

Untuk kedua cabang, $m + n - 1 = 6$, dan keenam perbandingan di atas tepat mencapai batas itu. Ketika kedua list tidak bertumpang tindih, seperti `low + high` pada keluaran **Kode 8.31**, perbandingan berhenti begitu list yang lebih pendek habis: hanya 3.

Kode 8.31 `p811_merge_sorted.cpp`: Soal 8.11, menggabungkan dua linked list terurut dengan simpul dummy.

```cpp
// p811_merge_sorted.cpp - Soal 8.11: menggabungkan dua linked list terurut dengan menautkan ulang simpulnya.
#include <iostream>
using namespace std;

struct Node {
    int id;
    Node *next;
};

int comparisons = 0;

// Mengambil alih simpul-simpul a dan b (keduanya terurut naik) dan mengembalikan satu list terurut.
Node *merge(Node *a, Node *b) {
    Node dummy{0, nullptr};      // simpul bantu di stack, tidak pernah dicetak
    Node *tail = &dummy;         // tail = simpul terakhir list gabungan sejauh ini
    while (a != nullptr && b != nullptr) {
        comparisons++;
        if (a->id <= b->id) {    // <= : bila seri, list a didahulukan (stabil)
            tail->next = a;
            a = a->next;
        } else {
            tail->next = b;
            b = b->next;
        }
        tail = tail->next;
    }
    tail->next = (a != nullptr) ? a : b;   // sambungkan sisanya, sekaligus
    return dummy.next;
}

Node *build(const int *ids, int n) {
    Node *head = nullptr;
    for (int i = n - 1; i >= 0; i--) {
        head = new Node{ids[i], head};
    }
    return head;
}
```

Kode 8.31 (Lanjutan) *p811_merge_sorted.cpp: Soal 8.11, menggabungkan dua linked list terurut dengan simpul dummy.*

```
void printAndFree(const char *label, Node *head) {
    cout << label;
    while (head != nullptr) {
        cout << " " << head->id;
        Node *gone = head;
        head = head->next;
        delete gone;
    }
    cout << "  (" << comparisons << " comparisons)\n";
}

int main() {
    const int branchA[] = {1003, 1007, 1010, 1015};
    const int branchB[] = {1001, 1007, 1012};

    comparisons = 0;
    printAndFree("A + B       :", merge(build(branchA, 4), build(branchB, 3)));
    comparisons = 0;
    printAndFree("A + empty   :", merge(build(branchA, 4), nullptr));
    comparisons = 0;
    printAndFree("empty+empty :", merge(nullptr, nullptr));
    const int low[] = {1, 2, 3};
    const int high[] = {7, 8, 9};
    comparisons = 0;
    printAndFree("low + high  :", merge(build(low, 3), build(high, 3)));
    return 0;
}
```

```
A + B       : 1001 1003 1007 1007 1010 1012 1015  (6 comparisons)
A + empty   : 1003 1007 1010 1015  (0 comparisons)
empty+empty :  (0 comparisons)
low + high  : 1 2 3 7 8 9  (3 comparisons)
```

Hitungan-hitungan itu menegaskan Persamaan (8.6): 6 untuk kedua cabang, 0 bila salah satu list kosong, dan 3 bila setiap nilai satu list lebih kecil daripada setiap nilai list lainnya. **Kode 8.32** dan **Kode 8.33** mencetak baris-baris yang sama; di sana dummy adalah objek biasa.

Kode 8.32 `P811MergeSorted.java`: Soal 8.11 dalam Java.

```java
// P811MergeSorted.java - Soal 8.11: menggabungkan dua linked list terurut dengan menautkan ulang simpulnya.
public class P811MergeSorted {

    static class Node {
        int id;
        Node next;

        Node(int id, Node next) {
            this.id = id;
            this.next = next;
        }
    }

    static int comparisons = 0;

    // Mengambil alih simpul-simpul a dan b (keduanya terurut naik) dan mengembalikan satu list terurut.
    static Node merge(Node a, Node b) {
        Node dummy = new Node(0, null);    // simpul bantu, tidak pernah dicetak
        Node tail = dummy;                 // tail = simpul terakhir list gabungan sejauh ini
        while (a != null && b != null) {
            comparisons++;
            if (a.id <= b.id) {            // <= : bila seri, list a didahulukan (stabil)
                tail.next = a;
                a = a.next;
            } else {
                tail.next = b;
                b = b.next;
            }
            tail = tail.next;
        }
        tail.next = (a != null) ? a : b;   // sambungkan sisanya, sekaligus
        return dummy.next;
    }

    static Node build(int... ids) {
        Node head = null;
        for (int i = ids.length - 1; i >= 0; i--) {
            head = new Node(ids[i], head);
        }
        return head;
    }
```

Kode 8.32 (Lanjutan) *P811MergeSorted.java: Soal 8.11 dalam Java.*

```
    static void print(String label, Node head) {
        StringBuilder b = new StringBuilder(label);
        for (Node p = head; p != null; p = p.next) {
            b.append(" ").append(p.id);
        }
        System.out.println(b + "  (" + comparisons + " comparisons)");
    }

    public static void main(String[] args) {
        comparisons = 0;
        print("A + B       :", merge(build(1003, 1007, 1010, 1015), build(1001, 1007,
1012)));
        comparisons = 0;
        print("A + empty   :", merge(build(1003, 1007, 1010, 1015), null));
        comparisons = 0;
        print("empty+empty :", merge(null, null));
        comparisons = 0;
        print("low + high  :", merge(build(1, 2, 3), build(7, 8, 9)));
    }
}
```

Kode 8.33 `p811_merge_sorted.py`: Soal 8.11 dalam Python.

```python
# p811_merge_sorted.py - Soal 8.11: menggabungkan dua linked list terurut dengan menautkan ulang simpulnya.


class Node:
    __slots__ = ("id", "next")

    def __init__(self, id, next=None):
        self.id = id
        self.next = next


comparisons = 0


def merge(a, b):
    """Mengambil alih simpul-simpul a dan b (keduanya terurut naik) dan mengembalikan satu list terurut."""
    global comparisons
    dummy = Node(0)             # simpul bantu, tidak pernah dicetak
    tail = dummy                # tail = simpul terakhir list gabungan sejauh ini
    while a is not None and b is not None:
        comparisons += 1
        if a.id <= b.id:        # <= : bila seri, list a didahulukan (stabil)
            tail.next = a
            a = a.next
        else:
            tail.next = b
            b = b.next
        tail = tail.next
    tail.next = a if a is not None else b   # sambungkan sisanya, sekaligus
    return dummy.next


def build(*ids):
    head = None
    for i in reversed(ids):
        head = Node(i, head)
    return head


def show(label, head):
    parts = [label]
    while head is not None:
        parts.append(str(head.id))
        head = head.next
    print(" ".join(parts) + f"  ({comparisons} comparisons)")
```

Kode 8.33 (Lanjutan) *p811_merge_sorted.py: Soal 8.11 dalam Python.*

```python
def main():
    global comparisons
    comparisons = 0
    show("A + B       :", merge(build(1003, 1007, 1010, 1015), build(1001, 1007, 1012)))
    comparisons = 0
    show("A + empty   :", merge(build(1003, 1007, 1010, 1015), None))
    comparisons = 0
    show("empty+empty :", merge(None, None))
    comparisons = 0
    show("low + high  :", merge(build(1, 2, 3), build(7, 8, 9)))


if __name__ == "__main__":
    main()
```

Penyelesaian keliru yang menggoda. Ketika perulangan berakhir, rasanya pekerjaan sudah selesai, dan baris terakhir terlupakan atau diganti dengan `tail->next = nullptr;`. Program `wrong/p811_wrong.cpp` melakukan hal itu dan, agar dapat membebaskannya, menyisihkan simpul-simpul sisa lalu mencetaknya sebagai "lost" (hilang):

```
A + B       : 1001 1003 1007 1007 1010 1012  (6 comparisons)
lost        : 1015  (6 comparisons)
```

Perulangan berhenti ketika *salah satu* list kosong, sehingga list lainnya hampir selalu masih memiliki simpul, dan semuanya lebih besar daripada hasil. Kasus tepi yang membongkar *bug* ini hanyalah "list-listnya memiliki nilai terbesar yang berbeda", yang berlaku untuk hampir setiap masukan; tetapi pengujian yang memakai dua list yang berakhir dengan nilai sama, atau yang nilai terbesarnya kebetulan dibandingkan paling akhir, dapat melewatkannya.

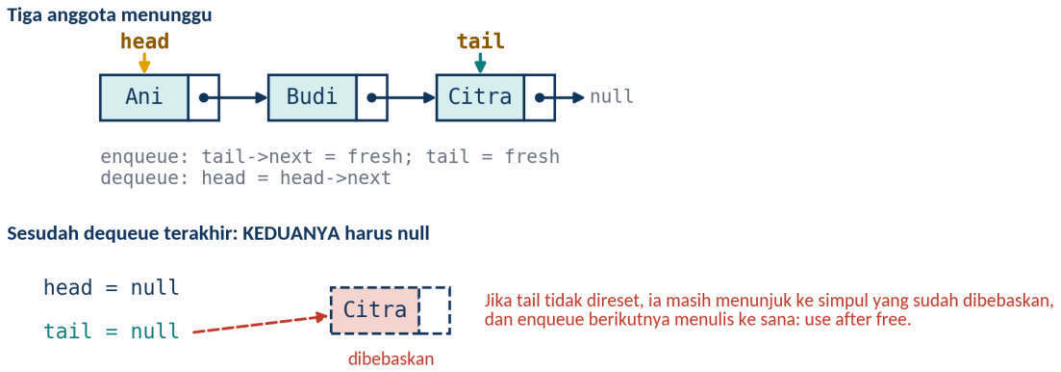
> **UJI DIRI.** Gabungkan `a` = 2, 4, 6, 8 dengan `b` = 5. Daftarlah perbandingannya, dan sebutkan list mana yang kosong ketika perulangan berhenti.
>
> *Jawaban.* 2 dengan 5 (ambil 2), 4 dengan 5 (ambil 4), 6 dengan 5 (ambil 5). Kini `b` kosong sesudah 3 perbandingan, dan sisa `a`, yaitu 6, 8, disambungkan dengan satu penugasan: 2, 4, 5, 6, 8. Batas pada Persamaan (8.6) adalah $4 + 1 - 1 = 4$.

8.10.3 Soal 8.12 — Queue dari simpul bertaut

> **SOAL.** Hanya satu eksemplar sebuah buku populer yang tersedia, dan anggota yang menginginkannya mengantre. Implementasikan queue berisi nama sebagai linked list dengan pointer `head` (depan, tempat anggota dilayani) dan pointer `tail` (belakang, tempat anggota bergabung), sehingga `enqueue` dan `dequeue` sama-sama berbiaya waktu $O(1)$. Ujilah dengan meng-*enqueue* Ani, Budi, dan Citra, melayani ketiganya, lalu meng-*enqueue* Dewi dan Eko dan melayani satu lagi; `dequeue` pada queue kosong harus dilaporkan, bukan membuat program terhenti.

Dengan kata-kata kita sendiri. Queue sirkular pada Soal 8.7 memiliki kapasitas tetap; queue ini tumbuh selama memori masih ada. Ia adalah list dengan head dan tail dari Bab 7, dipakai menurut aturan Bab 5: bergabung di belakang (`backInsertion`, yang dengan pointer tail tidak memerlukan penelusuran), keluar di depan (`frontDeletion`).

Rancangan. Invariannya terdiri atas tiga bagian: `head` menunjuk ke simpul anggota yang paling lama, atau bernilai `nullptr` jika queue kosong; `tail` menunjuk ke simpul anggota yang paling baru, atau bernilai `nullptr` jika queue kosong; dan `count` adalah banyaknya simpul. Kedua operasi harus menjaga ketiganya tetap benar, dan **Gambar 8.11** memperlihatkan saat yang berbahaya.



Gambar 8.11 Queue bertaut Soal 8.12. *Enqueue* bekerja di `tail` dan *dequeue* di `head`; ketika simpul terakhir pergi, `tail` juga harus disetel ulang menjadi `null`, atau ia tertinggal menunjuk ke memori yang sudah dibebaskan.

- **Enqueue.** Buat simpulnya. Jika queue kosong, simpul baru sekaligus menjadi yang pertama dan yang terakhir, sehingga `head` dan `tail` sama-sama menunjuk kepadanya. Jika tidak, tautkan simpul itu sesudah simpul terakhir saat ini, `tail->next = fresh`, dan jadikan ia yang terakhir, `tail = fresh`.
- **Dequeue.** Jika queue kosong, lempar eksepsi. Jika tidak, ingat simpul pertama, pindahkan `head` ke penerusnya, dan bebaskan simpul pertama yang lama. Kemudian baris yang mudah terlupa: jika `head` menjadi `nullptr`, queue kini kosong, dan **`tail` juga harus menjadi `nullptr`**, karena simpul yang ditunjuknya baru saja dibebaskan.

Kode 8.34 `p812_linked_queue.cpp`: Soal 8.12, queue yang dibangun dari linked list dengan `head` dan `tail`.

```cpp
// p812_linked_queue.cpp - Soal 8.12: queue yang dibangun dari linked list dengan head dan tail.
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;

struct Node {
    string name;
    Node *next;
};

class LinkedQueue {
    Node *head = nullptr;    // depan: anggota berikutnya yang dilayani
    Node *tail = nullptr;    // belakang: anggota yang datang terakhir
    int count = 0;

public:
    LinkedQueue() = default;
    LinkedQueue(const LinkedQueue &) = delete;             // satu pemilik untuk simpul-simpulnya
    LinkedQueue &operator=(const LinkedQueue &) = delete;
    ~LinkedQueue() {
        while (!isEmpty()) dequeue();
    }

    bool isEmpty() const { return head == nullptr; }
    int size() const { return count; }

    void enqueue(const string &name) {                     // O(1): tidak perlu berjalan ke belakang
        Node *fresh = new Node{name, nullptr};
        if (isEmpty()) {
            head = fresh;                                  // satu-satunya simpul menjadi sekaligus
            tail = fresh;                                  // depan dan belakang
        } else {
            tail->next = fresh;                            // belakang lama -> simpul baru
            tail = fresh;                                  // simpul baru menjadi belakang
        }
        count++;
    }
```

Kode 8.34 (Lanjutan) *p812_linked_queue.cpp: Soal 8.12, queue yang dibangun dari linked list dengan head dan tail.*

```
    string dequeue() {                                      // O(1)
        if (isEmpty()) throw underflow_error("dequeue from an empty queue");
        Node *gone = head;
        string name = gone->name;
        head = head->next;
        if (head == nullptr) {
            tail = nullptr;                                 // queue kosong kembali
        }
        delete gone;
        count--;
        return name;
    }

    void show() const {
        cout << "  queue:";
        for (Node *p = head; p != nullptr; p = p->next) cout << " " << p->name;
        cout << "  (size " << count << ", head " << (head ? head->name : "null")
             << ", tail " << (tail ? tail->name : "null") << ")\n";
    }
};

int main() {
    LinkedQueue waiting;            // anggota yang menunggu satu-satunya eksemplar sebuah
buku
    waiting.enqueue("Ani");
    waiting.enqueue("Budi");
    waiting.enqueue("Citra");
    waiting.show();
    cout << "served " << waiting.dequeue() << "\n";
    cout << "served " << waiting.dequeue() << "\n";
    cout << "served " << waiting.dequeue() << "\n";
    waiting.show();
    waiting.enqueue("Dewi");        // saat genting: enqueue sesudah queue sempat kosong
    waiting.enqueue("Eko");
    waiting.show();
    cout << "served " << waiting.dequeue() << "\n";
    try {
        LinkedQueue empty;
        empty.dequeue();
    } catch (const underflow_error &e) {
        cout << "caught: " << e.what() << "\n";
    }
    return 0;                       // destruktor membebaskan simpul milik Eko
}
```

```
  queue: Ani Budi Citra  (size 3, head Ani, tail Citra)
served Ani
served Budi
served Citra
  queue:  (size 0, head null, tail null)
  queue: Dewi Eko  (size 2, head Dewi, tail Eko)
served Dewi
caught: dequeue from an empty queue
```

Tampilan keadaan yang kedua, `(size 0, head null, tail null)`, adalah invarian sesudah queue kosong, dan *enqueue* Dewi yang menyusul hanya berhasil karenanya. Kelas ini juga memiliki destruktor yang meng-*dequeue* apa pun yang tersisa, sehingga simpul Eko dibebaskan ketika `waiting` keluar dari cakupannya, dan penyalinan dilarang dengan alasan yang sama seperti pada Soal 8.6. **Kode 8.35** dan **Kode 8.36** mencetak baris-baris yang sama.

Kode 8.35 `P812LinkedQueue.java`: Soal 8.12 dalam Java.

```java
// P812LinkedQueue.java - Soal 8.12: queue yang dibangun dari linked list dengan head dan tail.
public class P812LinkedQueue {

    static class Node {
        String name;
        Node next;

        Node(String name) {
            this.name = name;
        }
    }

    static class LinkedQueue {
        private Node head = null;      // depan: anggota berikutnya yang dilayani
        private Node tail = null;      // belakang: anggota yang datang terakhir
        private int count = 0;

        boolean isEmpty() { return head == null; }
        int size() { return count; }

        void enqueue(String name) {    // O(1): tidak perlu berjalan ke belakang
            Node fresh = new Node(name);
            if (isEmpty()) {
                head = fresh;          // satu-satunya simpul menjadi sekaligus
                tail = fresh;          // depan dan belakang
            } else {
                tail.next = fresh;     // belakang lama -> simpul baru
                tail = fresh;          // simpul baru menjadi belakang
            }
            count++;
        }

        String dequeue() {             // O(1)
            if (isEmpty()) throw new IllegalStateException("dequeue from an empty queue");
            String name = head.name;
            head = head.next;
            if (head == null) {
                tail = null;           // queue kosong kembali
            }
            count--;
            return name;
        }
```

Kode 8.35 (Lanjutan) *P812LinkedQueue.java: Soal 8.12 dalam Java.*

```
        void show() {
            StringBuilder b = new StringBuilder("  queue:");
            for (Node p = head; p != null; p = p.next) b.append(" ").append(p.name);
            b.append("  (size ").append(count)
             .append(", head ").append(head != null ? head.name : "null")
             .append(", tail ").append(tail != null ? tail.name : "null").append(")");
            System.out.println(b);
        }
    }

    public static void main(String[] args) {
        LinkedQueue waiting = new LinkedQueue();   // anggota yang menunggu satu-satunya
eksemplar sebuah buku
        waiting.enqueue("Ani");
        waiting.enqueue("Budi");
        waiting.enqueue("Citra");
        waiting.show();
        System.out.println("served " + waiting.dequeue());
        System.out.println("served " + waiting.dequeue());
        System.out.println("served " + waiting.dequeue());
        waiting.show();
        waiting.enqueue("Dewi");       // saat genting: enqueue sesudah queue sempat kosong
        waiting.enqueue("Eko");
        waiting.show();
        System.out.println("served " + waiting.dequeue());
        try {
            new LinkedQueue().dequeue();
        } catch (IllegalStateException e) {
            System.out.println("caught: " + e.getMessage());
        }
    }
}
```

Kode 8.36 `p812_linked_queue.py`: Soal 8.12 dalam Python.

```python
# p812_linked_queue.py - Soal 8.12: queue yang dibangun dari linked list dengan head dan tail.


class Node:
    __slots__ = ("name", "next")

    def __init__(self, name):
        self.name = name
        self.next = None


class LinkedQueue:
    def __init__(self):
        self._head = None       # depan: anggota berikutnya yang dilayani
        self._tail = None       # belakang: anggota yang datang terakhir
        self._count = 0

    def is_empty(self):
        return self._head is None

    def size(self):
        return self._count

    def enqueue(self, name):    # O(1): tidak perlu berjalan ke belakang
        fresh = Node(name)
        if self.is_empty():
            self._head = fresh  # satu-satunya simpul menjadi sekaligus
            self._tail = fresh  # depan dan belakang
        else:
            self._tail.next = fresh     # belakang lama -> simpul baru
            self._tail = fresh          # simpul baru menjadi belakang
        self._count += 1

    def dequeue(self):          # O(1)
        if self.is_empty():
            raise IndexError("dequeue from an empty queue")
        name = self._head.name
        self._head = self._head.next
        if self._head is None:
            self._tail = None   # queue kosong kembali
        self._count -= 1
        return name
```

Kode 8.36 (Lanjutan) *p812_linked_queue.py: Soal 8.12 dalam Python.*

```
    def show(self):
        names = []
        p = self._head
        while p is not None:
            names.append(p.name)
            p = p.next
        head = self._head.name if self._head else "null"
        tail = self._tail.name if self._tail else "null"
        print(f"  queue:{''.join(' ' + n for n in names)}  "
              f"(size {self._count}, head {head}, tail {tail})")


def main():
    waiting = LinkedQueue()     # anggota yang menunggu satu-satunya eksemplar sebuah buku
    waiting.enqueue("Ani")
    waiting.enqueue("Budi")
    waiting.enqueue("Citra")
    waiting.show()
    print("served", waiting.dequeue())
    print("served", waiting.dequeue())
    print("served", waiting.dequeue())
    waiting.show()
    waiting.enqueue("Dewi")     # saat genting: enqueue sesudah queue sempat kosong
    waiting.enqueue("Eko")
    waiting.show()
    print("served", waiting.dequeue())
    try:
        LinkedQueue().dequeue()
    except IndexError as e:
        print("caught:", e)


if __name__ == "__main__":
    main()
```

Biaya. `enqueue` dan `dequeue` masing-masing mengerjakan sejumlah tetap penugasan pointer: waktu $O(1)$, tanpa penelusuran list. Memorinya satu simpul per anggota yang menunggu.

Penyelesaian keliru yang menggoda. Tinggalkan ketiga baris yang menyetel ulang `tail`, dan uji kekosongan queue dengan melihat `tail` (seperti dilakukan sebagian buku teks). Queue bekerja sempurna sampai ia menjadi kosong. Pada saat itu `tail` masih menyimpan alamat simpul Citra yang sudah dibebaskan, *enqueue* berikutnya melihat `tail` yang tidak nol, dan mengeksekusi `tail->next = fresh`: operasi tulis ke memori yang sudah dibebaskan. Tanpa *sanitizer*, program itu bahkan mungkin tampak berjalan. Program `wrong/p812_wrong.cpp`, yang dikompilasi dengan AddressSanitizer, dihentikan tepat di baris itu; baris-baris awal laporannya (alamatnya dipersingkat) adalah

```
served Ani
served Budi
served Citra
now enqueue Dewi
==...==ERROR: AddressSanitizer: heap-use-after-free on address 0x...
WRITE of size 8 at 0x... thread T0
    #0 0x... in LinkedQueue::enqueue(...) wrong/p812_wrong.cpp:34
0x... is located 32 bytes inside of 40-byte region [0x...,0x...)
freed by thread T0 here:
```

Bacalah seperti sebuah kalimat: operasi tulis 8 bait (sebuah pointer) ke memori yang sudah dibebaskan, dilakukan oleh `enqueue` di baris 34, mendarat 32 bait di dalam blok berukuran 40 bait. Blok itu adalah sebuah `Node` (`std::string` 32 bait disusul pointer `next` 8 bait), dan posisi 32

tepat merupakan medan `next`-nya: `tail->next` yang basi. Laporan itu bahkan menyebut baris yang membebaskan blok tersebut. Inilah *bug* paling umum pada queue bertaut, dan ia ditemukan dalam hitungan detik oleh Langkah 5 metode kita ("strukturnya menjadi kosong") yang digabung dengan Langkah 7 ("jalankan di bawah pemeriksa memori").

UJI DIRI. Mulai dari queue kosong: *enqueue* X; *dequeue*; *enqueue* Y; *enqueue* Z; *dequeue*. Berikan `head`, `tail`, dan `count` sesudah setiap operasi.

Jawaban. Sesudah *enqueue* X: keduanya menunjuk ke X, `count` 1. Sesudah *dequeue*: keduanya `nullptr`, `count` 0 (baris penyetel ulang berjalan). Sesudah *enqueue* Y: keduanya menunjuk ke Y, `count` 1. Sesudah *enqueue* Z: `head` Y, `tail` Z, `count` 2. Sesudah *dequeue* (mengembalikan Y): `head` dan `tail` sama-sama Z, `count` 1.

8.11 Perbandingan Soal-Soal dalam Bab Ini

Tabel 8.8 menjajarkan kedua belas soal. Bacalah kolom terakhirnya dari atas ke bawah. Setiap penyelesaian keliru yang menggoda dalam bab ini gagal pada satu jenis masukan tertentu, dan pada kebanyakan soal masukan itu *bukan* contoh yang diberikan dalam pernyataan soal: contoh Soal 8.3 tidak memuat bilangan negatif, contoh Soal 8.5 tidak memuat salinan yang bersebelahan, pengujian `325*+` pada Soal 8.8 tidak memuat pengurangan. Itulah argumen terkuat yang dapat diajukan bab ini untuk Langkah 5 metode kita.

Tabel 8.8 Kedua belas soal dalam bab ini: struktur yang dipakai masing-masing, biayanya, dan masukan yang membuat penyelesaian kelirunya gagal. n adalah banyaknya simpul, L panjang dalam karakter.

Soal	Struktur	Waktu	Memori tambahan	Masukan yang membuat penyelesaian keliru gagal
8.1	stack larik	$O(1)$ per operasi	$\Theta(\text{MAX})$	setiap `pop` (membaca butir di bawah puncak)
8.2	list dua arah	$\Theta(n^2)$ untuk membangun, satu langkah per gerakan	$O(1)$	setiap petunjuk yang simpul berikutnya memiliki langkah berbeda
8.3	dua list terurut	$\Theta(n^2)$ kasus terburuk	satu simpul per nilai	bilangan ganjil negatif
8.4	list sebagai stack	$\Theta(L)$	satu simpul per kata	kalimat sepanjang 100 karakter atau lebih
8.5	list, dua pointer	$\Theta(n^2)$	$O(1)$	dua salinan sebuah nilai yang bersebelahan
8.6	stack larik	$\Theta(L)$	$O(L)$	pasangan bersilangan seperti `([)]`
8.7	larik sirkular	$O(1)$ per operasi	$\Theta(\text{CAPACITY})$	*enqueue* sesudah beberapa *dequeue*
8.8	stack pustaka	$\Theta(L)$	$O(L)$	pengurangan atau pembagian
8.9	parameter fungsi	$O(1)$	$O(1)$	setiap pemanggilan yang seharusnya mengubah variabel pemanggil
8.10	list, tiga pointer	$\Theta(n)$	$O(1)$	setiap list berisi dua simpul atau lebih
8.11	dua list terurut	$\Theta(m+n)$	$O(1)$	list-list dengan nilai terbesar yang berbeda
8.12	list dengan head dan tail	$O(1)$ per operasi	satu simpul per butir	*enqueue* sesudah queue sempat kosong

Biaya-biaya itu mengulang tiga pola, dan mengenalinya sekilas pandang sangat berguna. **Sejumlah tetap perubahan pointer atau indeks** berbiaya $O(1)$: setiap operasi stack atau queue,

baik yang dibangun di atas larik maupun di atas simpul. **Satu kali penelusuran struktur** berbiaya $\Theta(n)$: pembalikan, penggabungan, pencetakan, pencocokan tanda kurung, pengubahan ke postfiks. **Penelusuran di dalam penelusuran**, yang untuk setiap simpul melihat semua simpul sesudahnya, berbiaya $\Theta(n^2)$ menurut jumlah $\frac{n(n-1)}{2}$ pada Persamaan (8.3) dan Persamaan (8.4): penyisipan terurut n nilai, pembuangan duplikat, dan menambah n kali ke list tanpa tail. Setiap kali program dari golongan ketiga terlalu lambat, pertanyaan yang tepat adalah struktur mana yang akan mengubah penelusuran dalamnya menjadi satu langkah: pointer tail (Bab 9), *hash set* (Bab 15), atau *heap* (Bab 14).

8.12 Profitina dalam Praktik

Profitina adalah platform visibilitas AI (GEO) yang beroperasi di app.profitina.com, yang mengukur seberapa siap situs web sebuah merek untuk dikutip oleh mesin jawab berbasis AI dan membantu memperbaiki apa yang ditemukannya. Struktur-struktur dalam bab ini kecil, tetapi *bentuknya* tampak di mana-mana dalam produk semacam itu. Kita menggambarkan bentuk masalahnya dan golongan penyelesaiannya, bukan cara Profitina sendiri dibangun.

Pekerjaan yang menunggu gilirannya adalah queue. Profitina memindai ulang situs setiap merek secara otomatis dengan irama yang bergantung pada tingkat langganan, dan memunculkan peringatan ketika sebuah skor berubah. Sistem apa pun yang melakukan hal itu, di suatu tempat, memiliki pekerjaan yang menunggu untuk dikerjakan menurut urutan jatuh temponya: pertama masuk, pertama keluar, seperti pada Soal 8.7 dan 8.12. Umpan komunitas *real-time*, yang dikirim ke setiap halaman terbuka melalui *Server-Sent Events*, memiliki bentuk yang sama dalam skala lebih kecil: setiap penonton harus menerima unggahan baru dalam urutan terbitnya. Sebuah *server* streaming juga harus memutuskan apa yang terjadi ketika koneksi seorang penonton lambat dan unggahan untuknya menumpuk. Queue berkapasitas tetap, yaitu queue sirkular pada Soal 8.7, adalah cara klasik untuk membatasi memori yang dapat dihabiskan seorang pembaca lambat, dan kasus “penuh”-nya memaksa perancang memutuskan secara eksplisit apa yang dibuang atau kapan koneksi diputus.

Mengikuti rujukan adalah menelusuri list. Dalam diskusi berutas, setiap balasan merujuk ke unggahan yang dijawabnya, dan *quote-repost* merujuk ke unggahan yang dikutipnya. Berjalan dari balasan yang dalam kembali ke awal utasnya berarti mengikuti rujukan-rujukan itu satu per satu, persis seperti mengikuti `prev` pada Soal 8.2, dan hal itu memunculkan pertanyaan yang sama: bagaimana jika unggahan yang dirujuk sudah dihapus (sebuah `nullptr` di tengah penelusuran), dan bagaimana jika rantai rujukan berputar kembali ke dirinya sendiri (lingkaran pada Persamaan (8.2))? Daftar notifikasi di dasbor, yang terbaru di atas, memiliki bentuk list pada Soal 8.4: setiap butir baru ditaruh di depan.

Simpan apa yang Anda perlukan sebelum mengubah apa pun. Penyusun unggahan Profitina mengosongkan kotak teks begitu unggahan dikirim, sehingga pengunggahan terasa seketika, dan memulihkan teksnya jika pengiriman gagal. Perilaku itu hanya mungkin jika program menyimpan salinan teks *sebelum* mengosongkan kotaknya: aturan Subbab 8.1, dalam antarmuka pengguna, bukan dalam linked list.

Server-nya ditulis dalam Python, sehingga tidak ada insinyur Profitina yang membebaskan simpul dengan tangan, dan *bug* pada Soal 8.10 dan 8.12 tidak mungkin terjadi dalam bentuk seperti di sini. Sepupu-sepupunya mungkin terjadi: queue yang rujukan “butir terakhir”-nya menjadi basi, *cache* yang membuat objek tetap hidup sesudah seharusnya lenyap. Pertahanannya sama dengan pertahanan dalam bab ini, yaitu pengujian yang membidik kasus-kasus tepi, dan Profitina tidak

merilis apa pun sebelum seluruh rangkaian uji otomatisnya, jauh lebih dari seribu uji, lolos. Versi latihan yang dapat dijalankan untuk gagasan-gagasan ini adalah Profitina Laundry, studi kasus seri buku ini; Soal Latihan 8.13 mengubah queue pada Soal 8.12 menjadi antrean mesin cucinya.

8.13 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Menimpa satu-satunya pointer ke sesuatu yang masih Anda perlukan. `cur->next = prev` sebelum menyimpan `cur->next` (Soal 8.10), `cur->next = fresh` sebelum `fresh->next = cur->next` (Soal 8.3). Beri nomor pada penugasan-penugasan di gambar Anda dan periksalah masing-masing dengan pertanyaan "apa yang menjadi tak terjangkau?"

Membaca dari simpul sesudah meninggalkannya. Syarat perulangan seperti `k < cur->step` yang membaca ulang simpul selagi `cur` bergerak (Soal 8.2). Salinlah petunjuk sebuah simpul ke variabel lokal sebelum bergerak.

Membebaskan sebelum melepas tautan. `delete dup; runner->next = dup->next;` membaca memori yang sudah dibebaskan. Lepaskan tautannya dulu, baru bebaskan (Soal 8.5).

Meninggalkan pointer ke memori yang sudah dibebaskan. `tail` yang masih menunjuk ke simpul yang dibebaskan oleh `dequeue` terakhir (Soal 8.12), atau fungsi yang mengembalikan alamat variabel lokalnya sendiri (Soal 8.9). Setiap kali sebuah simpul dibebaskan, tanyakan pointer lain mana yang masih menuju kepadanya.

Mencampur dua konvensi untuk satu indeks. `top` sebagai "indeks butir teratas" di satu fungsi dan "sel kosong berikutnya" di fungsi lain (Soal 8.1); `rear` sebagai posisi padahal pertanyaannya adalah hitungan (Soal 8.7). Tuliskan invariannya sebagai komentar di samping medan itu.

Memajukan pointer pemindai sesudah penghapusan. Sesudah penghapusan, penerusnya adalah simpul baru yang belum diperiksa (Soal 8.5).

Melupakan sisa. Perulangan *merge* berakhir ketika *satu* list kosong; list lainnya masih harus disambungkan (Soal 8.11).

Mengira `x % 2 == 1` berarti ganjil. Di C++ dan Java, sisa bagi bilangan negatif bernilai negatif. Ujilah `x % 2 == 0` (Soal 8.3).

Mem-*pop* operand dalam urutan yang salah. Nilai pertama yang di-*pop* adalah operand kanan (Soal 8.8).

Menghitung alih-alih mengingat. Penghitung tidak dapat membedakan `(` dari `[`; stack dapat (Soal 8.6).

Penyangga berukuran tetap untuk masukan yang panjangnya tidak diketahui. `char str[100]` meluap pada baris panjang pertama; pakailah `std::string` (Soal 8.4).

Lupa bahwa penanda akhir bukan data. Perulangan pembacaan `do`–`while` ikut menyimpan *sentinel*-nya (Soal 8.2 dan 8.5). Putuskan secara eksplisit apakah penanda itu termasuk dalam struktur.

Hanya menguji contoh. Pada Soal 8.3, 8.5, 8.8, dan 8.11, contoh masukan meloloskan penyelesaian yang keliru. Ujilah struktur kosong, satu simpul, posisi pertama dan terakhir, tetangga yang sama, dan nilai negatif, lalu jalankan uji C++ sekali di bawah AddressSanitizer atau Valgrind.

8.14 Ringkasan Bab

- Soal tentang stack, queue, dan linked list diselesaikan dengan tujuh langkah (Subbab 8.2): menyatakan ulang soal disertai contoh, memilih struktur, menggambar kotak dan pointer, menuliskan invarian, mendaftar kasus tepi, mengurutkan perubahan pointer secara aman, dan menguji, termasuk di bawah pemeriksa memori.
- Satu aturan menjelaskan sebagian besar *bug* dalam bab ini: sebelum mengubah sebuah tautan, pertahankan jalan menuju semua yang masih akan Anda perlukan.
- Stack: stack larik adalah satu larik dan satu indeks dengan invarian tertulis, Persamaan (8.1) (Soal 8.1); tanda kurung cocok dengan stack karena yang terakhir dibuka harus yang pertama ditutup (Soal 8.6); algoritme *shunting-yard* mengubah infiks menjadi postfiks, dan evaluasi mem-*pop* operand kanan lebih dulu (Soal 8.8).
- Queue: larik sirkular memutar indeksnya dengan operator sisa bagi, Persamaan (8.5), dan menyimpan hitungan agar penuh dan kosong berbeda (Soal 8.7); queue bertaut dengan head dan tail mengerjakan kedua operasi dalam $O(1)$ dan wajib menyetel ulang `tail` ketika kosong (Soal 8.12).
- Pointer dan fungsi: by value menyalin, by reference memberi nama lain, by pointer mengirim alamat; Java dan Python mengirim referensi ke objek secara by value, sehingga keduanya mengembalikan simpul pertama yang baru di tempat C++ menerima `Node *&` (Soal 8.9, serta Soal 8.3 sampai 8.5).
- Singly linked list: list dua arah memerlukan kedua panah disetel pada setiap penyisipan, dan penelusuran deterministik yang mengunjungi lebih dari n simpul tidak pernah berakhir, Persamaan (8.2) (Soal 8.2); penyisipan terurut berdiri di pendahulu dan berbiaya Persamaan (8.3) (Soal 8.3); penyisipan depan membalik urutan (Soal 8.4); pembuangan duplikat melepas tautan, lalu membebaskan, lalu tidak bergerak, dengan biaya Persamaan (8.4) (Soal 8.5); pembalikan di tempat memerlukan tiga pointer (Soal 8.10); penggabungan dengan simpul dummy berbiaya paling banyak $m + n - 1$ perbandingan, Persamaan (8.6) (Soal 8.11).
- Doubly linked list, *deque*, dan list sirkular, yang di sini hanya disentuh sejauh diperlukan Soal 8.2, adalah pokok bahasan Bab 9.

8.15 Soal Latihan

Soal Latihan 8.1 sampai 8.5 dan 8.10 meminta kode; penyelesaiannya adalah program-program pada Subbab 8.16. Yang lain meminta penalaran dan dijawab dengan kata-kata.

1. **(Stack.)** Tambahkan `isFull()` dan `clear()` pada stack Soal 8.1. Apa yang dikembalikan `isFull()` sesudah 1000 kali *push*, dan apa yang dilakukan `clear()` terhadap nilai-nilai yang tersimpan di larik?
2. **(Panjang dan jumlah.)** Tulislah satu fungsi yang menelusuri singly linked list sekali dan menghitung panjangnya sekaligus jumlah nilainya. Di C++, kembalikan kedua hasil melalui parameter referensi. Berapa hasilnya untuk 4, 8, 15, 16, 23, 42?
3. **(Simpul tengah.)** Temukan simpul tengah sebuah list dalam satu kali penelusuran, dengan dua pointer: `slow` bergerak satu simpul per langkah dan `fast` dua. Untuk panjang genap, kembalikan yang kedua dari dua simpul tengah. Manakah simpul tengah 4, 8, 15, 16, 23, 42, dan 4, 8, 15, 16, 23?
4. **(Menghapus simpul ke-k.)** Tulislah fungsi yang menghapus simpul nomor k (simpul pertama bernomor 1) dan melaporkan apakah simpul k ada. Hapuslah simpul 3, lalu simpul 1, lalu simpul 9 dari 4, 8, 15, 16, 23, 42.

5. **(Palindrom.)** Tentukan apakah sebuah frasa terbaca sama dari belakang, dengan mengabaikan spasi dan huruf besar/kecil, dengan mem-*push* hurufnya ke stack *dan* ke queue lalu membandingkan apa yang keluar. Cobalah "Kasur ini rusak" dan "stack".
6. **(Penelusuran dengan tangan.)** Apa yang dicetak program Soal 8.2 untuk masukan `f 1 o`, `f 2 k`, `f 2 y`, `b 1 a`, `0 0 0`? Gambarlah list dan gerakan-gerakannya.
7. **(Queue sirkular dengan tangan.)** Dengan `CAPACITY` = 3, lakukan: *enqueue* 1, 2, 3; *dequeue*; *enqueue* 4; *dequeue*; *dequeue*; *enqueue* 5, 6. Berikan slot, `front`, dan `count` sesudah setiap operasi, serta urutan pelayanan butir-butir yang tersisa.
8. **(Postfiks dengan tangan.)** Ubahlah `a*(b+c*d)-e` ke postfiks dengan aturan Soal 8.8, perlihatkan stack sesudah setiap karakter, lalu evaluasi `234*+5-`.
9. **(Aritmetika pointer.)** Tebaklah keluaran potongan kode berikut, lalu sebutkan isi larik di akhir: `int v[] = {10, 20, 30, 40}; int *p = v; *(p + 2) += 5; p++; *p = *p * 2; int *q = &v[3]; cout << *p + *q << " " << q - p;`
10. **(Duplikat pada list terurut.)** Jika list terurut, semua salinan sebuah nilai bersebelahan. Buanglah duplikatnya dalam satu kali penelusuran, dengan waktu $O(n)$ dan memori tambahan $O(1)$. Ujilah pada 1, 1, 2, 3, 3, 3, 7, 7.
11. **(Queue dari dua stack.)** Jelaskan cara membangun queue dari dua stack, `in` dan `out`, sehingga `enqueue` mem-*push* ke `in` dan `dequeue` mem-*pop* dari `out`. Kapan butir harus dipindahkan dari `in` ke `out`? Tunjukkan bahwa, pada urutan operasi apa pun, setiap butir di-*push* dan di-*pop* paling banyak dua kali, sehingga biaya rata-rata per operasi adalah $O(1)$.
12. **(Tanpa penanda tersimpan.)** Tulis ulang perulangan pembacaan Soal 8.5 sehingga penanda akhir -1 tidak pernah disimpan. Bagaimana kedua baris cetakan untuk contoh masukan, dan fungsi mana yang menjadi tidak diperlukan?
13. **(Profitina Laundry.)** Profitina Laundry menyimpan pesanan yang menunggu mesin cucinya dalam queue Soal 8.12. Setiap pesanan memiliki nomor pesanan dan berat dalam kilogram. Sebutkan bagian mana dari **Kode 8.34** yang berubah, tambahkan operasi `peek` yang mengembalikan pesanan berikutnya tanpa mengambilnya, dan sebutkan apa yang harus dilakukan `peek` pada queue kosong.

8.16 Penyelesaian Soal Latihan

Soal Latihan 1 sampai 5 dan 10. **Kode 8.37**, **Kode 8.38**, dan **Kode 8.39** memuat semua fungsi yang diminta dan mencetak baris-baris yang sama.

Kode 8.37 `ex08_solutions.cpp`: soal latihan pemrograman Bab 8 dalam C++.

```cpp
// ex08_solutions.cpp - penyelesaian soal latihan pemrograman Bab 8.
#include <cctype>
#include <iostream>
#include <queue>
#include <stack>
#include <string>
using namespace std;

// ---------- Soal Latihan 8.1: isFull() dan clear() untuk stack larik ----------
const int MAX = 1000;

class Stack {
    int top = -1;
    int a[MAX];

public:
    bool isEmpty() const { return top < 0; }
    bool isFull() const { return top == MAX - 1; }
    void clear() { top = -1; }            // nilai lama tetap di a[], tetapi bukan lagi
butir tumpukan
    bool push(int x) {
        if (isFull()) return false;
        a[++top] = x;
        return true;
    }
    int size() const { return top + 1; }
};

// ---------- list yang dipakai Soal Latihan 8.2 sampai 8.4 dan 8.10 ----------
struct Node {
    int data;
    Node *next;
};

Node *build(const int *v, int n) {
    Node *head = nullptr;
    for (int i = n - 1; i >= 0; i--) head = new Node{v[i], head};
    return head;
}

void print(const char *label, const Node *p) {
    cout << label << "[";
    for (; p != nullptr; p = p->next) cout << p->data << (p->next ? " " : "");
    cout << "]\n";
}
```

Kode 8.37 (Lanjutan) *ex08_solutions.cpp: soal latihan pemrograman Bab 8 dalam C++.*

```
void destroy(Node *p) {
    while (p != nullptr) {
        Node *after = p->next;
        delete p;
        p = after;
    }
}

// ---------- Soal Latihan 8.2: panjang dan jumlah, satu kali penelusuran ----------
void lengthAndSum(const Node *head, int &length, long long &sum) {
    length = 0;
    sum = 0;
    for (const Node *p = head; p != nullptr; p = p->next) {
        length++;
        sum += p->data;
    }
}

// ---------- Soal Latihan 8.3: simpul tengah, pointer lambat dan cepat ----------
const Node *middle(const Node *head) {
    const Node *slow = head, *fast = head;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;           // satu langkah
        fast = fast->next->next;     // dua langkah
    }
    return slow;                     // untuk panjang genap: yang kedua dari dua simpul
tengah
}

// ---------- Soal Latihan 8.4: menghapus simpul ke-k (k = 1 adalah yang pertama) ---------
-
bool deleteKth(Node *&head, int k) {
    if (head == nullptr || k < 1) return false;
    if (k == 1) {
        Node *gone = head;
        head = head->next;
        delete gone;
        return true;
    }
    Node *before = head;             // berjalan ke simpul k-1
    for (int i = 1; i < k - 1 && before != nullptr; i++) before = before->next;
    if (before == nullptr || before->next == nullptr) return false;   // tidak ada simpul k
    Node *gone = before->next;
    before->next = gone->next;       // lompati simpul k ...
    delete gone;                     // ... lalu bebaskan
    return true;
}
```

Kode 8.37 (Lanjutan) *ex08_solutions.cpp: soal latihan pemrograman Bab 8 dalam C++.*

```
// ---------- Soal Latihan 8.5: palindrom dengan stack dan queue ----------
bool isPalindrome(const string &text) {
    stack<char> s;
    queue<char> q;
    for (char c : text) {
        if (isalpha(static_cast<unsigned char>(c))) {
            char lower = static_cast<char>(tolower(static_cast<unsigned char>(c)));
            s.push(lower);           // stack mengembalikan huruf-huruf secara terbalik
            q.push(lower);           // queue mengembalikannya secara berurutan
        }
    }
    while (!s.empty()) {
        if (s.top() != q.front()) return false;
        s.pop();
        q.pop();
    }
    return true;
}

// ---------- Soal Latihan 8.10: buang duplikat dari list TERURUT dalam satu penelusuran --
--------
void removeSortedDuplicates(Node *head) {
    Node *cur = head;
    while (cur != nullptr && cur->next != nullptr) {
        if (cur->next->data == cur->data) {
            Node *dup = cur->next;
            cur->next = dup->next;   // nilai yang sama selalu bertetangga dalam list
terurut
            delete dup;
        } else {
            cur = cur->next;
        }
    }
}

int main() {
    cout << boolalpha;
    Stack s;
    for (int v = 1; v <= MAX; v++) s.push(v);
    cout << "8.1  after 1000 pushes: isFull = " << s.isFull() << ", push(0) = " <<
s.push(0) << "\n";
    s.clear();
    cout << "8.1  after clear(): isEmpty = " << s.isEmpty() << ", size = " << s.size() <<
"\n";
```

Kode 8.37 (Lanjutan) *ex08_solutions.cpp: soal latihan pemrograman Bab 8 dalam C++.*

```
    const int v[] = {4, 8, 15, 16, 23, 42};
    Node *list = build(v, 6);
    int len;
    long long sum;
    lengthAndSum(list, len, sum);
    cout << "8.2  length = " << len << ", sum = " << sum << "\n";
    cout << "8.3  middle of 6 nodes = " << middle(list)->data;
    Node *odd = build(v, 5);
    cout << ", middle of 5 nodes = " << middle(odd)->data << "\n";
    destroy(odd);

    deleteKth(list, 3);
    print("8.4  after deleting node 3: ", list);
    deleteKth(list, 1);
    print("8.4  after deleting node 1: ", list);
    cout << "8.4  deleting node 9: " << deleteKth(list, 9) << "\n";
    destroy(list);

    cout << "8.5  \"Kasur ini rusak\" -> " << isPalindrome("Kasur ini rusak") << "\n";
    cout << "8.5  \"stack\" -> " << isPalindrome("stack") << "\n";

    const int w[] = {1, 1, 2, 3, 3, 3, 7, 7};
    Node *sorted = build(w, 8);
    removeSortedDuplicates(sorted);
    print("8.10 ", sorted);
    destroy(sorted);
    return 0;
}
```

Kode 8.38 `Ex08Solutions.java`: soal latihan pemrograman Bab 8 dalam Java.

```java
// Ex08Solutions.java - penyelesaian soal latihan pemrograman Bab 8.
import java.util.ArrayDeque;

public class Ex08Solutions {

    // ---------- Soal Latihan 8.1: isFull() dan clear() untuk stack larik ----------
    static final int MAX = 1000;

    static class Stack {
        private int top = -1;
        private final int[] a = new int[MAX];

        boolean isEmpty() { return top < 0; }
        boolean isFull() { return top == MAX - 1; }
        void clear() { top = -1; }         // nilai lama tetap di a[], tetapi bukan lagi
butir tumpukan
        boolean push(int x) {
            if (isFull()) return false;
            a[++top] = x;
            return true;
        }
        int size() { return top + 1; }
    }

    // ---------- list yang dipakai Soal Latihan 8.2 sampai 8.4 dan 8.10 ----------
    static class Node {
        int data;
        Node next;

        Node(int data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    static Node build(int... v) {
        Node head = null;
        for (int i = v.length - 1; i >= 0; i--) head = new Node(v[i], head);
        return head;
    }

    static String chain(Node p) {
        StringBuilder b = new StringBuilder("[");
        for (; p != null; p = p.next) b.append(p.data).append(p.next != null ? " " : "");
        return b.append("]").toString();
    }
```

Kode 8.38 (Lanjutan) *Ex08Solutions.java: soal latihan pemrograman Bab 8 dalam Java.*

```
    // ---------- Soal Latihan 8.2: panjang dan jumlah, satu kali penelusuran (dikembalikan
bersama) ----------
    static long[] lengthAndSum(Node head) {
        long length = 0, sum = 0;
        for (Node p = head; p != null; p = p.next) {
            length++;
            sum += p.data;
        }
        return new long[] {length, sum};
    }

    // ---------- Soal Latihan 8.3: simpul tengah, referensi lambat dan cepat ----------
    static Node middle(Node head) {
        Node slow = head, fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;              // satu langkah
            fast = fast.next.next;         // dua langkah
        }
        return slow;                       // untuk panjang genap: yang kedua dari dua
simpul tengah
    }

    // ---------- Soal Latihan 8.4: menghapus simpul ke-k (k = 1 adalah yang pertama) -----
-----
    // Mengembalikan simpul pertama (mungkin yang baru); deleted[0] melaporkan
keberhasilan.
    static Node deleteKth(Node head, int k, boolean[] deleted) {
        deleted[0] = false;
        if (head == null || k < 1) return head;
        if (k == 1) {
            deleted[0] = true;
            return head.next;
        }
        Node before = head;                // berjalan ke simpul k-1
        for (int i = 1; i < k - 1 && before != null; i++) before = before.next;
        if (before == null || before.next == null) return head;   // tidak ada simpul k
        before.next = before.next.next;    // lompati simpul k
        deleted[0] = true;
        return head;
    }
```

Kode 8.38 (Lanjutan) *Ex08Solutions.java: soal latihan pemrograman Bab 8 dalam Java.*

```
    // ---------- Soal Latihan 8.5: palindrom dengan stack dan queue ----------
    static boolean isPalindrome(String text) {
        ArrayDeque<Character> stack = new ArrayDeque<>();
        ArrayDeque<Character> queue = new ArrayDeque<>();
        for (char c : text.toCharArray()) {
            if (Character.isLetter(c)) {
                char lower = Character.toLowerCase(c);
                stack.push(lower);         // push/pop di depan: sebuah stack
                queue.addLast(lower);      // tambah di belakang, ambil dari depan: sebuah
queue
            }
        }
        while (!stack.isEmpty()) {
            if (!stack.pop().equals(queue.pollFirst())) return false;
        }
        return true;
    }

    // ---------- Soal Latihan 8.10: buang duplikat dari list TERURUT dalam satu
penelusuran ----------
    static void removeSortedDuplicates(Node head) {
        Node cur = head;
        while (cur != null && cur.next != null) {
            if (cur.next.data == cur.data) {
                cur.next = cur.next.next;  // nilai yang sama selalu bertetangga dalam list
terurut
            } else {
                cur = cur.next;
            }
        }
    }

    public static void main(String[] args) {
        Stack s = new Stack();
        for (int v = 1; v <= MAX; v++) s.push(v);
        System.out.println("8.1  after 1000 pushes: isFull = " + s.isFull() + ", push(0) =
" + s.push(0));
        s.clear();
        System.out.println("8.1  after clear(): isEmpty = " + s.isEmpty() + ", size = " +
s.size());

        Node list = build(4, 8, 15, 16, 23, 42);
        long[] ls = lengthAndSum(list);
        System.out.println("8.2  length = " + ls[0] + ", sum = " + ls[1]);
        System.out.println("8.3  middle of 6 nodes = " + middle(list).data
                + ", middle of 5 nodes = " + middle(build(4, 8, 15, 16, 23)).data);
```

Kode 8.38 (Lanjutan) *Ex08Solutions.java: soal latihan pemrograman Bab 8 dalam Java.*

```
        boolean[] done = new boolean[1];
        list = deleteKth(list, 3, done);
        System.out.println("8.4  after deleting node 3: " + chain(list));
        list = deleteKth(list, 1, done);
        System.out.println("8.4  after deleting node 1: " + chain(list));
        list = deleteKth(list, 9, done);
        System.out.println("8.4  deleting node 9: " + done[0]);

        System.out.println("8.5  \"Kasur ini rusak\" -> " + isPalindrome("Kasur ini
rusak"));
        System.out.println("8.5  \"stack\" -> " + isPalindrome("stack"));

        Node sorted = build(1, 1, 2, 3, 3, 3, 7, 7);
        removeSortedDuplicates(sorted);
        System.out.println("8.10 " + chain(sorted));
    }
}
```

Kode 8.39 `ex08_solutions.py`: soal latihan pemrograman Bab 8 dalam Python.

```
# ex08_solutions.py - penyelesaian soal latihan pemrograman Bab 8.
from collections import deque

# ---------- Soal Latihan 8.1: is_full() dan clear() untuk stack larik ----------
MAX = 1000


class Stack:
    def __init__(self):
        self._top = -1
        self._a = [0] * MAX

    def is_empty(self):
        return self._top < 0

    def is_full(self):
        return self._top == MAX - 1

    def clear(self):
        self._top = -1          # nilai lama tetap di _a, tetapi bukan lagi butir tumpukan

    def push(self, x):
        if self.is_full():
            return False
        self._top += 1
        self._a[self._top] = x
        return True

    def size(self):
        return self._top + 1


# ---------- list yang dipakai Soal Latihan 8.2 sampai 8.4 dan 8.10 ----------
class Node:
    __slots__ = ("data", "next")

    def __init__(self, data, next=None):
        self.data = data
        self.next = next
```

Kode 8.39 (Lanjutan) *ex08_solutions.py: soal latihan pemrograman Bab 8 dalam Python.*

```
def build(*values):
    head = None
    for v in reversed(values):
        head = Node(v, head)
    return head


def chain(p):
    out = []
    while p is not None:
        out.append(str(p.data))
        p = p.next
    return "[" + " ".join(out) + "]"


# ---------- Soal Latihan 8.2: panjang dan jumlah, satu kali penelusuran ----------
def length_and_sum(head):
    length, total = 0, 0
    p = head
    while p is not None:
        length += 1
        total += p.data
        p = p.next
    return length, total


# ---------- Soal Latihan 8.3: simpul tengah, nama lambat dan cepat ----------
def middle(head):
    slow = fast = head
    while fast is not None and fast.next is not None:
        slow = slow.next            # satu langkah
        fast = fast.next.next       # dua langkah
    return slow                     # untuk panjang genap: yang kedua dari dua simpul
tengah


# ---------- Soal Latihan 8.4: menghapus simpul ke-k (k = 1 adalah yang pertama) ----------
def delete_kth(head, k):
    """Mengembalikan (simpul pertama yang baru, apakah ada simpul yang dihapus)."""
    if head is None or k < 1:
        return head, False
    if k == 1:
        return head.next, True
    before = head                   # berjalan ke simpul k-1
    i = 1
    while i < k - 1 and before is not None:
        before = before.next
```

Kode 8.39 (Lanjutan) *ex08_solutions.py: soal latihan pemrograman Bab 8 dalam Python.*

```
        i += 1
    if before is None or before.next is None:
        return head, False          # tidak ada simpul k
    before.next = before.next.next  # lompati simpul k
    return head, True


# ---------- Soal Latihan 8.5: palindrom dengan stack dan queue ----------
def is_palindrome(text):
    stack, queue = [], deque()
    for c in text:
        if c.isalpha():
            stack.append(c.lower())     # stack mengembalikan huruf-huruf secara terbalik
            queue.append(c.lower())     # queue mengembalikannya secara berurutan
    while stack:
        if stack.pop() != queue.popleft():
            return False
    return True


# ---------- Soal Latihan 8.10: buang duplikat dari list TERURUT dalam satu penelusuran ---
-------
def remove_sorted_duplicates(head):
    cur = head
    while cur is not None and cur.next is not None:
        if cur.next.data == cur.data:
            cur.next = cur.next.next    # nilai yang sama selalu bertetangga dalam list
terurut
        else:
            cur = cur.next


def main():
    s = Stack()
    for v in range(1, MAX + 1):
        s.push(v)
    print(f"8.1  after 1000 pushes: isFull = {str(s.is_full()).lower()}, "
          f"push(0) = {str(s.push(0)).lower()}")
    s.clear()
    print(f"8.1  after clear(): isEmpty = {str(s.is_empty()).lower()}, size = {s.size()}")

    lst = build(4, 8, 15, 16, 23, 42)
    length, total = length_and_sum(lst)
    print(f"8.2  length = {length}, sum = {total}")
    print(f"8.3  middle of 6 nodes = {middle(lst).data}, "
          f"middle of 5 nodes = {middle(build(4, 8, 15, 16, 23)).data}")
```

Kode 8.39 (Lanjutan) *ex08_solutions.py: soal latihan pemrograman Bab 8 dalam Python.*

```
    lst, _ = delete_kth(lst, 3)
    print("8.4  after deleting node 3:", chain(lst))
    lst, _ = delete_kth(lst, 1)
    print("8.4  after deleting node 1:", chain(lst))
    lst, done = delete_kth(lst, 9)
    print("8.4  deleting node 9:", str(done).lower())

    print('8.5  "Kasur ini rusak" ->', str(is_palindrome("Kasur ini rusak")).lower())
    print('8.5  "stack" ->', str(is_palindrome("stack")).lower())

    srt = build(1, 1, 2, 3, 3, 3, 7, 7)
    remove_sorted_duplicates(srt)
    print("8.10", chain(srt))


if __name__ == "__main__":
    main()
```

```
8.1  after 1000 pushes: isFull = true, push(0) = false
8.1  after clear(): isEmpty = true, size = 0
8.2  length = 6, sum = 108
8.3  middle of 6 nodes = 16, middle of 5 nodes = 15
8.4  after deleting node 3: [4 8 16 23 42]
8.4  after deleting node 1: [8 16 23 42]
8.4  deleting node 9: false
8.5  "Kasur ini rusak" -> true
8.5  "stack" -> false
8.10 [1 2 3 7]
```

Beberapa kata untuk masing-masing. **(1)** `isFull()` adalah `top == MAX - 1`, bernilai benar sesudah 1000 kali *push*, dan *push* ke-1001 gagal. `clear()` hanya mengembalikan `top` ke -1; nilai-nilai lama tetap berada di larik, tetapi menurut invarian nilai-nilai itu bukan lagi butir, dan *push* berikutnya akan menimpanya. Biayanya $O(1)$. **(2)** Panjang 6, jumlah 108; satu kali penelusuran, $\Theta(n)$. Jumlahnya bertipe `long long` agar list panjang berisi nilai-nilai besar tidak membuatnya meluap. **(3)** Ketika `fast` mencapai ujung, `slow` baru bergerak separuhnya: simpul tengah list enam simpul adalah 16 (yang kedua dari 15 dan 16), dan list lima simpul 15. **(4)** Menghapus simpul 3 membuang 15, menghapus simpul 1 membuang 4 (dan mengubah simpul pertama, sehingga C++ menerima `Node *&`), dan simpul 9 tidak ada, sehingga tidak ada yang berubah dan fungsi mengembalikan `false`. Penelusurannya berhenti di simpul $k-1$, sang pendahulu, seperti pada Soal 8.5. **(5)** Stack mengembalikan huruf secara terbalik dan queue secara berurutan; frasa itu palindrom tepat ketika kedua urutan sama. “Kasur ini rusak” palindrom; “stack” bukan. **(10)** Bandingkan setiap simpul dengan penerusnya; lepaskan penerus itu jika sama, dan maju hanya jika berbeda. Hasilnya 1, 2, 3, 7.

Soal Latihan 6. Simpul-simpulnya adalah simpul 0 `f 1 o`, simpul 1 `f 2 k`, simpul 2 `f 2 y`, simpul 3 `b 1 a`, dan simpul 4 sebagai akhir. Penelusuran mencetak `o` di simpul 0 lalu pindah ke simpul 1; mencetak `k` lalu maju 2 ke simpul 3; mencetak `a` lalu mundur 1 ke simpul 2; mencetak `y` lalu maju 2 ke simpul 4, akhirnya. Keluarannya `okay`.

Soal Latihan 7. *Enqueue* 1, 2, 3: slot `1 2 3`, `front` 0, `count` 3 (penuh). *Dequeue* mengembalikan 1: `- 2 3`, `front` 1, `count` 2. *Enqueue* 4 masuk ke $(1+2) \bmod 3 = 0$: `4 2 3`, `count` 3. *Dequeue* mengembalikan 2: `4 - 3`, `front` 2, `count` 2. *Dequeue* mengembalikan 3: `4 - -`, `front` $(2+1) \bmod 3 = 0$, `count` 1. *Enqueue* 5 masuk ke 1 dan 6 ke 2: `4 5 6`, `front` 0, `count` 3. Butir-butir itu akan dilayani dengan urutan 4, 5, 6.

Soal Latihan 8. `a` ke keluaran; `*` di-*push*; `(` di-*push*; `b` keluar; `+` di-*push* di atas `(` (derajat 1 > 0); `c` keluar; `*` di-*push* (2 > 1); `d` keluar; `)` mengeluarkan `*` dan `+` lalu membuang `(`; keluaran kini `abcd*+`, stack berisi `*`; `-` mengeluarkan `*` ($2 \geq 1$) lalu di-*push*; `e` keluar; akhir ekspresi mengeluarkan `-`. Hasilnya `abcd*+*e-`. Untuk `234*+5-`: $3 \cdot 4 = 12$, $2 + 12 = 14$, $14 - 5 = 9$.

Soal Latihan 9. `*(p + 2) += 5` membuat `v[2]` bernilai 35. `p++` memindahkan `p` ke `v[1]` (satu `int` lebih jauh, menurut aritmetika pointer). `*p = *p * 2` membuat `v[1]` bernilai 40. `q` menunjuk ke `v[3]`, yang bernilai 40. Keluarannya `80 2`: $40 + 40$, dan `q - p` adalah banyaknya `int` di antara keduanya, $3 - 1 = 2$. Larik berakhir sebagai 10, 40, 35, 40.

Soal Latihan 11. `enqueue(x)` mem-*push* `x` ke `in`. `dequeue()` mem-*pop* dari `out`; hanya ketika `out` kosong ia lebih dulu memindahkan *setiap* butir dari `in` ke `out`, mem-*pop* dari yang satu dan mem-*push* ke yang lain, yang membalik urutannya sehingga butir tertua berakhir di puncak `out`. Setiap butir di-*push* ke `in` sekali, di-*pop* dari `in` sekali, di-*push* ke `out` sekali, dan di-*pop* dari `out` sekali: empat operasi $O(1)$ sepanjang hidupnya. Jadi k operasi berbiaya total $O(k)$, meskipun satu `dequeue` dapat berbiaya $O(n)$ ketika harus memindahkan semuanya. Rata-rata yang diambil atas seluruh urutan seperti ini disebut biaya *teramortisasi* (*amortized*) [1].

Soal Latihan 12. Uji sebelum menyimpan: `while (cin >> data && data != -1) slist_pushBack(myList, data);`. Baris pertama menjadi `1 2 3 4 1 5 6 3 2` dan baris kedua `1 2 3 4 5 6`; `slist_popBack` tidak lagi diperlukan. Contoh keluaran soal memperlihatkan -1 pada baris pertama, dan dari situlah kita tahu bahwa perulangan aslinya menyimpan penanda itu.

Soal Latihan 13. Hanya simpulnya yang berubah: ia memuat `int orderNo` dan `double kg` (atau sebuah `struct Order` kecil) alih-alih `string`, dan `enqueue` menerima sebuah `Order`. Logika pointer pada `enqueue` dan `dequeue`, serta penyetelan ulang `tail`, sama sekali tidak berubah, karena keduanya tidak pernah melihat datanya. `peek` mengembalikan `head->data` dan tidak menggeser apa pun; pada queue kosong ia tidak boleh mengikuti `head`, yang bernilai `nullptr`, sehingga ia melempar `underflow_error` yang sama dengan `dequeue`, dan pemanggil yang cermat bertanya `isEmpty()` lebih dulu.

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[3] Free Software Foundation, “Program Instrumentation Options,” in *Using the GNU Compiler Collection (GCC)*, GCC 16.2, Aug. 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://gcc.gnu.org/onlinedocs/

[4] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in *Proc. USENIX Annual Technical Conf.*, Boston, MA, USA, 2012, pp. 309–318.

[5] Valgrind Developers, *Valgrind User Manual*, release 3.27.1, May 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://valgrind.org/docs/manual/

[6] B. Stroustrup, *A Tour of C++*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2022.

[7] Oracle, “Class ArrayDeque,” *Java SE 25 API Specification*, and “The Java Language Specification, Java SE 25 Edition,” §15.17.3 and §15.12.4. [Online]. Available: https://docs.oracle.com/en/java/javase/25/

[8] Python Software Foundation, “Data model,” “Binary arithmetic operations,” “collections.deque,” and “Programming FAQ: How do I write a function with output parameters?,” *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/

[9] cppreference.com, “Arithmetic operators,” “Reference declaration,” “new expression,” and “std::stack.” Accessed: Sep. 24, 2026. [Online]. Available: https://en.cppreference.com/w/cpp

[10] E. W. Dijkstra, “Algol 60 translation: An Algol 60 translator for the X1 and making a translator for Algol 60,” Mathematisch Centrum, Amsterdam, Report MR 35, 1961.

[11] M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, 4th ed. Boston, MA, USA: Pearson, 2014.

[12] A. Newell and J. C. Shaw, “Programming the Logic Theory Machine,” in *Proc. Western Joint Computer Conf.*, Los Angeles, CA, USA, 1957, pp. 230–240.

[13] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Python*. Hoboken, NJ, USA: Wiley, 2013.

[14] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011.

Stack, queue, linked list, langkah *merge*, dan analisis teramortisasi dibahas dalam Cormen dkk. [1], dan alokasi bertaut beserta sejarahnya dalam Knuth [2]; linked list sendiri diperkenalkan oleh Newell dan Shaw untuk bahasa IPL yang mereka bangun bersama Simon di RAND pada pertengahan 1950-an [12]. Aturan C++ tentang referensi, `new` dan `delete`, serta sisa bagi bilangan negatif mengikuti Stroustrup [6] dan cppreference [9]; aturan Java tentang `%` dan pengiriman argumen mengikuti spesifikasi bahasanya [7], dan aturan Python mengikuti rujukan bahasanya [8]. Algoritme *shunting-yard* adalah karya Dijkstra [10]. Weiss [11], Goodrich dkk. [13], serta Sedgewick dan Wayne [14] menyajikan struktur-struktur yang sama berturut-turut dalam C++, Python, dan Java, dan merupakan pendamping yang baik untuk bab ini. Pemeriksa memori dijelaskan dalam manual GCC [3], manual Valgrind [5], dan makalah asli AddressSanitizer [4].

Bab 9

Double Linked List dan Linked List Sirkular

Di akhir bab ini Anda akan sanggup menjelaskan mengapa pointer kedua di setiap simpul, yang menunjuk ke belakang, menghapus dua batasan yang tidak dapat diatasi Bab 7; menyatakan invarian yang mengikat setiap tautan maju dengan pasangan mundurnya dan memeriksanya dalam kode; mendeklarasikan simpul *double linked list* (senarai berantai ganda) dalam C++, Java, dan Python serta menyebutkan dengan tepat berapa memori yang dimakannya; mengimplementasikan *double linked list* tanpa simpul kepala (dicapai lewat `head`, lalu lewat `head` dan `tail`) beserta seluruh operasi sehari-harinya (inisialisasi, uji kosong, penyisipan di depan, di belakang, sesudah simpul, sebelum simpul, dan pada posisi terurut; penghapusan di depan, di belakang, berdasarkan kunci, dan berdasarkan pointer; pencarian; serta penelusuran ke dua arah); membangun list yang sama dengan simpul kepala dan menyaksikan separuh kasus khususnya lenyap; menutup list menjadi lingkaran, mula-mula tanpa lalu dengan simpul *sentinel*, dan membuktikan bahwa *sentinel* menghapus sisa kasus khusus; memakai hasilnya untuk membangun *deque* bagi meja pengembalian Pustaka Ceria dan daftar "baru dipinjam" yang selalu tahu judul mana yang harus dibuang berikutnya; membandingkan *single linked list*, *double linked list*, dan larik operasi demi operasi, dengan ongkos yang diturunkan lalu diukur; membuktikan dengan Valgrind dan AddressSanitizer bahwa program tidak membocorkan memori dan tidak membaca memori yang sudah dibebaskan; serta memilih di antara `std::list`, `LinkedList` milik Java, dan `collections.deque` milik Python. Setiap operasi digambar sebelum dan sesudahnya, setiap ongkos diturunkan, dan setiap program lengkap, dikompilasi, dijalankan, dan diperiksa memorinya.

9.1 Dari Mana Kita Melanjutkan

Bab 7 membangun *single linked list* dan jujur tentang apa yang tidak sanggup dilakukannya. Dua batasan bertahan melewati setiap trik dalam bab itu.

Batasan pertama adalah *berjalan mundur*. Simpul *single linked list* hanya mengenal simpul sesudahnya. Begitu pointer pembantu sudah melewati sebuah simpul, satu-satunya cara untuk kembali kepadanya adalah memulai lagi dari `head` dan berjalan maju. Mencetak list dengan urutan terbalik, mundur satu langkah dari posisi sekarang, atau membandingkan sebuah simpul dengan pendahulunya, semuanya memakan satu perjalanan dari depan.

Batasan kedua adalah *menghapus di belakang*. Subbab 7.8 menambahkan pointer `tail` dan membuat penyisipan di belakang menjadi $O(1)$, dan Subbab 7.8.2 menjelaskan mengapa penghapusan di belakang tetap $\Theta(n)$: untuk menghapus simpul terakhir kita harus mengubah medan `next` milik simpul *sebelumnya*, dan simpul *single linked* tidak dapat memberi tahu simpul mana itu. Batasan yang sama bersembunyi di dalam penghapusan berdasarkan kunci: Subbab 7.6.5 membutuhkan pointer pengekor `prev` yang berjalan satu langkah di belakang `cur`, karena sebuah simpul tidak dapat menemukan pendahulunya sendiri. Bahkan ketika program sudah memegang pointer ke simpul yang ingin dihapusnya, *single linked list* harus mencari dari depan simpul yang berada di mukanya.

Bab 8 kemudian bertemu list yang simpulnya membawa pointer kedua. Soal 8.2 pada Subbab 8.4 menyusuri list dua arah maju dan mundur menurut arahan, dan Subbab 8.4.2 memberikan gagasannya secukupnya untuk menyelesaikan soal itu: setiap simpul punya `next` dan `prev`, dan setiap panah maju harus punya panah mundur pasangannya. Bab ini mengembangkan gagasan itu sepenuhnya.

Rencananya mudah dinyatakan. Kita menambahkan satu medan, `prev`, ke setiap simpul. Medan tunggal itu memungkinkan kita berjalan ke dua arah, menghapus simpul terakhir dalam $O(1)$, dan menghapus simpul *mana pun* yang pointernya kita pegang dalam $O(1)$, tanpa pencarian dan tanpa pointer pengekor. Medan itu juga menggandakan pekerjaan pointer pada setiap penyisipan dan penghapusan, menambahkan 8 bita ke setiap simpul, dan menciptakan cara baru untuk merusak list: memperbarui satu arah dan melupakan arah yang lain. Semua ini akan kita hadapi dengan kecermatan yang sama seperti di Bab 7. Lalu kita akan menutup list menjadi lingkaran dan menambahkan *sentinel*, simpul yang tidak menyimpan data dan satu-satunya tugasnya adalah melenyapkan setiap kasus khusus dalam bab ini. Dua aplikasi untuk perpustakaan Pustaka Ceria menyusul: *deque* untuk meja pengembalian dan daftar judul yang baru dipinjam.

9.2 Dua Panah di Setiap Simpul

9.2.1 Kereta yang dapat disusuri ke dua arah

Bayangkan kereta jarak jauh, jenis yang melintasi Jawa semalam suntuk. Setiap gerbong disambungkan dengan gerbong di depannya dan dengan gerbong di belakangnya, dan pintu di tiap ujung gerbong memungkinkan kondektur berjalan ke gerbong tetangga di sisi mana pun. Kondektur yang memeriksa tiket dapat berjalan dari lokomotif ke gerbong terakhir, lalu, tanpa turun dari kereta, berjalan kembali sampai ke depan. Bila sebuah gerbong harus dilepas di stasiun, petugas melepas sambungannya di kedua ujung lalu menyambungkan kedua tetangganya satu sama lain. Mereka tidak perlu berjalan ke kepala kereta untuk mengetahui gerbong mana yang berada di depan gerbong yang rusak; mereka dapat melihatnya.

Bandingkan dengan perburuan harta karun dari Subbab 7.2.1. Di sana setiap persinggahan hanya tahu di mana persinggahan *berikutnya*. Seorang anak yang berdiri di persinggahan kelima tidak dapat berjalan ke persinggahan keempat kecuali dengan memulai lagi dari gerbang. Kereta adalah perburuan harta karun dengan petunjuk kedua di setiap persinggahan: “persinggahan sebelum ini ada di balik pintu perpustakaan.”

Tiga ciri kereta itu penting bagi kita.

- **Setiap gerbong tersambung ke dua arah.** Satu sambungan menuju ke depan, satu ke belakang.
- **Sambungan dua gerbong yang bertetangga saling sepakat.** Bila gerbong A tersambung ke depan dengan gerbong B, maka B tersambung ke belakang dengan A. Kereta yang gerbong A-nya mengira B ada di depannya sementara B mengira gerbong lain yang ada di belakangnya adalah kereta yang rusak.
- **Melepas atau menambah gerbong menyentuh kedua tetangga.** Melepas gerbong di tengah mengubah dua sambungan; memasangnya mengubah empat.

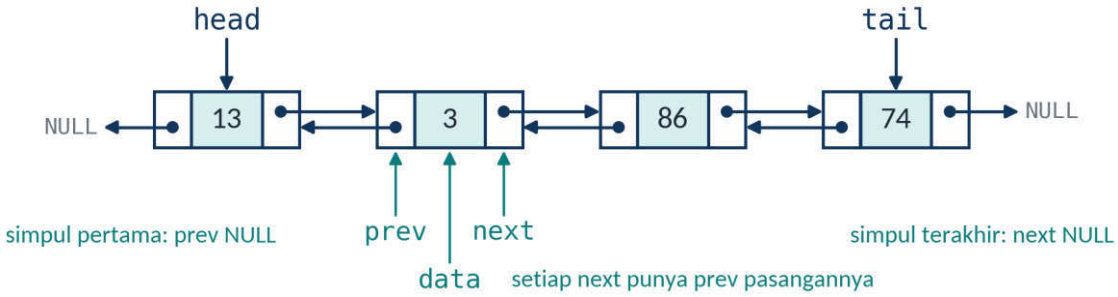
Double linked list adalah kereta ini, dibangun di dalam memori komputer.

9.2.2 Definisi dan kosakatanya

> **DEFINISI.** *Double linked list* adalah barisan simpul yang setiap simpulnya menyimpan satu elemen (*data*-nya) dan dua pointer: `next`, ke simpul sesudahnya, dan `prev`, ke simpul sebelumnya. Pada *double linked list non-circular* (*DLLNC*), `prev` milik simpul pertama dan `next` milik simpul terakhir bernilai null. List dicapai lewat pointer luar `head` ke simpul pertama, sering kali bersama pointer luar kedua `tail` ke simpul terakhir. List kosong tidak punya simpul, dan `head` (serta `tail`)-nya bernilai null.

Namanya terdiri atas tiga bagian, persis seperti di Bab 7. *Double* berarti dua medan pointer per simpul, satu untuk setiap arah. *Linked list* berarti simpul-simpul disambungkan oleh pointer, tidak disimpan berdampingan. *Non-circular* berarti kedua rantai berujung: mengikuti `next` dari simpul pertama pada akhirnya tiba di null, demikian pula mengikuti `prev` dari simpul terakhir. Subbab 9.6 menghapus kata terakhir itu.

Gambar 9.1 memperlihatkan *double linked list* yang berisi 13, 3, 86, dan 74. Kini setiap simpul digambar dengan tiga ruang: medan `prev` di kiri, data di tengah, dan medan `next` di kanan. Panah `next` keluar dari ruang kanan di bagian atasnya dan berakhir di tepi kiri simpul berikutnya; panah `prev` keluar dari ruang kiri di bagian bawahnya dan berakhir di tepi kanan simpul sebelumnya. Karena itu di antara dua tetangga selalu ada dua panah yang berlawanan arah.



Gambar 9.1 *Double linked list* yang berisi 13, 3, 86, dan 74. Setiap simpul punya medan `prev`, medan data, dan medan `next`; `head` menunjuk simpul pertama dan `tail` simpul terakhir, dan kedua ujung rantai bernilai NULL.

Ciri kedua kereta, bahwa dua tetangga harus sepakat, adalah aturan terpenting bab ini. Kita menyatakannya sebagai invarian, sifat yang harus berlaku sebelum dan sesudah setiap operasi.

> **DEFINISI.** *Invarian dua arah* dari *double linked list* non-sirkular: untuk setiap simpul p,

$$p.next \neq \text{NIL} \;\Rightarrow\; p.next.prev = p \qquad \text{dan} \qquad p.prev \neq \text{NIL} \;\Rightarrow\; p.prev.next = p \tag{9.1}$$

ditambah $head.prev = \text{NIL}$ dan $tail.next = \text{NIL}$ bila list tidak kosong. Dengan kata-kata: bila Anda melangkah maju lalu mundur, Anda kembali ke tempat semula, dan hal yang sama berlaku bila Anda melangkah mundur lalu maju. Persamaan (9.1) adalah aturan yang disebut Subbab 8.4.2 sebagai invarian list dua arah, kini ditulis untuk kedua arah.

Pembahasan klasik Knuth tentang struktur berantai menyajikan *double linked list* persis dengan cara ini, sebagai list yang setiap simpulnya membawa dua tautan, satu untuk setiap arah, yang harus saling mencerminkan [1]. Mengapa invarian itu harus dijaga? Karena setiap operasi dalam bab ini mengandalkannya tanpa memeriksanya. `backDeletion` akan percaya bahwa `tail->prev` adalah simpul sebelum yang terakhir; `deleteNode(p)` akan percaya bahwa `p->prev->next` adalah `p` itu sendiri. Bila invarian dilanggar di mana pun, operasi-operasi ini diam-diam merusak list lebih jauh. Subbab 9.4.5 menulis fungsi yang memeriksa Persamaan (9.1) untuk seluruh list, dan

Subbab 9.11 memperlihatkan apa yang terjadi di memori ketika sebuah operasi melupakan separuhnya.

> **DALAM BAHASA SEHARI-HARI.** *Double linked list* adalah rantai kotak dengan dua panah di antara setiap pasangan tetangga, satu ke setiap arah. Kedua panah di antara dua tetangga harus selalu sepakat. Anda dapat mulai dari ujung mana pun dan berjalan ke ujung yang lain.

9.2.3 Mendeklarasikan simpul, dan membuat simpul baru

Simpul dari Bab 7 mendapat tepat satu medan tambahan. Dalam gaya C tradisional, dengan nama tag dan `typedef`:

```
typedef struct MyNode {
    int data;
    struct MyNode *prev;
    struct MyNode *next;
} TNode;
```

Struktur ini tetap *self-referential* (merujuk diri sendiri): `prev` maupun `next` bertipe "pointer ke `struct MyNode`". Tidak ada yang lain yang berubah. Urutan medan tidak berpengaruh bagi program, tetapi menurut kebiasaan `prev` ditulis sebelum `next`, sesuai urutan kiri-ke-kanan pada diagram.

Simpul baru dibuat dengan `new`, dan, sebagaimana dituntut definisi bagi simpul yang belum menjadi bagian list mana pun, kedua pointernya diawali dengan null:

```
int newData;
// ...
TNode *newNode;
newNode = new TNode;
newNode->data = newData;
newNode->prev = nullptr;
newNode->next = nullptr;
```

Sesudah baris-baris ini, simpul baru adalah list sepanjang satu yang belum ditautkan oleh siapa pun. Sisa bab ini membahas cara menautkan simpul semacam itu ke dalam list, dan melepaskannya lagi, tanpa pernah melanggar Persamaan (9.1).

Kode 9.1 membangun *double linked list* tiga simpul dengan tangan, tanpa fungsi list sama sekali, menyusurinya maju dari `head` dan mundur dari `tail`, memeriksa invarian di satu simpul, mencetak ukuran sebuah simpul, dan melepaskan setiap simpul.

Kode 9.1 `dll_node.cpp`: tiga simpul *double linked* dibuat dengan `new`, ditautkan ke dua arah dengan tangan, disusuri ke dua arah, lalu dilepaskan.

```
// dll_node.cpp -- a doubly linked node: declared, created, and linked by hand.
#include <cstdio>

typedef struct MyNode {
    int data;
    struct MyNode *prev;  // address of the node before, or nullptr at the front
    struct MyNode *next;    // address of the node after, or nullptr at the back
} TNode;

TNode *makeNode(int newData) {
    TNode *newNode = new TNode;
    newNode->data = newData;
    newNode->prev = nullptr;              // a new node is linked to nothing yet
    newNode->next = nullptr;
    return newNode;
}

int main() {
    TNode *a = makeNode(13);
    TNode *b = makeNode(3);
    TNode *c = makeNode(86);

    a->next = b;  b->prev = a;            // every forward link ...
    b->next = c;  c->prev = b;            // ... gets its backward twin
    TNode *head = a;
    TNode *tail = c;

    std::printf("forward : ");
    for (TNode *aux = head; aux != nullptr; aux = aux->next)
        std::printf("%d ", aux->data);
    std::printf("\nbackward: ");
    for (TNode *aux = tail; aux != nullptr; aux = aux->prev)
        std::printf("%d ", aux->data);
    std::printf("\n");

    std::printf("head->next->prev == head : %s\n",
                head->next->prev == head ? "yes" : "no");
    std::printf("tail->prev->data         = %d\n", tail->prev->data);
    std::printf("head->prev, tail->next   = %s, %s\n",
                head->prev == nullptr ? "NULL" : "?",
                tail->next == nullptr ? "NULL" : "?");
    std::printf("sizeof(TNode)            = %zu bytes\n", sizeof(TNode));
```

Kode 9.1 (Lanjutan) *dll_node.cpp: tiga simpul double linked dibuat dengan new, ditautkan ke dua arah dengan tangan, disusuri ke dua arah, lalu dilepaskan.*

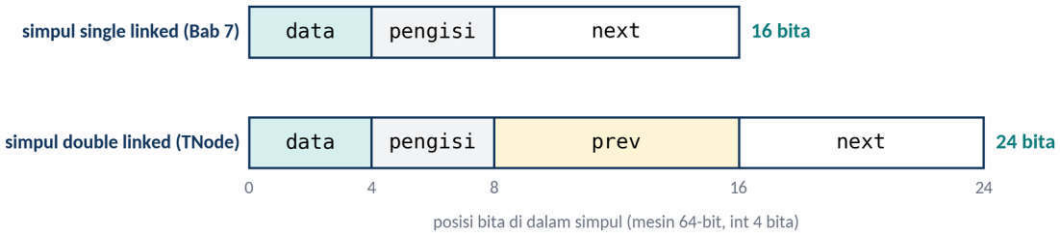
```
    int released = 0;
    while (head != nullptr) {  // save, move, delete -- as in Chapter 7
        TNode *del = head;
        head = head->next;
        delete del;
        ++released;
    }
    std::printf("nodes released           = %d\n", released);
    return 0;
}
```

```
forward : 13 3 86
backward: 86 3 13
head->next->prev == head : yes
tail->prev->data         = 3
head->prev, tail->next   = NULL, NULL
sizeof(TNode)            = 24 bytes
nodes released           = 3
```

Perhatikan dulu kedua baris penautan. Masing-masing melakukan dua penugasan, `a->next = b;` `b->prev = a;`, satu maju dan satu mundur, sehingga sesudah kedua baris itu ketiga simpul memenuhi Persamaan (9.1). Penyusuran maju adalah perulangan dari Bab 7. Penyusuran mundur adalah perulangan yang sama dengan `tail` sebagai ganti `head` dan `prev` sebagai ganti `next`; tidak ada yang baru di dalamnya kecuali arahnya. Baris keluaran `head->next->prev == head : yes` adalah Persamaan (9.1) pada simpul pertama, dibacakan: melangkah maju dari `head`, melangkah mundur, dan Anda kembali di `head`.

9.2.4 Harga pointer kedua

Keluaran juga menyatakan `sizeof(TNode) = 24 bytes`. **Gambar 9.2** memperlihatkan ke mana perginya 24 bita itu pada mesin 64-bit. `int` menempati bita 0 sampai 3. Sebuah pointer harus mulai di alamat yang habis dibagi 8, sehingga kompilator mengosongkan bita 4 sampai 7 (pengisi, *padding*) dan menaruh `prev` di bita 8 sampai 15 serta `next` di bita 16 sampai 23. Simpul *single linked* dari Subbab 7.4.4 berukuran 16 bita; pointer kedua menambahkan tepat 8.



Gambar 9.2 Tata letak memori simpul yang berisi `int` pada mesin 64-bit: 16 bita untuk simpul *single linked*, 24 bita untuk simpul *double linked*. Pengisi menjaga setiap pointer berada di alamat yang habis dibagi 8.

Secara umum, dengan notasi rumus memori dari Subbab 7.3.3 (data d bita, pointer p bita, beban pengalokasi a bita per simpul), *double linked list* berisi n elemen menempati

$$M_{\mathrm{DLL}} = n\,(d + 2p + a) \quad \text{bita, dibandingkan} \quad M_{\mathrm{SLL}} = n\,(d + p + a). \tag{9.2}$$

Selisihnya tepat $n\,p$, satu pointer per elemen, 8 MB untuk sejuta elemen. Untuk list berisi `int` ini pajak yang besar: 24 bita simpul (dan, pada pustaka C GNU, potongan pengalokasi 32 bita) untuk membawa 4 bita data. Untuk list berisi catatan buku 200 bita, besarnya 4 persen. Kaidah praktis yang sama seperti di Bab 7 berlaku, dengan tekanan lebih besar: *double linked list* berisi elemen kecil itu mahal; *double linked list* berisi catatan besar tidak.

9.2.5 Cara membaca diagram pointer dalam bab ini

Diagram dalam bab ini memperluas konvensi Subbab 7.4.5.

- Sebuah simpul punya tiga ruang: `prev`, data, `next`. Titik di ruang pointer adalah pangkal panah; panah `next` berpangkal di bagian atas dan mengarah ke kanan, panah `prev` berpangkal di bagian bawah dan mengarah ke kiri. Pointer yang bernilai null berakhir pada kata NULL.
- Warnanya sama dengan Bab 7: **teal** untuk simpul biasa, **teal lebih gelap** untuk simpul yang sedang dikunjungi pointer pembantu, **amber** untuk simpul yang sedang disisipkan, **merah bertepi putus-putus** untuk simpul yang sedang dihapus, dan **abu-abu** untuk simpul kepala atau *sentinel*, yang tidak menyimpan data.
- **Panah amber** adalah pointer yang baru saja diubah oleh langkah yang digambar, dan angka amber kecilnya adalah urutan kode mengubahnya. **Panah abu-abu putus-putus** adalah tautan yang baru saja digantikan. **Panah merah** adalah tautan yang salah.

- Simpul yang sedang disisipkan atau dihapus sering digambar *di bawah* barisan, supaya panah yang melompatinya dan panah yang mencapainya sama-sama terlihat.
- Pada list sirkular, tautan yang melingkar dari simpul terakhir ke simpul pertama berjalan *di bawah* barisan: yang lebih atas dari dua garis panjang itu adalah tautan `next`, yang lebih bawah adalah tautan `prev`.

9.3 Double Linked List dengan Pointer Head

Kita mulai dari versi lengkap yang paling kecil: *double linked list* non-sirkular yang dicapai lewat satu pointer luar, `head`, yang selalu menunjuk simpul pertama atau bernilai null ketika list kosong. Inilah list *tanpa simpul kepala*: setiap simpulnya menyimpan data, dan simpul data pertama adalah simpul pertama. Aturan *head* dari Subbab 7.5 tetap berlaku: jangan pernah memindahkan `head` untuk menyusuri list; susurilah dengan pointer pembantu.

Fungsi-fungsinya menerima `head` secara referensi (`TNode *&head`) bila fungsi itu dapat mengubah simpul mana yang pertama, dan secara nilai sebagai `const TNode *` bila hanya membaca. Untuk setiap operasi kita berikan gagasannya, prosedurnya, gambar sebelum dan sesudahnya, dan ongkosnya. Program lengkapnya menyusul di Subbab 9.3.8.

9.3.1 `init` dan `isEmpty`

Tidak ada yang berubah dari Bab 7: list kosong adalah list yang `head`-nya null.

```
void init(TNode *&head) { head = nullptr; }
bool isEmpty(const TNode *head) { return head == nullptr; }
```

Keduanya berjalan dalam waktu $O(1)$. List yang tidak pernah diinisialisasi bukanlah list kosong; ia sampah, dan perulangan pertama atasnya akan berkelana ke memori acak.

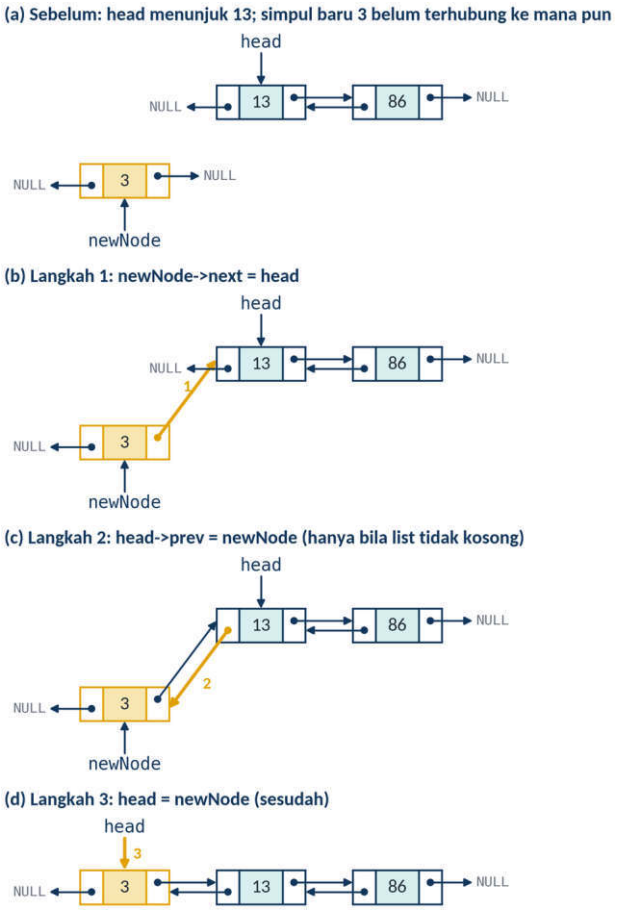
9.3.2 `frontInsertion`: simpul pertama yang baru

Pada *single linked list*, penyisipan di depan mengubah dua pointer: `next` milik simpul baru dan `head`. Pada *double linked list* ia mengubah tiga, karena simpul pertama yang lama kini harus menunjuk *balik* ke simpul baru. **Pseudokode 9.1** memberikan prosedurnya; *L.head* adalah pointer *head* dari list *L*.

Pseudokode 9.1 `FRONT-INSERTION` pada *double linked list* yang dicapai lewat `head`: tiga penugasan pointer, dan yang tengah hanya bila list tidak kosong.

```
FRONT-INSERTION(L, x):
 1.  z ← simpul baru;  z.data ← x;  z.prev ← NIL
 2.  z.next ← L.head            // langkah 1: simpul baru menunjuk maju ke simpul pertama lama
 3.  if L.head ≠ NIL
 4.      L.head.prev ← z        // langkah 2: simpul pertama lama menunjuk balik ke simpul baru
 5.  L.head ← z                 // langkah 3: simpul baru menjadi simpul pertama
```

Gambar 9.3 memperlihatkan ketiga langkah pada list 13, 86 ketika 3 disisipkan. Pada panel (a) simpul baru sudah ada tetapi belum tertaut ke mana pun: kedua pointernya NULL. Langkah 1 (panel b) membuatnya menunjuk maju ke 13. Langkah 2 (panel c) membuat 13 menunjuk balik kepadanya; pada saat ini simpul baru sudah tertaut benar dengan 13 ke dua arah, tetapi list belum mengenalnya, karena `head` masih menunjuk 13. Langkah 3 (panel d) memindahkan `head`.



Gambar 9.3 `frontInsertion(3)` pada list 13, 86: (a) sebelum, (b) simpul baru menunjuk maju ke simpul pertama lama, (c) simpul pertama lama menunjuk balik ke simpul baru, (d) `head` pindah ke simpul baru.

Urutan penting di satu tempat. Langkah 3 harus terakhir, karena langkah 1 dan 2 sama-sama membaca `head` untuk menemukan simpul pertama lama; bila `head` pindah lebih dulu, keduanya akan menemukan simpul baru itu sendiri. Langkah 1 dan 2 boleh dipertukarkan.

List kosong. Ketika list kosong, `L.head` bernilai NIL, sehingga langkah 1 membuat `z.next` bernilai NIL, uji di baris 3 gagal, dan langkah 3 menjadikan `z` satu-satunya simpul, dengan kedua pointernya NIL. Uji di baris 3 adalah *kasus khusus* pertama dalam bab ini: pada *single linked list* penyisipan di depan tidak membutuhkannya (Soal 7.10), tetapi di sini simpul pertama lama harus diperbarui, dan pada list kosong tidak ada simpul pertama lama. Kita akan menghitung uji-uji semacam ini dengan cermat, karena Subbab 9.7 melenyapkan semuanya.

Ongkos. Sejumlah tetap penugasan dan satu perbandingan: $O(1)$.

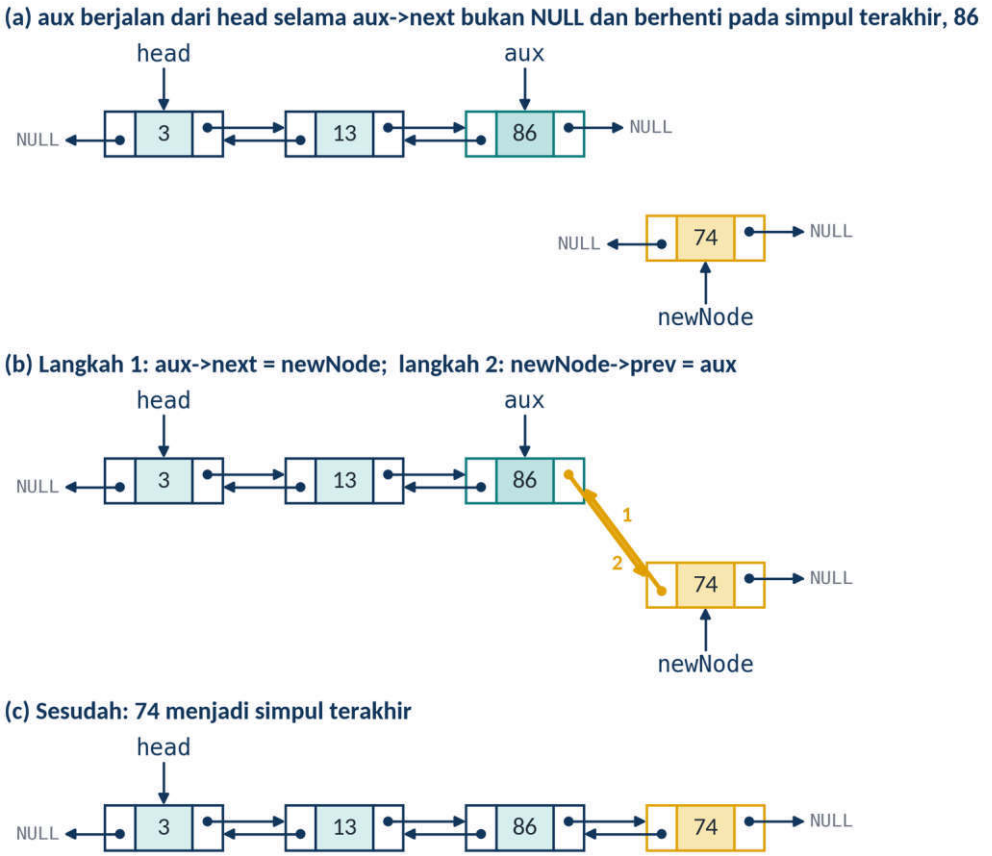
9.3.3 `backInsertion`: simpul terakhir yang baru

Dengan `head` saja, list tidak tahu di mana simpul terakhirnya, sehingga penyisipan di belakang tetap berjalan, persis seperti di Subbab 7.5.3: `aux` mulai di `head` dan maju selama `aux.next` bukan NIL. Ketika berhenti, dua penugasan menautkan simpul baru, satu untuk setiap arah.

Pseudokode 9.2 `BACK-INSERTION` dengan pointer *head* saja: berjalan ke simpul terakhir, lalu menautkan ke dua arah.

```
BACK-INSERTION(L, x):
 1.  z ← simpul baru;  z.data ← x;  z.prev ← NIL;  z.next ← NIL
 2.  if L.head = NIL              // list kosong: simpul baru juga yang pertama
 3.      L.head ← z
 4.  else
 5.      aux ← L.head
 6.      while aux.next ≠ NIL
 7.          aux ← aux.next
 8.      aux.next ← z             // langkah 1: simpul terakhir lama menunjuk maju ke z
 9.      z.prev ← aux             // langkah 2: z menunjuk balik ke simpul terakhir lama
```

Gambar 9.4 menelusuri penyisipan 74 ke dalam 3, 13, 86. Perjalanannya (panel a) sama dengan perjalanan di Bab 7. Satu-satunya baris baru adalah baris 9: tanpanya, penyusuran maju akan menampilkan 74 dengan benar, sedangkan penyusuran mundur yang dimulai di 74 langsung berhenti, karena `prev` milik 74 masih NIL.



Gambar 9.4 `backInsertion(74)` pada list 3, 13, 86: (a) `aux` berhenti di simpul terakhir, (b) dua tautan dipasang, satu untuk setiap arah, (c) sesudahnya.

Ongkos. $\Theta(n)$ untuk perjalanannya, sama dengan *single linked list*. Pointer kedua tidak menolong di sini, karena perjalanan dimulai dari depan dan hanya pointer ke *belakang* yang dapat memperpendeknya. Subbab 9.4 menambahkan pointer itu.

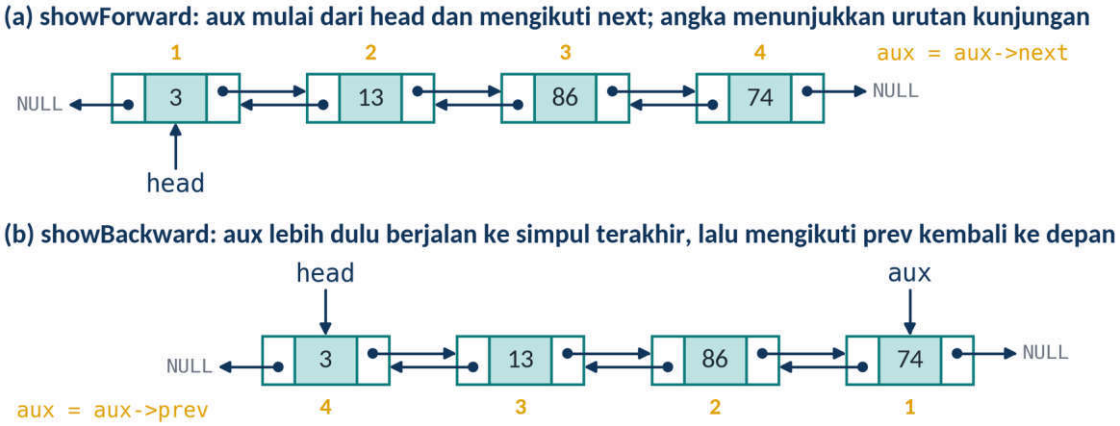
9.3.4 Menyusuri ke dua arah

Penyusuran maju adalah perulangan Subbab 7.5.4, tanpa perubahan. Penyusuran mundur adalah operasi yang baru. Dengan pointer `head` saja, kita harus lebih dulu menemukan simpul terakhir, lalu mengikuti `prev`:

Pseudokode 9.3 `SHOW-BACKWARD` dengan pointer *head* saja: berjalan ke simpul terakhir, lalu mengunjungi setiap simpul dari yang terakhir sampai yang pertama.

```
SHOW-BACKWARD(L):
 1.  if L.head = NIL
 2.      return
 3.  aux ← L.head
 4.  while aux.next ≠ NIL        // tahap 1: berjalan maju ke simpul terakhir
 5.      aux ← aux.next
 6.  while aux ≠ NIL             // tahap 2: kunjungi setiap simpul, mengikuti prev
 7.      kunjungi aux
 8.      aux ← aux.prev
```

Gambar 9.5 memperlihatkan kedua penyusuran atas list 3, 13, 86, 74 beserta urutan kunjungannya. Perhatikan lagi dua syarat perulangan dari Subbab 7.5.4: tahap 1 menguji `aux.next`, karena ia harus *berhenti pada* simpul terakhir, dan tahap 2 menguji `aux`, karena ia harus *mengunjungi* setiap simpul, termasuk yang pertama, dan selesai ketika `aux` sudah melangkah melewati simpul pertama ke NIL.



Gambar 9.5 Menyusuri list 3, 13, 86, 74: (a) maju dari `head`, mengikuti `next`; (b) mundur, mengikuti `prev` dari simpul terakhir. Angka-angka menunjukkan urutan kunjungan.

Ongkos. Penyusuran maju $\Theta(n)$. Penyusuran mundur dengan `head` saja juga $\Theta(n)$, sekitar $2n$ langkah: $n-1$ untuk menemukan simpul terakhir dan n untuk kembali. *Single linked list* tidak dapat berbuat lebih baik dari $\Theta(n)$ bahkan untuk mencetak *satu* elemen pada posisi terbalik; mencetak seluruh list secara mundur memakan waktu $\Theta(n^2)$ atau memori tambahan $\Theta(n)$ untuk sebuah *stack*. Di sini cukup dua perjalanan tanpa memori tambahan sama sekali. Dengan pointer `tail`, tahap 1 lenyap.

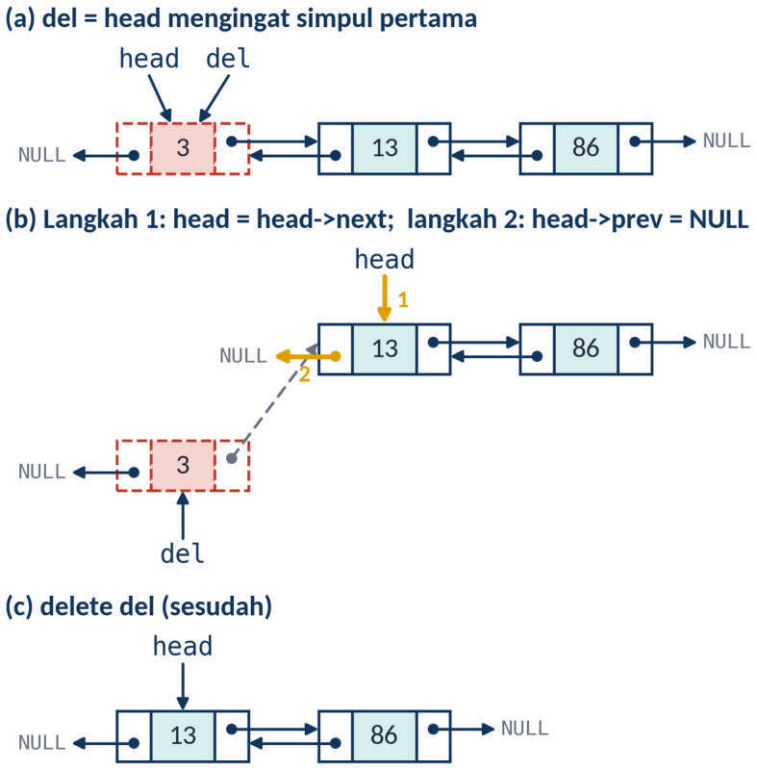
9.3.5 `frontDeletion`: membuang simpul pertama

Penghapusan di depan mengikuti aturan Subbab 7.5.5 (simpan, pindahkan, hapus) dan menambahkan satu penugasan: simpul pertama yang baru harus melupakan pendahulunya, karena pendahulu itu akan segera dilepaskan.

Pseudokode 9.4 `FRONT-DELETION`: ingat simpul pertama, pindahkan `head`, kosongkan `prev` milik simpul pertama baru, lepaskan simpul lama.

```
FRONT-DELETION(L):
 1.  if L.head = NIL
 2.      laporkan "list kosong" dan return
 3.  del ← L.head                // ingat simpul pertama
 4.  L.head ← L.head.next        // langkah 1: simpul kedua menjadi yang pertama
 5.  if L.head ≠ NIL
 6.      L.head.prev ← NIL       // langkah 2: simpul pertama baru tidak punya pendahulu
 7.  free del
```

Gambar 9.6 memperlihatkan langkah-langkahnya pada 3, 13, 86. Baris 6 adalah baris yang dilupakan pemrogram yang datang dari *single linked list*. Tanpanya list masih *tercetak* benar ke arah maju, tetapi `prev` milik 13 tetap menyimpan alamat simpul yang dilepaskan baris 7: pointer menggantung di dalam list. Setiap penyusuran mundur berikutnya yang tiba di 13 akan berlanjut ke memori yang sudah dibebaskan.



Gambar 9.6 `frontDeletion()` pada list 3, 13, 86: (a) `del` mengingat simpul pertama, (b) `head` pindah ke 13 dan `prev` milik 13 menjadi NULL, (c) simpul pertama lama dihapus.

Uji di baris 5 juga kasus khusus: ketika list hanya berisi satu simpul, `L.head` menjadi NIL di baris 4 dan tidak ada simpul pertama baru yang perlu diperbarui. **Ongkos:** $O(1)$.

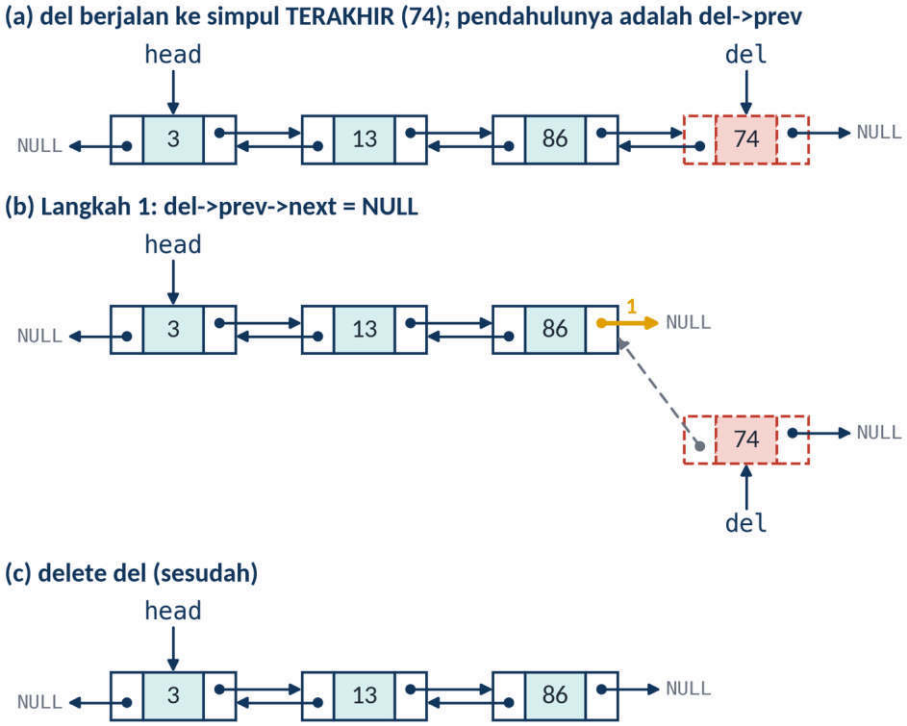
9.3.6 `backDeletion`: membuang simpul terakhir

Di sinilah pointer kedua mulai membayar, bahkan tanpa `tail`. Di Subbab 7.5.6 pembantu harus berhenti pada simpul *kedua dari belakang*, dengan uji yang janggal `aux.next.next ≠ NIL`, karena `next` milik simpul itulah yang harus diubah. Pada *double linked list* pembantu boleh berjalan sampai ke simpul terakhir itu sendiri, karena simpul sebelumnya tinggal `del.prev`.

Pseudokode 9.5 `BACK-DELETION` dengan pointer *head* saja: berjalan ke simpul terakhir; pendahulunya adalah `del.prev`.

```
BACK-DELETION(L):
 1.  if L.head = NIL
 2.      laporkan "list kosong" dan return
 3.  del ← L.head
 4.  while del.next ≠ NIL        // berjalan ke simpul TERAKHIR
 5.      del ← del.next
 6.  if del.prev = NIL           // ia juga yang pertama: satu-satunya simpul
 7.      L.head ← NIL
 8.  else
 9.      del.prev.next ← NIL     // pendahulunya menjadi simpul terakhir
10.  free del
```

Gambar 9.7 memperlihatkan penghapusan 74 dari 3, 13, 86, 74. Bandingkan perulangannya dengan Bab 7. Ini perulangan biasa "berhenti pada simpul terakhir" milik penyisipan di belakang, dan tidak ada lagi bahaya mengevaluasi `aux.next.next` pada list satu simpul: kasus satu simpul dikenali dengan `del.prev = NIL` sesudah perjalanan, bukan sebelumnya.



Gambar 9.7 `backDeletion()` dengan `head` saja pada list 3, 13, 86, 74: (a) `del` berjalan ke simpul terakhir, (b) pendahulunya, `del->prev`, berhenti menunjuknya, (c) simpul itu dihapus.

Ongkos. Tetap $\Theta(n)$, untuk perjalanannya. Pointer `prev` membuat kodenya lebih sederhana, tetapi tidak dapat memperpendek perjalanan selama satu-satunya pintu masuk ada di depan. Itu tugas `tail`, di Subbab 9.4.

9.3.7 `clear`

`clear` adalah penghapusan di depan yang diulang, persis seperti di Subbab 7.5.7. Ia tidak pernah perlu menyentuh medan `prev`, karena setiap simpul yang dikunjunginya akan segera dilepaskan: tidak ada lagi yang akan berjalan mundur. Ia berjalan dalam waktu $\Theta(n)$ dan meninggalkan `head` bernilai null.

9.3.8 Program lengkapnya

Kode 9.2 menyatukan operasi-operasi tadi. Fungsi `main` melakukan urutan yang sama dengan program `sll_head.cpp` dari Subbab 7.5.8 (13 dan 3 di depan, 86 dan 74 di belakang, satu penghapusan di setiap ujung, lalu `clear`) dan mencetak list ke dua arah sesudah setiap kelompok perubahan.

Kode 9.2 `dll_head.cpp`: *double linked list* dengan pointer *head*, beserta setiap operasi dari Subbab 9.3.

```cpp
// dll_head.cpp -- a doubly linked list (non-circular) reached through head
// only.
#include <cstdio>

typedef struct MyNode {
    int data;
    struct MyNode *prev;
    struct MyNode *next;
} TNode;

void init(TNode *&head) {                 // head is passed by reference
    head = nullptr;
}

bool isEmpty(const TNode *head) {
    return head == nullptr;
}

TNode *makeNode(int newData) {
    TNode *newNode = new TNode;
    newNode->data = newData;
    newNode->prev = nullptr;
    newNode->next = nullptr;
    return newNode;
}

void frontInsertion(TNode *&head, int newData) {
    TNode *newNode = makeNode(newData);
    newNode->next = head;  // 1. new node points forward at the old first
    if (!isEmpty(head)) {
        head->prev = newNode;  // 2. old first node points back at the new one
    }
    head = newNode;                       // 3. head moves to the new node
    std::printf("%d has been inserted by frontInsertion()\n", newData);
}
```

Kode 9.2 (Lanjutan) *dll_head.cpp: double linked list dengan pointer head, beserta setiap operasi dari Subbab 9.3.*

```
void backInsertion(TNode *&head, int newData) {
    TNode *newNode = makeNode(newData);
    if (isEmpty(head)) {
        head = newNode;
    } else {
        TNode *aux = head;
        while (aux->next != nullptr) {    // walk to the last node
            aux = aux->next;
        }
        aux->next = newNode;  // 1. old last node points forward at it
        newNode->prev = aux;  // 2. new node points back at the old last
    }
    std::printf("%d has been inserted by backInsertion()\n", newData);
}

void showForward(const TNode *head) {
    if (isEmpty(head)) {
        std::printf("The linked list is empty\n");
        return;
    }
    for (const TNode *aux = head; aux != nullptr; aux = aux->next) {
        std::printf("%d ", aux->data);
    }
    std::printf("\n");
}

void showBackward(const TNode *head) {
    if (isEmpty(head)) {
        std::printf("The linked list is empty\n");
        return;
    }
    const TNode *aux = head;
    while (aux->next != nullptr) {  // with head only, first find the last node
        aux = aux->next;
    }
    // then follow prev back to the front
    for (; aux != nullptr; aux = aux->prev) {
        std::printf("%d ", aux->data);
    }
    std::printf("\n");
}
```

Kode 9.2 (Lanjutan) *dll_head.cpp: double linked list dengan pointer head, beserta setiap operasi dari Subbab 9.3.*

```
void frontDeletion(TNode *&head) {
    if (isEmpty(head)) {
        std::printf("The linked list is empty\n");
        return;
    }
    TNode *del = head;                    // 1. remember the first node
    int d = del->data;
    head = head->next;                    // 2. head moves to the second node
    if (head != nullptr) {
        head->prev = nullptr;  // 3. the new first node has no predecessor
    }
    delete del;                           // 4. release the old first node
    std::printf("%d has been deleted by frontDeletion()\n", d);
}

void backDeletion(TNode *&head) {
    if (isEmpty(head)) {
        std::printf("The linked list is empty\n");
        return;
    }
    TNode *del = head;
    while (del->next != nullptr) {  // walk to the LAST node, not the one before
        del = del->next;
    }
    int d = del->data;
    if (del->prev == nullptr) {           // it was the only node
        head = nullptr;
    } else {
        del->prev->next = nullptr;  // its predecessor becomes the last node
    }
    delete del;
    std::printf("%d has been deleted by backDeletion()\n", d);
}

void clear(TNode *&head) {
    while (head != nullptr) {
        TNode *del = head;
        head = head->next;
        delete del;
    }
    std::printf("The linked list has been cleared\n");
}
```

Kode 9.2 (Lanjutan) *dll_head.cpp: double linked list dengan pointer head, beserta setiap operasi dari Subbab 9.3.*

```
int main() {
    TNode *head;
    init(head);

    std::printf("Front insertion...\n");
    frontInsertion(head, 13);
    frontInsertion(head, 3);

    std::printf("\nBack insertion...\n");
    backInsertion(head, 86);
    backInsertion(head, 74);

    std::printf("\nForward : ");
    showForward(head);
    std::printf("Backward: ");
    showBackward(head);

    std::printf("\nFront deletion...\n");
    frontDeletion(head);
    std::printf("\nBack deletion...\n");
    backDeletion(head);

    std::printf("\nForward : ");
    showForward(head);
    std::printf("Backward: ");
    showBackward(head);

    std::printf("\n");
    clear(head);
    showForward(head);
    return 0;
}
```

```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()

Back insertion...
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()

Forward : 3 13 86 74
Backward: 74 86 13 3

Front deletion...
3 has been deleted by frontDeletion()

Back deletion...
74 has been deleted by backDeletion()

Forward : 13 86
Backward: 86 13

The linked list has been cleared
The linked list is empty
```

Baris-baris maju sama persis dengan baris yang dicetak `sll_head.cpp` di Subbab 7.5.8, dan itu pemeriksaan pertama yang baik: *double linked list* menyimpan barisan yang sama dengan sepupunya yang *single linked* setelah operasi yang sama. Setiap baris mundur adalah baris maju yang dibalik, dan itu memeriksa pointer `prev`: bila ada operasi yang melupakan penugasan mundurnya, baris mundur akan lebih pendek atau memuat nilai yang sudah dihapus. Inilah uji

cepat yang layak disalin ke setiap program yang Anda tulis dengan tautan dua arah: *cetaklah ke dua arah*.

Tabel 9.1 merangkum ongkos yang telah kita turunkan, di samping *single linked list* dari Subbab 7.5.

Tabel 9.1 Waktu jalan dengan pointer `head` saja, untuk list berisi n simpul: pointer kedua mengubah penyusuran mundur tetapi belum mengubah ujung belakang.

Operasi	*Single linked*, `head` saja	*Double linked*, `head` saja	Alasan
`init`, `isEmpty`	$O(1)$	$O(1)$	satu penugasan atau perbandingan
`frontInsertion`	$O(1)$	$O(1)$	tiga penugasan (**Pseudokode 9.1**)
`backInsertion`	$\Theta(n)$	$\Theta(n)$	berjalan ke simpul terakhir (**Pseudokode 9.2**)
penyusuran maju	$\Theta(n)$	$\Theta(n)$	kunjungi setiap simpul
penyusuran mundur	$\Theta(n^2)$, atau $\Theta(n)$ dengan *stack*	$\Theta(n)$, tanpa memori tambahan	berjalan ke ujung, lalu ikuti `prev` (**Pseudokode 9.3**)
`frontDeletion`	$O(1)$	$O(1)$	simpan, pindahkan, kosongkan `prev`, hapus (**Pseudokode 9.4**)
`backDeletion`	$\Theta(n)$	$\Theta(n)$	berjalan ke simpul terakhir (**Pseudokode 9.5**)
`clear`	$\Theta(n)$	$\Theta(n)$	hapus setiap simpul

9.3.9 List yang sama dalam Java dan Python

Kode 9.3 dan **Kode 9.4** adalah versi Java dan Python-nya. Seperti di Subbab 7.5.9, simpul adalah kelas kecil dengan medan bertipe dirinya sendiri, dan list adalah kelas yang medan `head`-nya memainkan peran parameter referensi di C++.

Kode 9.3 `DllHead.java`: *double linked list* dengan referensi *head* dalam Java.

```java
// DllHead.java -- a doubly linked list (non-circular) reached through head
// only.
public class DllHead {
    static class Node {
        int data;
        Node prev, next;

        Node(int data) {
            this.data = data;
        }
    }

    static class DoublyLinkedList {
        private Node head;                    // null means the list is empty

        boolean isEmpty() {
            return head == null;
        }

        void frontInsertion(int newData) {
            Node newNode = new Node(newData);
            newNode.next = head;  // 1. new node points forward at the old first
            // 2. old first points back at it
            if (!isEmpty()) head.prev = newNode;
            head = newNode;                   // 3. head moves to the new node
            System.out.println(newData
                    + " has been inserted by frontInsertion()");
        }

        void backInsertion(int newData) {
            Node newNode = new Node(newData);
            if (isEmpty()) {
                head = newNode;
            } else {
                Node aux = head;
                // walk to the last node
                while (aux.next != null) aux = aux.next;
                aux.next = newNode;  // 1. old last node points forward at it
                newNode.prev = aux;  // 2. new node points back at the old last
            }
            System.out.println(newData
                    + " has been inserted by backInsertion()");
        }
```

Kode 9.3 (Lanjutan) *DllHead.java: double linked list dengan referensi head dalam Java.*

```
    void showForward() {
        if (isEmpty()) {
            System.out.println("The linked list is empty");
            return;
        }
        StringBuilder sb = new StringBuilder();
        for (Node aux = head; aux != null; aux = aux.next)
            sb.append(aux.data).append(' ');
        System.out.println(sb);
    }

    void showBackward() {
        if (isEmpty()) {
            System.out.println("The linked list is empty");
            return;
        }
        Node aux = head;
        // first find the last node
        while (aux.next != null) aux = aux.next;
        StringBuilder sb = new StringBuilder();
        for (; aux != null; aux = aux.prev) sb.append(aux.data).append(' ');
        System.out.println(sb);
    }

    void frontDeletion() {
        if (isEmpty()) {
            System.out.println("The linked list is empty");
            return;
        }
        int d = head.data;
        head = head.next;  // the old first node becomes garbage
        // and nothing may point back at it
        if (head != null) head.prev = null;
        System.out.println(d + " has been deleted by frontDeletion()");
    }
```

Kode 9.3 (Lanjutan) *DllHead.java: double linked list dengan referensi head dalam Java.*

```
        void backDeletion() {
            if (isEmpty()) {
                System.out.println("The linked list is empty");
                return;
            }
            Node del = head;
            while (del.next != null) del = del.next;  // walk to the LAST node
            int d = del.data;
            if (del.prev == null) head = null;  // it was the only node
            else del.prev.next = null;  // its predecessor becomes the last node
            System.out.println(d + " has been deleted by backDeletion()");
        }

        void clear() {
            head = null;
            System.out.println("The linked list has been cleared");
        }
    }

    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();

        System.out.println("Front insertion...");
        list.frontInsertion(13);
        list.frontInsertion(3);

        System.out.println("\nBack insertion...");
        list.backInsertion(86);
        list.backInsertion(74);

        System.out.print("\nForward : ");
        list.showForward();
        System.out.print("Backward: ");
        list.showBackward();

        System.out.println("\nFront deletion...");
        list.frontDeletion();
        System.out.println("\nBack deletion...");
        list.backDeletion();

        System.out.print("\nForward : ");
        list.showForward();
        System.out.print("Backward: ");
        list.showBackward();
```

Kode 9.3 (Lanjutan) *DllHead.java: double linked list dengan referensi head dalam Java.*

```
        System.out.println();
        list.clear();
        list.showForward();
    }
}
```

Kode 9.4 `dll_head.py`: *double linked list* dengan referensi *head* dalam Python. `__slots__` menetapkan ketiga medan sebuah simpul.

```python
# dll_head.py -- a doubly linked list (non-circular) reached through head only.
class Node:
    __slots__ = ("data", "prev", "next")

    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None


class DoublyLinkedList:
    def __init__(self):
        self.head = None                        # None means the list is empty

    def is_empty(self):
        return self.head is None

    def front_insertion(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head  # 1. new node points forward at the old first
        if not self.is_empty():
            self.head.prev = new_node           # 2. old first points back at it
        self.head = new_node                    # 3. head moves to the new node
        print(f"{new_data} has been inserted by frontInsertion()")

    def back_insertion(self, new_data):
        new_node = Node(new_data)
        if self.is_empty():
            self.head = new_node
        else:
            aux = self.head
            while aux.next is not None:         # walk to the last node
                aux = aux.next
            aux.next = new_node  # 1. old last node points forward at it
            new_node.prev = aux  # 2. new node points back at the old last
        print(f"{new_data} has been inserted by backInsertion()")
```

Kode 9.4 (Lanjutan) *dll_head.py: double linked list dengan referensi head dalam Python. __slots__ menetapkan ketiga medan sebuah simpul.*

```
    def show_forward(self):
        if self.is_empty():
            print("The linked list is empty")
            return
        parts, aux = [], self.head
        while aux is not None:
            parts.append(f"{aux.data} ")
            aux = aux.next
        print("".join(parts))

    def show_backward(self):
        if self.is_empty():
            print("The linked list is empty")
            return
        aux = self.head
        while aux.next is not None:            # first find the last node
            aux = aux.next
        parts = []
        while aux is not None:  # then follow prev back to the front
            parts.append(f"{aux.data} ")
            aux = aux.prev
        print("".join(parts))

    def front_deletion(self):
        if self.is_empty():
            print("The linked list is empty")
            return
        d = self.head.data
        self.head = self.head.next  # the old first node becomes garbage
        if self.head is not None:
            self.head.prev = None  # and nothing may point back at it
        print(f"{d} has been deleted by frontDeletion()")

    def back_deletion(self):
        if self.is_empty():
            print("The linked list is empty")
            return
        dele = self.head
        while dele.next is not None:           # walk to the LAST node
            dele = dele.next
        if dele.prev is None:                  # it was the only node
            self.head = None
        else:
            dele.prev.next = None  # its predecessor becomes the last node
        print(f"{dele.data} has been deleted by backDeletion()")
```

Kode 9.4 (Lanjutan) *dll_head.py: double linked list dengan referensi head dalam Python. __slots__ menetapkan ketiga medan sebuah simpul.*

```
    def clear(self):
        self.head = None
        print("The linked list has been cleared")


def main():
    lst = DoublyLinkedList()

    print("Front insertion...")
    lst.front_insertion(13)
    lst.front_insertion(3)

    print("\nBack insertion...")
    lst.back_insertion(86)
    lst.back_insertion(74)

    print("\nForward : ", end="")
    lst.show_forward()
    print("Backward: ", end="")
    lst.show_backward()

    print("\nFront deletion...")
    lst.front_deletion()
    print("\nBack deletion...")
    lst.back_deletion()

    print("\nForward : ", end="")
    lst.show_forward()
    print("Backward: ", end="")
    lst.show_backward()

    print()
    lst.clear()
    lst.show_forward()


if __name__ == "__main__":
    main()
```

Keduanya mencetak keluaran yang sama persis dengan **Kode 9.2**. Perhatikan `frontDeletion` pada kedua versi yang memakai *garbage collector*. Tidak ada `delete`, namun baris `head.prev = null` (Python: `self.head.prev = None`) tetap ada. Baris itu tidak lagi dibutuhkan untuk menghindari pembacaan memori yang sudah dibebaskan, karena referensi tidak pernah dapat menunjuk memori yang sudah dibebaskan, tetapi masih dibutuhkan demi invarian: bila simpul pertama baru tetap menyimpan referensi ke simpul lama, penyusuran mundur akan mengunjungi simpul yang sudah bukan bagian list, dan, yang kurang kasatmata, simpul lama itu tidak pernah dapat dipungut selama list masih hidup, karena list masih merujuknya. Tautan mundur yang basi dalam Java atau Python sekaligus merupakan kesalahan logika dan kebocoran memori.

9.4 Head dan Tail: Setiap Ujung dalam O(1)

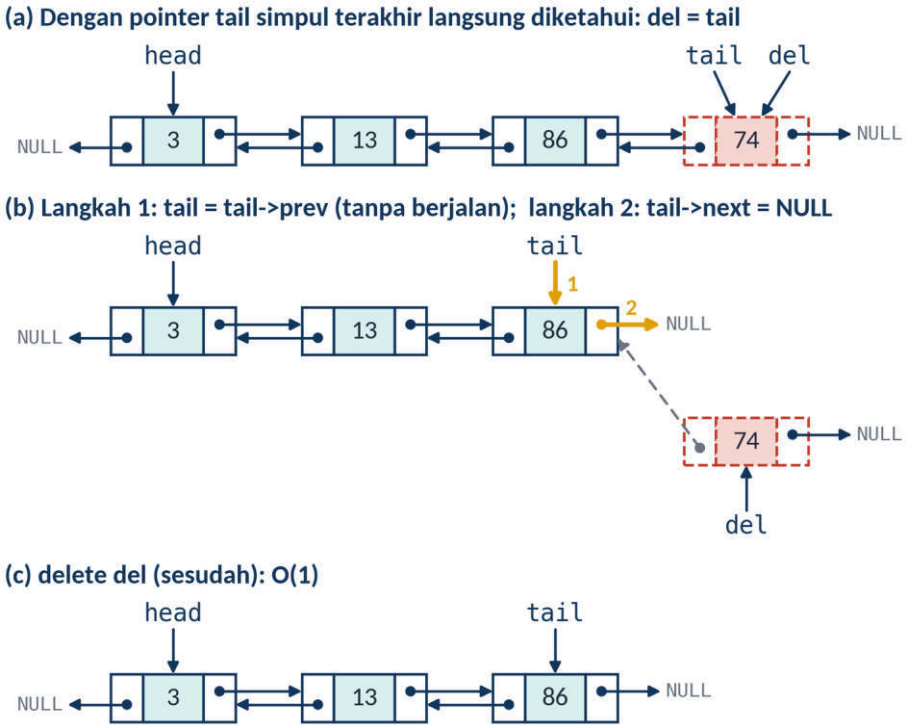
9.4.1 Menambahkan `tail`, dan akhirnya penghapusan di belakang dalam O(1)

Subbab 7.8 menambahkan pointer `tail` ke *single linked list* dan hanya memperoleh separuh dari yang diinginkannya: penyisipan di belakang menjadi $O(1)$, penghapusan di belakang tidak. Pada *double linked list*, `tail` memberikan kedua separuhnya. **Pseudokode 9.6** adalah penghapusan di belakang dengan pointer *tail*.

Pseudokode 9.6 `BACK-DELETION` dengan *head* dan *tail*: simpul sebelum yang terakhir adalah `tail.prev`, sehingga tidak perlu berjalan.

```
BACK-DELETION-WITH-TAIL(L):
 1.  if L.tail = NIL
 2.      laporkan "list kosong" dan return
 3.  del ← L.tail
 4.  L.tail ← L.tail.prev        // langkah 1: tanpa berjalan: simpul terakhir baru adalah
tail.prev
 5.  if L.tail ≠ NIL
 6.      L.tail.next ← NIL       // langkah 2: simpul terakhir baru tidak menunjuk apa pun
 7.  else
 8.      L.head ← NIL            // list tadinya satu simpul dan kini kosong
 9.  free del
```

Baris 4 adalah baris yang tidak dapat ditulis Bab 7. **Gambar 9.8** memperlihatkannya pada 3, 13, 86, 74: `tail` mundur satu langkah menyusuri pointer `prev`, 86 menjadi simpul terakhir, dan 74 dilepaskan, dalam sejumlah tetap langkah berapa pun panjang list.



Gambar 9.8 Penghapusan di belakang dengan pointer *tail* pada 3, 13, 86, 74: (a) `del` = `tail`, (b) `tail` mundur menyusuri `prev` dan `next` milik simpul terakhir baru menjadi NULL, (c) sesudahnya: $O(1)$, tanpa berjalan.

Penghapusan di belakang adalah bayangan cermin penghapusan di depan, baris demi baris, dengan `head` dan `tail` dipertukarkan serta `next` dan `prev` dipertukarkan. Simetri itulah seluruh rancangan *double linked list* dengan *head* dan *tail*: setiap operasi di satu ujung punya kembaran di ujung yang lain, yang diperoleh dengan menukar kedua kata itu. Penyisipan di belakang pun menjadi bayangan cermin penyisipan di depan: `z.prev ← L.tail`; bila list tidak kosong, `L.tail.next ← z`, bila kosong `L.head ← z`; terakhir `L.tail ← z`.

Menambahkan `tail` menambahkan satu klausul pada invarian: `head` dan `tail` sama-sama null atau sama-sama tidak null, dan ketika tidak null, `head.prev` serta `tail.next` bernilai null. Setiap operasi yang dapat membuat list kosong (penghapusan terakhir) atau tidak kosong (penyisipan pertama) kini harus memperbarui *kedua* pointer luar. Dari sanalah kasus-kasus khusus berasal, dan Subbab 9.4.2 menghitungnya.

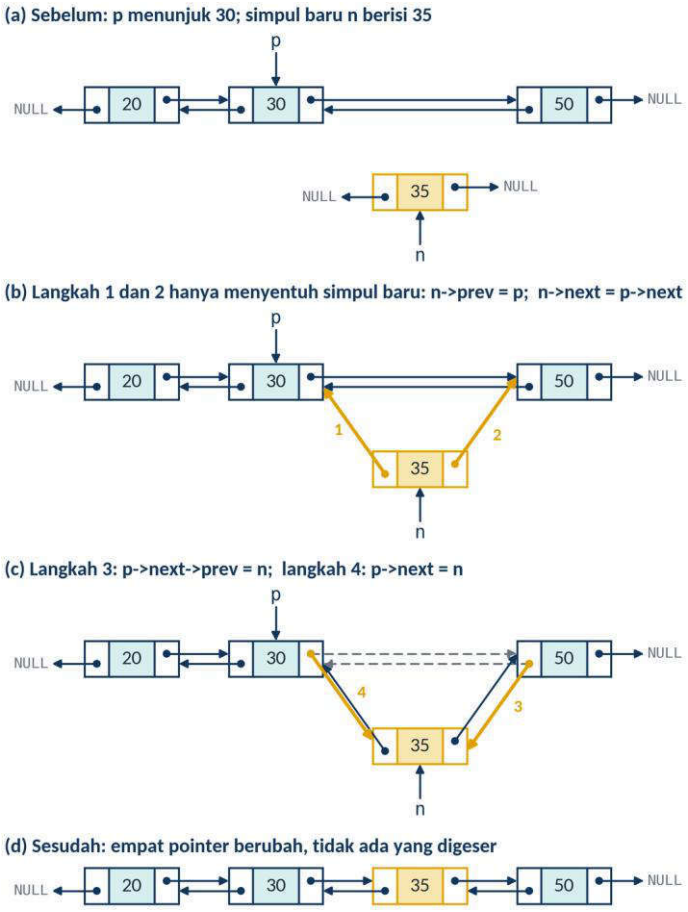
9.4.2 `insertAfter` dan `insertBefore`: menyisipkan di samping simpul yang dipegang

Misalkan program memegang pointer `p` ke sebuah simpul list, mungkin hasil pencarian, dan ingin menyisipkan nilai baru tepat sesudahnya. Subbab 7.6.3 melakukannya untuk *single linked list* dengan dua penugasan. *Double linked list* membutuhkan empat, karena ada empat panah di antara `p`, simpul baru `n`, dan penerus `p` yaitu s: $p \to n$ dan $n \to s$ ke arah maju, $s \to n$ dan $n \to p$ ke arah mundur.

Pseudokode 9.7 `INSERT-AFTER`: menautkan simpul baru sesudah simpul `p` dengan empat penugasan pointer, dalam waktu $O(1)$; langkah 4 harus sesudah langkah 2 dan 3.

```
INSERT-AFTER(L, p, x):
 1.  n ← simpul baru;  n.data ← x
 2.  n.prev ← p                  // langkah 1
 3.  n.next ← p.next             // langkah 2
 4.  if p.next = NIL             // p tadinya simpul terakhir
 5.      L.tail ← n
 6.  else
 7.      p.next.prev ← n         // langkah 3: penerus menunjuk balik ke n
 8.  p.next ← n                  // langkah 4: p menunjuk maju ke n
```

Gambar 9.9 menelusuri `insertAfter(p, 35)` dengan `p` di 30 pada list 20, 30, 50. Langkah 1 dan 2 (panel b) hanya menyentuh simpul baru: keduanya membuat simpul itu menunjuk kedua calon tetangganya, dan belum ada yang berubah di dalam list. Langkah 3 dan 4 (panel c) lalu mengarahkan kedua tetangga kepadanya. Panah abu-abu putus-putus adalah tautan langsung antara 30 dan 50 yang digantikan kedua langkah itu.



Gambar 9.9 `insertAfter(p, 35)` dengan `p` di 30: (a) sebelum, (b) simpul baru menunjuk kedua tetangganya, (c) kedua tetangga menunjuk simpul baru, (d) sesudahnya.

Mengapa urutannya demikian? Baris 3 dan baris 7 sama-sama membaca `p.next` untuk menemukan penerus. Baris 8 menimpa `p.next`. Bila baris 8 dijalankan lebih dulu, `p.next` sudah bernilai `n`, baris 3 membuat `n` menunjuk maju ke dirinya sendiri, baris 7 membuat `n` menunjuk balik ke dirinya sendiri, dan simpul 50, bersama segala yang sesudahnya, tak lagi terjangkau ke arah maju. Subbab 9.11 menggambar kecelakaan itu. Aturan aman yang berlaku untuk setiap penyisipan dalam bab ini adalah: *pasang dulu kedua pointer milik simpul baru, lalu perbarui tetangganya, dan perbarui paling akhir tautan yang Anda pakai untuk menemukan tetangga.*

`insertBefore(p, x)` adalah bayangan cerminnya: `n.next ← p`, `n.prev ← p.prev`, lalu `L.head ← n` (bila p tadinya simpul pertama) atau `p.prev.next ← n`, dan terakhir `p.prev ← n`. Penyisipan *sebelum* sebuah simpul mustahil dalam $O(1)$ pada *single linked list*, yang harus berjalan dari depan untuk menemukan simpul di muka `p` (Soal 7.9 membutuhkan pointer ke tautan untuk itu). Pada *double linked list* pendahulunya hanya selangkah jauhnya.

Ongkos. Empat penugasan dan satu uji: $O(1)$, berapa pun n. Perhatikan makna $O(1)$ di sini: penyisipan itu murah *setelah program memegang* `p`. Menemukan `p` berdasarkan nilai adalah pencarian, yang memakan $\Theta(n)$ pada kasus terburuk. Peringatan yang sama dari Subbab 7.12.4 berlaku: *double linked list* bersinar ketika program sudah memegang posisi yang diubahnya.

Menghitung kasus khusus. `insertAfter` punya satu kasus khusus (baris 4: simpul baru masuk di belakang), dan `insertBefore` punya satu (simpul baru masuk di depan). Penyisipan di depan dan di belakang masing-masing punya satu (list tadinya kosong). Ingat angka-angka ini untuk Subbab 9.7.

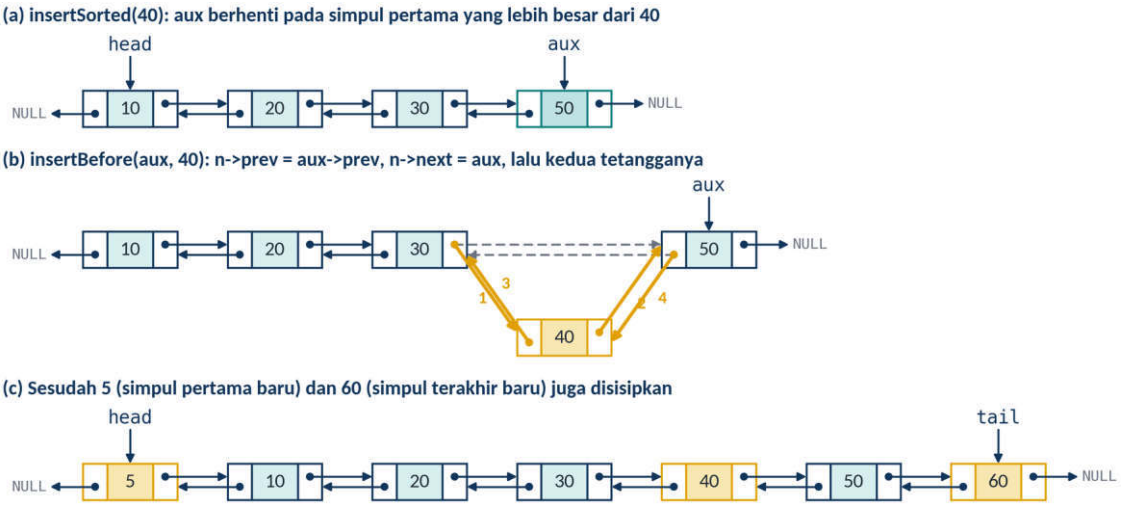
9.4.3 `insertSorted`: menjaga list tetap terurut

Untuk menyisipkan x ke dalam list menaik, berjalanlah sampai simpul pertama yang datanya tidak lebih kecil dari x, lalu sisipkan sebelum simpul itu; bila tidak ada simpul semacam itu, x masuk di belakang.

Pseudokode 9.8 `INSERT-SORTED`: temukan simpul pertama yang tidak lebih kecil dari `x`, lalu sisipkan sebelumnya.

```
INSERT-SORTED(L, x):
 1.  aux ← L.head
 2.  while aux ≠ NIL and aux.data < x
 3.      aux ← aux.next
 4.  if aux = NIL
 5.      BACK-INSERTION(L, x)    // x lebih besar dari setiap nilai (atau L kosong)
 6.  else
 7.      INSERT-BEFORE(L, aux, x)
```

Bandingkan dengan Subbab 7.6.4. Di sana pembantu harus berhenti pada simpul terakhir yang *lebih kecil* dari x, satu simpul lebih awal, karena *single linked list* hanya dapat menyisipkan sesudah sebuah simpul, dan ujinya harus melihat satu simpul ke depan. Di sini pembantu berhenti di tempat ia wajar berhenti, yaitu simpul pertama yang lebih besar, dan penyisipan terjadi sebelumnya. **Gambar 9.10** memperlihatkan `insertSorted(40)` pada 10, 20, 30, 50, dan list sesudah 5 (simpul pertama baru) dan 60 (simpul terakhir baru) juga disisipkan.



Gambar 9.10 `insertSorted`: (a) untuk 40, `aux` berhenti di 50, nilai pertama yang lebih besar, (b) simpul baru ditautkan sebelum `aux` dalam empat langkah, (c) list sesudah 5 dan 60 juga disisipkan, di depan dan di belakang.

Ongkos. Perjalanannya paling banyak n langkah dan penyisipannya $O(1)$, sehingga `insertSorted` bernilai $\Theta(n)$ pada kasus terburuk, sama seperti pada *single linked list*. Versi *double linked* tidak lebih cepat, tetapi lebih sederhana.

9.4.4 `search`, `deleteNode`, dan `deleteKey`

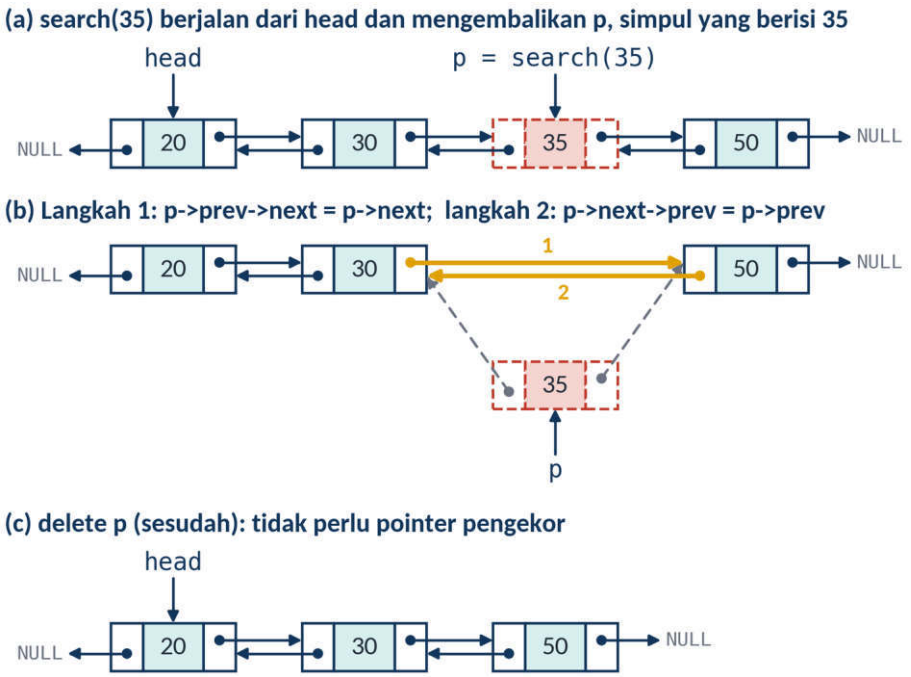
`search(L, key)` adalah pencarian sekuensial dari Subbab 7.6.1, tanpa perubahan: berjalan dari `head` dan kembalikan simpul pertama yang berisi `key`, atau NIL. Bila `tail` tersedia, pencarian dapat pula dimulai dari belakang, yang menemukan kemunculan *terakhir* lebih dulu.

Penghapusan adalah tempat *double linked list* paling jelas mengungguli *single linked list*. Diberikan pointer `p` ke simpul mana pun, simpul sebelumnya adalah `p.prev` dan simpul sesudahnya adalah `p.next`; dua penugasan membuat keduanya melompati `p`.

Pseudokode 9.9 `DELETE-NODE`: melepas simpul `p` dalam waktu $O(1)$; tidak perlu pencarian maupun pointer pengekor.

```
DELETE-NODE(L, p):
 1.  if p.prev = NIL              // p adalah simpul pertama
 2.      L.head ← p.next
 3.  else
 4.      p.prev.next ← p.next     // langkah 1: pendahulu melompati p ke arah maju
 5.  if p.next = NIL              // p adalah simpul terakhir
 6.      L.tail ← p.prev
 7.  else
 8.      p.next.prev ← p.prev     // langkah 2: penerus melompati p ke arah mundur
 9.  free p
```

Gambar 9.11 memperlihatkan `deleteKey(35)`: pencarian menemukan `p` (panel a), kedua tetangga ditautkan satu sama lain (panel b), dan simpul itu dilepaskan (panel c). Perhatikan bahwa pointer milik `p` sendiri *dibaca* tetapi tidak pernah diubah; simpul itu cukup ditinggalkan oleh tetangganya sebelum dibebaskan. `deleteKey(L, key)` tidak lain adalah `p ← SEARCH(L, key)` diikuti `DELETE-NODE(L, p)` bila `p` bukan NIL. `frontDeletion` adalah `DELETE-NODE(L, L.head)`, dan `backDeletion` adalah `DELETE-NODE(L, L.tail)`: satu prosedur ini, dengan dua uji kasus khususnya, mencakup setiap penghapusan.



Gambar 9.11 `deleteKey(35)`: (a) `search` mengembalikan `p`, (b) pendahulu dan penerus ditautkan satu sama lain, (c) `p` dihapus.

Bandingkan dengan prosedur `DELETE-KEY` dari Subbab 7.6.5, yang pointer pengekor `prev`-nya berjalan satu langkah di belakang `cur` selama pencarian, dan simpul hanya dapat dibuang karena pencarian menyimpan pendahulunya. Itu baik bila penghapusan menyusul pencarian, tetapi tak berguna bila pointer ke simpul datang dari tempat lain: dari *hash map* (Subbab 9.9), dari iterator yang dipegang bagian lain program, atau dari simpul itu sendiri. *Single linked list* lalu harus mencari pendahulunya, $\Theta(n)$; *double linked list* tidak, $O(1)$. Subbab 9.10 mengukur perbedaannya.

Ongkos. `DELETE-NODE` bernilai $O(1)$. `deleteKey` bernilai $\Theta(n)$ pada kasus terburuk, untuk pencariannya.

9.4.5 Memeriksa tautan

Invarian Persamaan (9.1) murah untuk diperiksa dalam program uji. `checkLinks` di **Kode 9.5** menyusuri list maju sekali dan memverifikasi bahwa setiap `next` punya `prev` pasangannya, bahwa `head.prev` dan `tail.next` bernilai null, dan bahwa `head` dan `tail` sama-sama null atau sama-sama tidak null. Fungsi ini berjalan dalam waktu $\Theta(n)$, sehingga tidak layak diletakkan di dalam operasi-operasi itu sendiri, tetapi memanggilnya sesudah setiap operasi dalam sebuah uji adalah cara tercepat untuk menangkap operasi yang melupakan salah satu dari keempat penugasannya. Penugasan mundur yang terlupa menghasilkan list yang tercetak sempurna ke arah maju, galat klasik yang tak kasatmata dalam bab ini; `checkLinks` langsung membuatnya terlihat.

9.4.6 Program lengkapnya

Kode 9.5 mengimplementasikan list dengan `head` dan `tail` serta setiap operasi subbab ini. Tipe list-nya adalah `struct DList` kecil yang memegang kedua pointer, dilewatkan secara referensi.

Kode 9.5 `dll_ops.cpp`: *double linked list* dengan *head* dan *tail*: penyisipan dan penghapusan di depan dan di belakang, penyisipan sesudah dan sebelum simpul, penyisipan terurut, pencarian, penghapusan berdasarkan pointer dan kunci, kedua penyusuran, serta pemeriksaan tautan.

```cpp
// dll_ops.cpp -- a doubly linked list with head AND tail: every end operation
// is O(1), and so is inserting or deleting next to a node we already hold.
#include <cstdio>

struct Node {
    int data;
    Node *prev;
    Node *next;
};

struct DList {
    Node *head = nullptr;                // first node, or nullptr when empty
    Node *tail = nullptr;                // last node, or nullptr when empty
};

bool isEmpty(const DList &L) { return L.head == nullptr; }

void frontInsertion(DList &L, int x) {
    Node *n = new Node{x, nullptr, L.head};
    if (isEmpty(L)) L.tail = n;          // the only node is also the last
    else L.head->prev = n;
    L.head = n;
}

void backInsertion(DList &L, int x) {
    Node *n = new Node{x, L.tail, nullptr};
    if (isEmpty(L)) L.head = n;          // the only node is also the first
    else L.tail->next = n;
    L.tail = n;
}

void insertAfter(DList &L, Node *p, int x) {      // p must be a node of L
    // 1. n->prev = p   2. n->next = p's successor
    Node *n = new Node{x, p, p->next};
    if (p->next == nullptr) L.tail = n;   // p was the last node
    else p->next->prev = n;               // 3. the successor points back at n
    p->next = n;                          // 4. p points forward at n
}
```

Kode 9.5 (Lanjutan) *dll_ops.cpp: double linked list dengan head dan tail: penyisipan dan penghapusan di depan dan di belakang, penyisipan sesudah dan sebelum simpul, penyisipan terurut, pencarian, penghapusan berdasarkan pointer dan kunci, kedua penyusuran, serta pemeriksaan tautan.*

```
// the mirror image of insertAfter
void insertBefore(DList &L, Node *p, int x) {
    Node *n = new Node{x, p->prev, p};
    if (p->prev == nullptr) L.head = n;   // p was the first node
    else p->prev->next = n;
    p->prev = n;
}

void insertSorted(DList &L, int x) {      // keeps an ascending list ascending
    Node *aux = L.head;
    while (aux != nullptr && aux->data < x) aux = aux->next;
    if (aux == nullptr) backInsertion(L, x);          // larger than everything
    else insertBefore(L, aux, x);  // before the first larger node
}

Node *search(const DList &L, int key) {
    Node *aux = L.head;
    while (aux != nullptr && aux->data != key) aux = aux->next;
    return aux;                           // the node holding key, or nullptr
}

// unlink p in O(1): no walk, no trailing ptr
void deleteNode(DList &L, Node *p) {
    if (p->prev == nullptr) L.head = p->next;         // p was the first node
    else p->prev->next = p->next;                     // 1. predecessor skips p
    if (p->next == nullptr) L.tail = p->prev;         // p was the last node
    else p->next->prev = p->prev;  // 2. successor skips p back
    delete p;
}

void frontDeletion(DList &L) { if (!isEmpty(L)) deleteNode(L, L.head); }
void backDeletion(DList &L) { if (!isEmpty(L)) deleteNode(L, L.tail); }

bool deleteKey(DList &L, int key) {
    Node *p = search(L, key);
    if (p == nullptr) return false;
    deleteNode(L, p);
    return true;
}
```

Kode 9.5 (Lanjutan) *dll_ops.cpp: double linked list dengan head dan tail: penyisipan dan penghapusan di depan dan di belakang, penyisipan sesudah dan sebelum simpul, penyisipan terurut, pencarian, penghapusan berdasarkan pointer dan kunci, kedua penyusuran, serta pemeriksaan tautan.*

```
void showForward(const char *label, const DList &L) {
    std::printf("%-26s", label);
    for (Node *aux = L.head; aux != nullptr; aux = aux->next)
        std::printf("%d ", aux->data);
    std::printf("\n");
}

void showBackward(const char *label, const DList &L) {
    std::printf("%-26s", label);
    for (Node *aux = L.tail; aux != nullptr; aux = aux->prev)
        std::printf("%d ", aux->data);
    std::printf("\n");
}

// Checks the two-way invariant: every next has a matching prev, both ends are
// NULL.
bool checkLinks(const DList &L) {
    if (L.head == nullptr || L.tail == nullptr) return L.head == L.tail;
    if (L.head->prev != nullptr || L.tail->next != nullptr) return false;
    for (Node *p = L.head; p->next != nullptr; p = p->next) {
        if (p->next->prev != p) return false;
    }
    return true;
}

void clear(DList &L) {
    while (L.head != nullptr) {
        Node *del = L.head;
        L.head = L.head->next;
        delete del;
    }
    L.tail = nullptr;
}

int main() {
    DList L;
    backInsertion(L, 20);
    backInsertion(L, 30);
    backInsertion(L, 50);
    frontInsertion(L, 10);
    showForward("start:", L);
```

Kode 9.5 (Lanjutan) *dll_ops.cpp: double linked list dengan head dan tail: penyisipan dan penghapusan di depan dan di belakang, penyisipan sesudah dan sebelum simpul, penyisipan terurut, pencarian, penghapusan berdasarkan pointer dan kunci, kedua penyusuran, serta pemeriksaan tautan.*

```
    insertAfter(L, search(L, 30), 35);
    showForward("insertAfter(30, 35):", L);
    insertBefore(L, search(L, 10), 5);
    showForward("insertBefore(10, 5):", L);
    insertSorted(L, 40);
    insertSorted(L, 1);
    insertSorted(L, 60);
    showForward("insertSorted 40, 1, 60:", L);
    showBackward("backward:", L);
    std::printf("%-26s%s\n", "links consistent:", checkLinks(L) ? "yes" : "no");

    Node *p = search(L, 35);
    std::printf("%-26s%d <- %d -> %d\n", "search(35):",
                p->prev->data, p->data, p->next->data);
    std::printf("%-26s%s\n", "search(99):",
                search(L, 99) == nullptr ? "not found" : "found");

    deleteKey(L, 35);
    showForward("deleteKey(35):", L);
    frontDeletion(L);
    backDeletion(L);
    showForward("front and back deletion:", L);
    showBackward("backward:", L);
    std::printf("%-26s%s\n", "deleteKey(99):",
                deleteKey(L, 99) ? "deleted" : "not found");
    std::printf("%-26s%s\n", "links consistent:", checkLinks(L) ? "yes" : "no");

    clear(L);
    std::printf("%-26s%s\n", "after clear, empty:", isEmpty(L) ? "yes" : "no");
    return 0;
}
```

```
start:                     10 20 30 50
insertAfter(30, 35):       10 20 30 35 50
insertBefore(10, 5):       5 10 20 30 35 50
insertSorted 40, 1, 60:    1 5 10 20 30 35 40 50 60
backward:                  60 50 40 35 30 20 10 5 1
links consistent:          yes
search(35):                30 <- 35 -> 40
search(99):                not found
deleteKey(35):             1 5 10 20 30 40 50 60
front and back deletion:   5 10 20 30 40 50
backward:                  50 40 30 20 10 5
deleteKey(99):             not found
links consistent:          yes
after clear, empty:        yes
```

Ikuti jalannya. List mula-mula 10, 20, 30, 50. `insertAfter(30, 35)` menaruh 35 di antara 30 dan 50; `insertBefore(10, 5)` menaruh 5 di muka simpul pertama, sehingga `head` berubah. Ketiga penyisipan terurut memperlihatkan ketiga kasus **Pseudokode 9.8**: 40 masuk di tengah, 1 di depan, dan 60 di belakang. Baris mundur adalah baris maju yang dibalik, dan `checkLinks` menyetujuinya. `search(35)` mengembalikan simpul yang tetangganya, dibaca lewat `prev` dan `next` miliknya sendiri, adalah 30 dan 40. Penghapusan kemudian membuang 35 (di tengah), 1 (di depan), dan 60 (di belakang), masing-masing dalam $O(1)$ setelah simpulnya diketahui.

Tabel 9.2 mengumpulkan ongkos list dengan *head* dan *tail*.

Tabel 9.2 Ongkos dengan pointer *head* dan *tail*, untuk list berisi n simpul. *Double linked list* mengubah setiap operasi pada posisi yang diketahui menjadi $O(1)$, dengan harga pekerjaan pointer dua kali lipat.

Operasi	*Single linked*, *head* dan *tail* (Bab 7)	*Double linked*, *head* dan *tail*
penyisipan di depan, penghapusan di depan	$O(1)$	$O(1)$
penyisipan di belakang	$O(1)$	$O(1)$
penghapusan di belakang	$\Theta(n)$	$O(1)$ (**Pseudokode 9.6**)
sisipkan sesudah simpul yang dipegang	$O(1)$	$O(1)$ (**Pseudokode 9.7**)
sisipkan sebelum simpul yang dipegang	$\Theta(n)$	$O(1)$
hapus simpul yang dipegang	$\Theta(n)$	$O(1)$ (**Pseudokode 9.9**)
pencarian, hapus berdasarkan kunci, penyisipan terurut	$\Theta(n)$	$\Theta(n)$
penyusuran maju	$\Theta(n)$	$\Theta(n)$
penyusuran mundur	$\Theta(n^2)$, atau $\Theta(n)$ dengan *stack*	$\Theta(n)$
penugasan pointer per penyisipan / penghapusan	2 / 1	4 / 2

9.5 Double Linked List dengan Simpul Kepala

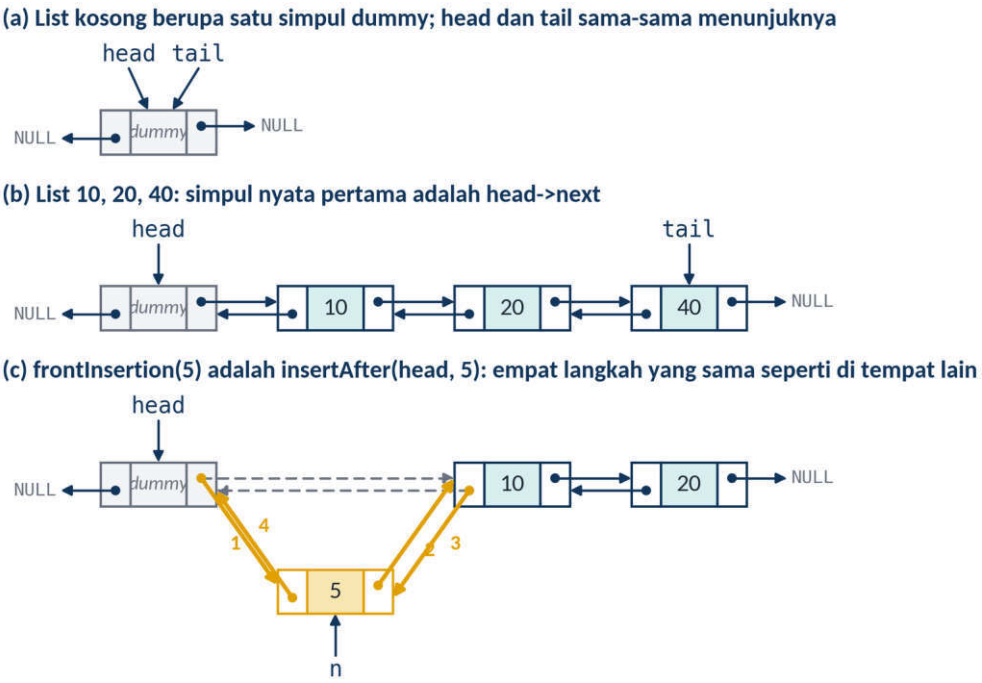
9.5.1 Dari mana sisa kasus khusus berasal

Hitunglah uji kasus khusus dalam list Subbab 9.4. `frontInsertion` dan `backInsertion` menguji list kosong. `insertAfter` menguji apakah `p` tadinya simpul terakhir, `insertBefore` apakah ia simpul pertama. `deleteNode` menguji kedua-duanya: apakah `p` simpul pertama dan apakah ia simpul terakhir. Setiap uji itu punya penyebab yang sama: simpul di *ujung* list punya pointer null di satu sisi, dan ketika sebuah operasi hendak menulis lewat pointer itu, ia harus menulis ke `head` atau `tail` sebagai gantinya.

Subbab 7.7 menjumpai masalah yang sama pada *single linked list* dan menghapus uji-uji di depan dengan *simpul kepala dummy*, simpul yang selalu berada paling depan dan tidak menyimpan data. Gagasan yang sama berlaku di sini, dan ia menghapus tepat separuh uji.

> **DEFINISI.** *Double linked list dengan simpul kepala* menyimpan satu simpul tambahan, *simpul kepala* (atau simpul *dummy*), secara permanen di depan. Datanya tidak pernah dipakai. `head` selalu menunjuknya, simpul nyata pertama adalah `head->next`, dan list kosong hanya terdiri atas simpul kepala. List *tanpa simpul kepala* dari Subbab 9.3 dan 9.4 adalah list yang `head`-nya langsung menunjuk simpul nyata pertama.

Gambar 9.12 memperlihatkan list kosong (panel a), list 10, 20, 40 (panel b), dan penyisipan di depan (panel c). Karena kini setiap simpul nyata punya pendahulu (pendahulu simpul nyata pertama adalah simpul kepala), tidak ada operasi yang perlu menulis ke `head` lagi. Penyisipan di depan menjadi `insertAfter(head, x)`, dan ia menjalankan empat langkah yang sama dengan penyisipan di tempat lain. Dengan cara yang sama `deleteNode(p)` selalu dapat menulis `p->prev->next = p->next` tanpa menguji, karena `p->prev` tidak pernah null.



Gambar 9.12 *Double linked list* dengan simpul kepala: (a) list kosong berupa satu simpul *dummy*, dan `tail` juga menunjuknya, (b) list 10, 20, 40, (c) penyisipan di depan adalah `insertAfter(head, 5)`, dengan empat langkah yang sama seperti penyisipan mana pun.

Ujung *belakang* masih punya pointer null, sehingga satu uji tersisa di setiap operasi: `insertAfter` harus memeriksa apakah `p->next` null (maka simpul baru menjadi simpul terakhir dan `tail` pindah), dan `deleteNode` harus memeriksa apakah `p->next` null (maka `tail` mundur). Satu detail yang praktis membuat penyisipan di belakang pun bebas dari uji list kosong: pada list kosong `tail` menunjuk simpul kepala, sehingga `backInsertion(x)` cukup `insertAfter(tail, x)`.

9.5.2 Programnya

Kode 9.6 mengimplementasikan list dengan simpul kepala. Tipe simpul dan tipe list-nya sama dengan Subbab 9.4, tetapi `init` membuat simpul kepala dan `destroy` melepaskannya bersama simpul-simpul nyata.

Kode 9.6 `dll_headnode.cpp`: *double linked list* dengan simpul kepala; setiap operasi tinggal punya satu kasus khusus, di ujung belakang.

```cpp
// dll_headnode.cpp -- a doubly linked list WITH a head node: a dummy node that
// is always first. Only the back end (a NULL next) still needs a special case.
#include <cstdio>

struct Node {
    int data;
    Node *prev;
    Node *next;
};

struct DList {
    Node *head;                          // always the dummy head node
    Node *tail;  // last node; the dummy itself when empty
};

void init(DList &L) {
    L.head = new Node{0, nullptr, nullptr};   // the dummy's data is never used
    L.tail = L.head;
}

bool isEmpty(const DList &L) { return L.head->next == nullptr; }

void insertAfter(DList &L, Node *p, int x) {  // p may be the dummy
    Node *n = new Node{x, p, p->next};
    if (p->next != nullptr) p->next->prev = n;
    else L.tail = n;  // the one special case: inserting at the back
    p->next = n;
}

void frontInsertion(DList &L, int x) { insertAfter(L, L.head, x); }
void backInsertion(DList &L, int x) { insertAfter(L, L.tail, x); }

void insertSorted(DList &L, int x) {
    Node *p = L.head;  // stop on the last node smaller than x
    while (p->next != nullptr && p->next->data < x) p = p->next;
    insertAfter(L, p, x);
}

void deleteNode(DList &L, Node *p) {      // p is a real node, never the dummy
    p->prev->next = p->next;  // a predecessor always exists: no test
    if (p->next != nullptr) p->next->prev = p->prev;
    else L.tail = p->prev;  // the one special case: deleting the last
    delete p;
}
```

Kode 9.6 (Lanjutan) *dll_headnode.cpp: double linked list dengan simpul kepala; setiap operasi tinggal punya satu kasus khusus, di ujung belakang.*

```
Node *search(const DList &L, int key) {
    Node *p = L.head->next;
    while (p != nullptr && p->data != key) p = p->next;
    return p;
}

void frontDeletion(DList &L) { if (!isEmpty(L)) deleteNode(L, L.head->next); }
void backDeletion(DList &L) { if (!isEmpty(L)) deleteNode(L, L.tail); }

void show(const char *label, const DList &L) {
    std::printf("%-22s", label);
    for (Node *p = L.head->next; p != nullptr; p = p->next)
        std::printf("%d ", p->data);
    std::printf("| back: ");
    for (Node *p = L.tail; p != L.head; p = p->prev)
        std::printf("%d ", p->data);
    std::printf("\n");
}

void destroy(DList &L) {  // releases the real nodes AND the dummy
    while (L.head != nullptr) {
        Node *del = L.head;
        L.head = L.head->next;
        delete del;
    }
    L.tail = nullptr;
}

int main() {
    DList L;
    init(L);
    show("empty:", L);
    frontInsertion(L, 20);
    frontInsertion(L, 10);
    backInsertion(L, 40);
    show("10, 20, 40:", L);
    insertSorted(L, 30);
    insertSorted(L, 5);
    insertSorted(L, 50);
    show("sorted 30, 5, 50:", L);
    deleteNode(L, search(L, 30));
    show("delete 30:", L);
    frontDeletion(L);
    backDeletion(L);
    show("front + back del:", L);
    while (!isEmpty(L)) frontDeletion(L);
```

Kode 9.6 (Lanjutan) *dll_headnode.cpp: double linked list dengan simpul kepala; setiap operasi tinggal punya satu kasus khusus, di ujung belakang.*

```
    show("delete the rest:", L);
    std::printf("%-22s%s\n", "empty again:", isEmpty(L) ? "yes" : "no");
    destroy(L);
    return 0;
}
```

```
empty:                | back:
10, 20, 40:           10 20 40 | back: 40 20 10
sorted 30, 5, 50:     5 10 20 30 40 50 | back: 50 40 30 20 10 5
delete 30:            5 10 20 40 50 | back: 50 40 20 10 5
front + back del:     10 20 40 | back: 40 20 10
delete the rest:      | back:
empty again:          yes
```

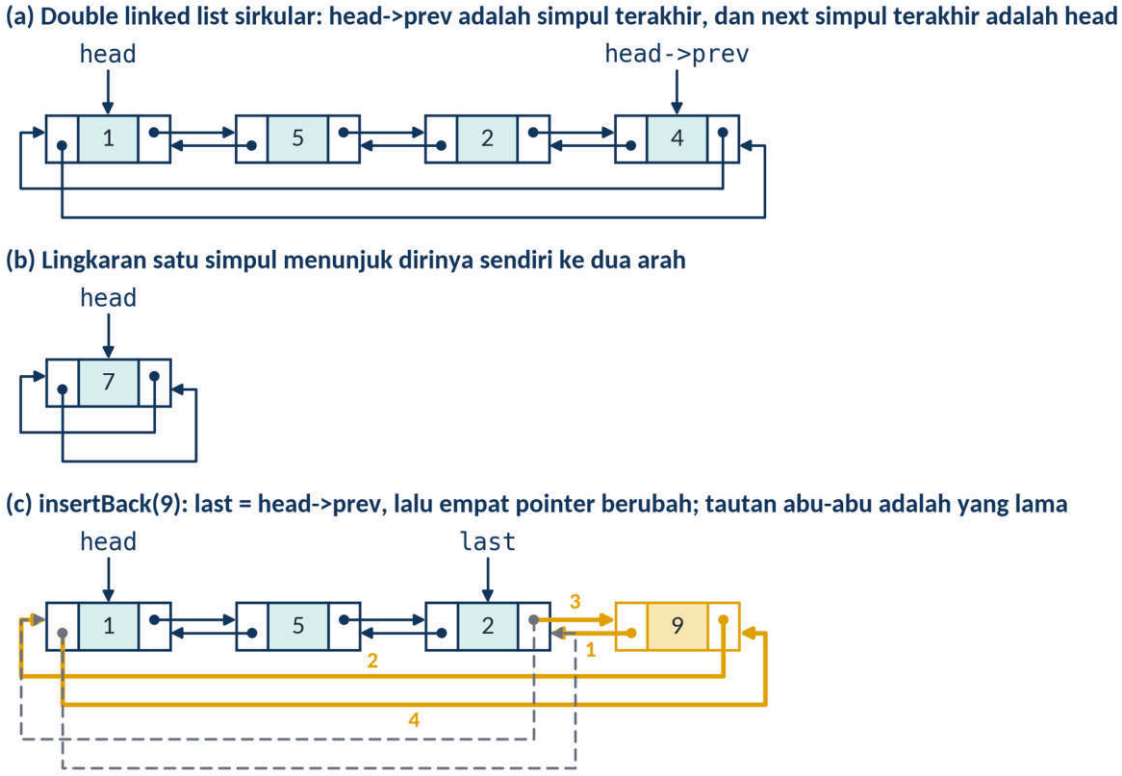
Setiap baris keluaran memperlihatkan list ke arah maju lalu ke arah mundur, dan penyusuran mundur berhenti ketika tiba di simpul kepala (`p != L.head`), bukan di null: simpul kepala berperan sebagai tanda berhenti di depan. Setelah semua simpul nyata dihapus, `isEmpty` melaporkan list kosong, dan `destroy` masih punya satu simpul untuk dilepaskan, yaitu simpul kepala itu sendiri. Melupakan `delete` terakhir itu adalah kebocoran yang nyata dan sering terjadi; Subbab 9.11 menangkap kebocoran yang sama pada sebuah *sentinel*.

9.6 Double Linked List Sirkular

9.6.1 Menutup kedua rantai

Di Subbab 7.9 simpul terakhir *single linked list* dibuat menunjuk kembali ke simpul pertama, dan list berubah menjadi cincin. *Double linked list* punya dua rantai, dan *double linked list sirkular* (*DLLC*) menutup keduanya: `next` milik simpul terakhir menunjuk simpul pertama, dan `prev` milik simpul pertama menunjuk simpul terakhir. Tidak ada pointer yang null pada list sirkular yang tidak kosong.

Gambar 9.13 memperlihatkan hasilnya. Panel (a) adalah list sirkular 1, 5, 2, 4 yang dicapai lewat `head`. Sesuatu yang menakjubkan terjadi: list tidak punya pointer `tail`, namun simpul terakhir hanya selangkah jauhnya, karena ia adalah `head->prev`. Satu pointer luar kini memberi akses $O(1)$ ke kedua ujung. Panel (b) adalah lingkaran satu simpul, yang menunjuk dirinya sendiri ke dua arah. Panel (c) menyisipkan 9 di belakang: simpul terakhir ditemukan sebagai `head->prev`, dan empat pointer berubah, seperti pada setiap penyisipan dalam bab ini.



Gambar 9.13 *Double linked list* sirkular yang dicapai lewat `head`: (a) empat simpul; simpul terakhir adalah `head->prev`, (b) lingkaran satu simpul, (c) `insertBack(9)` menautkan simpul baru di antara simpul terakhir dan `head`.

Amati panel (c) dengan saksama. Menyisipkan di belakang lingkaran berarti menyisipkan *di antara* simpul terakhir dan simpul pertama, dan menyisipkan di depan berarti menyisipkan di

tempat yang persis sama; satu-satunya perbedaan adalah apakah `head` pindah ke simpul baru sesudahnya. Itulah sebabnya program di Subbab 9.6.3 mengimplementasikan `insertFront` sebagai `head = insertBack(head, t)`.

9.6.2 Penyusuran butuh tanda berhenti yang baru

Seperti di Subbab 7.9.2, perulangan yang menunggu null tidak pernah selesai, karena tidak ada null. Penyusuran harus berhenti ketika kembali ke tempat ia mulai, dan karena ia juga harus mengunjungi simpul awal, perulangan yang wajar adalah `do-while`:

```
const Node *p = head;
do {
    visit(p);
    p = p->next;          // backward: start at head->prev and follow prev
} while (p != head);
```

List kosong harus diuji sebelum perulangan, karena `do-while` menjalankan badannya paling sedikit sekali. Menghapus dari lingkaran tanpa *sentinel* punya dua kasus khusus: satu-satunya simpul (maka `head` menjadi null) dan simpul pertama (maka `head` harus pindah ke simpul berikutnya).

9.6.3 Korsel di lobi

Lobi perpustakaan Pustaka Ceria punya layar yang menampilkan satu buku unggulan pada satu waktu. Pengunjung menekan **Next** dan **Previous**, dan layarnya berputar: sesudah buku unggulan terakhir muncul lagi yang pertama, dan sebelum yang pertama muncul yang terakhir. Itulah persis *double linked list* sirkular dengan kursor. **Kode 9.7** membangun korsel berisi empat judul, menekan Next lima kali dan Previous tiga kali, lalu membuang dua judul.

Kode 9.7 `cdll.cpp`: *double linked list* sirkular yang dicapai lewat `head`, dipakai sebagai korsel lobi; `next` dan `previous` berputar di kedua ujung.

```cpp
// cdll.cpp -- a circular doubly linked list reached through head (no sentinel):
// the featured-books carousel in the Pustaka Ceria lobby.
#include <cstdio>
#include <string>

struct Node {
    std::string title;
    Node *prev;
    Node *next;
};

// Inserts a node just before head, i.e. at the back of the circle. O(1).
Node *insertBack(Node *&head, const std::string &t) {
    Node *n = new Node{t, nullptr, nullptr};
    if (head == nullptr) {  // a circle of one points at itself, both ways
        n->prev = n;
        n->next = n;
        head = n;
    } else {
        Node *last = head->prev;  // the last node is one step BACK from head
        n->prev = last;                    // 1.
        n->next = head;                    // 2.
        last->next = n;                    // 3.
        head->prev = n;                    // 4.
    }
    return n;
}

void insertFront(Node *&head, const std::string &t) {
    // same place in the circle; it just becomes head
    head = insertBack(head, t);
}

void deleteNode(Node *&head, Node *p) {   // p must be a node of the circle
    if (p->next == p) {                   // special case 1: the only node
        head = nullptr;
    } else {
        p->prev->next = p->next;
        p->next->prev = p->prev;
        // special case 2: deleting the first node
        if (p == head) head = p->next;
    }
    delete p;
}
```

Kode 9.7 (Lanjutan) *cdll.cpp: double linked list sirkular yang dicapai lewat head, dipakai sebagai korsel lobi; next dan previous berputar di kedua ujung.*

```
void showForward(const Node *head) {
    if (head == nullptr) { std::printf("(empty)\n"); return; }
    const Node *p = head;
    do {  // no NULL anywhere: stop on returning to head
        std::printf("[%s] ", p->title.c_str());
        p = p->next;
    } while (p != head);
    std::printf("\n");
}

void showBackward(const Node *head) {
    if (head == nullptr) { std::printf("(empty)\n"); return; }
    const Node *p = head->prev;           // start at the last node
    do {
        std::printf("[%s] ", p->title.c_str());
        p = p->prev;
    } while (p != head->prev);
    std::printf("\n");
}

void destroy(Node *&head) {
    if (head == nullptr) return;
    head->prev->next = nullptr;  // cut the circle open, then free as a line
    while (head != nullptr) {
        Node *del = head;
        head = head->next;
        delete del;
    }
}

int main() {
    Node *head = nullptr;
    insertBack(head, "Laskar Pelangi");
    insertBack(head, "Bumi Manusia");
    insertBack(head, "Filosofi Kopi");
    insertFront(head, "Ronggeng Dukuh Paruk");
    std::printf("forward : "); showForward(head);
    std::printf("backward: "); showBackward(head);
```

Kode 9.7 (Lanjutan) *cdll.cpp: double linked list sirkular yang dicapai lewat head, dipakai sebagai korsel lobi; next dan previous berputar di kedua ujung.*

```
    std::printf("\nThe lobby screen, pressing Next 5 times, "
                "then Previous 3 times:\n");
    Node *screen = head;
    for (int i = 0; i < 5; ++i) {
        screen = screen->next;
        std::printf("  next     -> %s\n", screen->title.c_str());
    }
    for (int i = 0; i < 3; ++i) {
        screen = screen->prev;
        std::printf("  previous -> %s\n", screen->title.c_str());
    }

    std::printf("\nremove the book on screen (%s):\n", screen->title.c_str());
    deleteNode(head, screen);
    std::printf("forward : "); showForward(head);
    std::printf("remove the first book (%s):\n", head->title.c_str());
    deleteNode(head, head);
    std::printf("forward : "); showForward(head);
    std::printf("backward: "); showBackward(head);
    destroy(head);
    return 0;
}
```

```
forward : [Ronggeng Dukuh Paruk] [Laskar Pelangi] [Bumi Manusia] [Filosofi Kopi]
backward: [Filosofi Kopi] [Bumi Manusia] [Laskar Pelangi] [Ronggeng Dukuh Paruk]

The lobby screen, pressing Next 5 times, then Previous 3 times:
  next     -> Laskar Pelangi
  next     -> Bumi Manusia
  next     -> Filosofi Kopi
  next     -> Ronggeng Dukuh Paruk
  next     -> Laskar Pelangi
  previous -> Ronggeng Dukuh Paruk
  previous -> Filosofi Kopi
  previous -> Bumi Manusia

remove the book on screen (Bumi Manusia):
forward : [Ronggeng Dukuh Paruk] [Laskar Pelangi] [Filosofi Kopi]
remove the first book (Ronggeng Dukuh Paruk):
forward : [Laskar Pelangi] [Filosofi Kopi]
backward: [Filosofi Kopi] [Laskar Pelangi]
```

Sesudah empat kali Next dari *Ronggeng Dukuh Paruk*, layar kembali ke tempat semula, dan yang kelima menampilkan *Laskar Pelangi* lagi; tekanan Previous menelusuri lingkaran ke arah sebaliknya, melewati sambungan antara judul pertama dan terakhir tanpa menyadarinya. Kedua tombol tidak pernah bertemu pointer null, sehingga keduanya tidak butuh uji. Penghapusan pertama membuang buku yang sedang tampil, *Bumi Manusia*, sebuah simpul di tengah. Yang kedua membuang simpul pertama, yang menguji kasus khusus 2: `head` pindah ke *Laskar Pelangi*. `destroy` memakai trik kecil yang patut diingat: ia memotong lingkaran (`head->prev->next = nullptr`) lalu melepaskan simpul-simpulnya dengan perulangan biasa dari Bab 7.

9.7 Sentinel: Melenyapkan Setiap Kasus Khusus

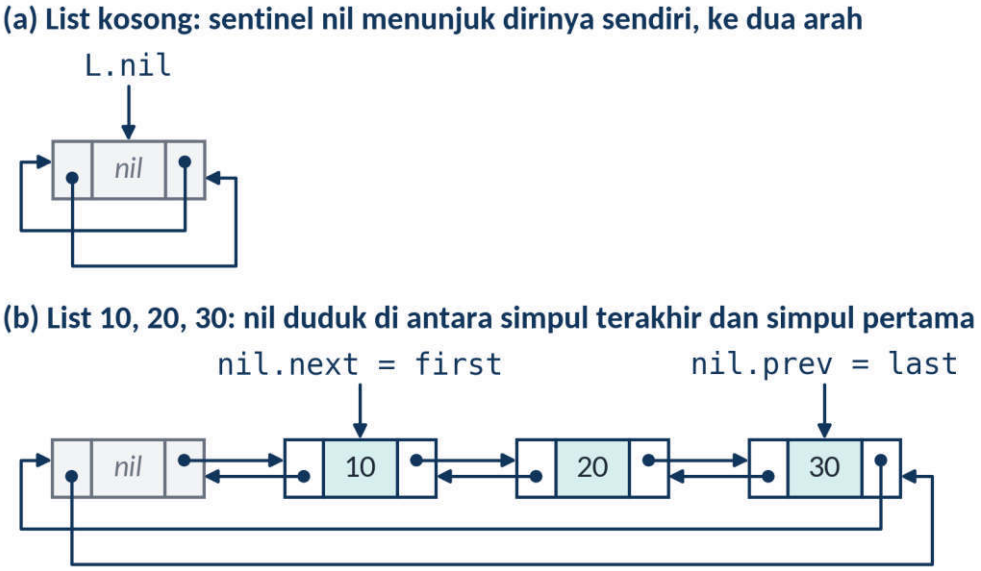
9.7.1 Gagasannya

Gabungkan dua gagasan dari dua subbab terakhir. Dari Subbab 9.5 ambillah simpul kepala, simpul tanpa data yang selalu ada. Dari Subbab 9.6 ambillah lingkaran. Hasilnya adalah *double*

linked list sirkular dengan satu simpul tanpa data, yang disebut *sentinel* (penjaga), duduk di dalam lingkaran di antara simpul terakhir dan simpul pertama.

> **DEFINISI.** *Double linked list sirkular dengan sentinel* menyimpan satu simpul tanpa data `nil` secara permanen di dalam lingkaran. `nil.next` adalah simpul pertama dan `nil.prev` adalah simpul terakhir. List kosong adalah `nil` seorang diri, dengan `nil.next = nil.prev = nil`. Karena itu setiap simpul nyata punya `prev` dan `next` yang nyata: paling buruk, *sentinel* itu.

Inilah representasi yang dipakai Cormen, Leiserson, Rivest, dan Stein untuk *linked list* dalam *Introduction to Algorithms* [2], dan inilah representasi di dalam kebanyakan *linked list* kelas produksi, seperti yang akan ditunjukkan Subbab 9.12. **Gambar 9.14** menggambar list kosong dan list 10, 20, 30.



Gambar 9.14 *Double linked list* sirkular dengan *sentinel*: (a) list kosong adalah *sentinel* seorang diri, menunjuk dirinya ke dua arah, (b) list 10, 20, 30; `nil.next` adalah simpul pertama dan `nil.prev` yang terakhir.

9.7.2 Mengapa kasus khusus lenyap

Setiap kasus khusus di Subbab 9.3 sampai 9.6 berasal dari salah satu dari dua keadaan: pointer yang null (sehingga tidak ada tetangga yang dapat diperbarui), atau pointer luar (`head`, `tail`) yang harus berubah sebagai ganti medan sebuah simpul. Dengan *sentinel*, kedua keadaan itu tidak mungkin muncul.

- **Tidak ada pointer yang pernah null.** Setiap simpul, termasuk `nil`, berada di dalam lingkaran, sehingga setiap `prev` dan `next` menyimpan alamat sebuah simpul.
- **Tidak ada pointer luar yang pernah berubah.** Satu-satunya pointer luar adalah alamat `nil`, dan `nil` tidak pernah berpindah. "Simpul pertama" dan "simpul terakhir" tidak disimpan di mana pun; keduanya dibaca sebagai `nil.next` dan `nil.prev`, yaitu medan biasa dari simpul biasa.

Jadi penyisipan dan penghapusan masing-masing dapat ditulis sekali, tanpa satu `if` pun. **Pseudokode 9.10** menyisipkan simpul sesudah simpul `p` mana pun, dan `p` boleh berupa *sentinel* itu sendiri.

Pseudokode 9.10 `LIST-INSERT-AFTER` pada list sirkular dengan *sentinel*: empat langkah **Pseudokode 9.7** tanpa ujinya.

```
LIST-INSERT-AFTER(p, x):     // p adalah simpul list, atau nil
 1.  n ← simpul baru;  n.data ← x
 2.  n.prev ← p                  // langkah 1
 3.  n.next ← p.next             // langkah 2
 4.  p.next.prev ← n             // langkah 3: tidak pernah NIL; paling buruk nil
 5.  p.next ← n                  // langkah 4
```

Pseudokode 9.11 `LIST-DELETE` pada list sirkular dengan *sentinel*: dua penugasan, tanpa uji.

```
LIST-DELETE(p):              // p adalah simpul nyata, tidak pernah nil
 1.  p.prev.next ← p.next
 2.  p.next.prev ← p.prev
 3.  free p
```

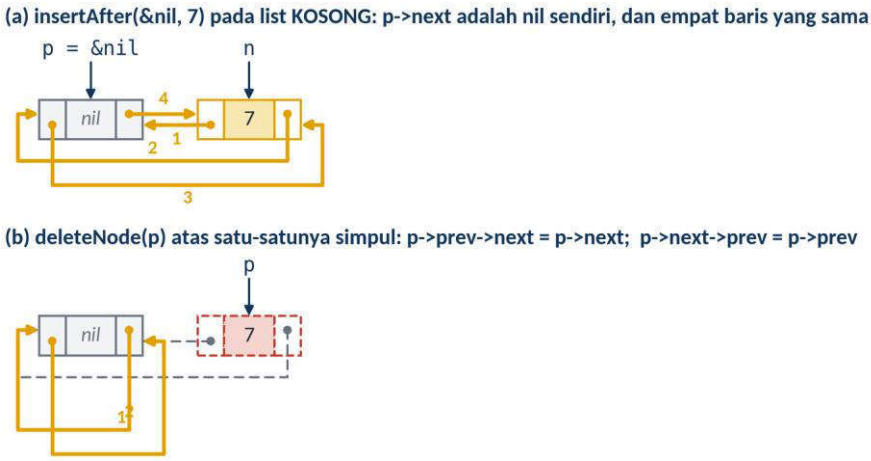
Dari kedua prosedur ini setiap operasi ujung menyusul sebagai satu baris:

$$\begin{aligned} \text{FRONT-INSERTION}(x) &= \text{LIST-INSERT-AFTER}(nil, x), \\ \text{BACK-INSERTION}(x) &= \text{LIST-INSERT-AFTER}(nil.prev, x), \\ \text{FRONT-DELETION}() &= \text{LIST-DELETE}(nil.next), \\ \text{BACK-DELETION}() &= \text{LIST-DELETE}(nil.prev). \end{aligned} \tag{9.3}$$

(Kedua penghapusan tetap mensyaratkan list tidak kosong, sebuah prasyarat operasi, bukan kasus khusus di dalamnya: menghapus dari list kosong adalah kesalahan pada representasi mana pun.)

Kedua keadaan paling berbahaya layak diperiksa dengan tangan, karena di sanalah kasus khusus dulu tinggal. **Gambar 9.15** melakukannya.

- **Penyisipan ke list kosong** (panel a). Sebelum pemanggilan, $nil.next = nil.prev = nil$, dan $p = nil$. Baris 2 membuat $n.prev = nil$. Baris 3 membuat $n.next = p.next = nil$. Baris 4 mengisi $p.next.prev$, yaitu $nil.prev$, dengan n. Baris 5 membuat $nil.next = n$. Kini $nil.next = nil.prev = n$ dan $n.next = n.prev = nil$: list satu elemen yang benar, yang simpul pertama dan terakhirnya sama-sama n.
- **Penghapusan satu-satunya simpul** (panel b). Sebelum pemanggilan, $p.prev = p.next = nil$. Baris 1 membuat $nil.next = nil$; baris 2 membuat $nil.prev = nil$. *Sentinel* menunjuk dirinya ke dua arah: list kosong.



Gambar 9.15 Dua kasus yang dulu butuh kode khusus, kini ditangani kode biasa: (a) menyisipkan sesudah *sentinel* ke list kosong, (b) menghapus satu-satunya simpul.

> **MENGAPA INI BERHASIL.** Kasus khusus adalah tempat struktur tidak *seragam*: simpul pertama tidak punya pendahulu, simpul terakhir tidak punya penerus, list kosong tidak punya

> simpul sama sekali. *Sentinel* membuat struktur seragam dengan menjamin bahwa setiap simpul nyata punya dua tetangga nyata. Harganya satu simpul memori dan sedikit perubahan cara berpikir: ujung-ujung list bukan lagi tempat, melainkan kedua tetangga `nil`.

Argumen yang sama membuktikan bahwa Persamaan (9.1) terjaga. Andaikan invarian berlaku sebelum `LIST-INSERT-AFTER`. Medan yang ditulis hanyalah $n.prev$, $n.next$, $s.prev$, dan $p.next$, dengan s adalah $p.next$ yang lama. Sesudahnya $p.next = n$ dan $n.prev = p$, sehingga pasangan (p, n) memenuhi invarian; $n.next = s$ dan $s.prev = n$, sehingga pasangan (n, s) pun demikian; setiap pasangan lain tidak tersentuh. Untuk `LIST-DELETE` argumennya lebih pendek: sesudah kedua penugasan, pendahulu dan penerus lama p saling menunjuk, dan p sudah pergi. Karena tidak ada analisis kasus, pembuktian ini mencakup list kosong, simpul pertama, simpul terakhir, dan satu-satunya simpul sekaligus. Pada list tanpa *sentinel*, pembuktian yang sama harus diulang untuk setiap cabang.

9.7.3 Mencari dengan kunci di dalam sentinel

Sentinel punya kegunaan kedua yang lebih tua. Pencarian sekuensial biasanya menguji dua hal di setiap langkah: apakah list sudah habis, dan apakah simpul sekarang berisi kunci. Sebelum mulai, kita dapat *menyimpan kunci di dalam sentinel*. Pencarian lalu tidak mungkin gagal menemukan kunci, karena *sentinel* menyimpannya, sehingga uji akhir list dapat dibuang dari perulangan. **Pseudokode 9.12** melakukan persis itu.

Pseudokode 9.12 `LIST-SEARCH-SENTINEL`: simpan kunci di dalam *sentinel*, sehingga perulangan cukup memuat satu uji per langkah.

```
LIST-SEARCH-SENTINEL(L, key):
 1.  nil.data ← key              // kini pencarian pasti berhenti
 2.  p ← nil.next
 3.  while p.data ≠ key
 4.      p ← p.next
 5.  if p = nil
 6.      return NIL              // berhenti di sentinel: kunci tidak ada
 7.  return p
```

Untuk pencarian yang tidak berhasil pada list berisi n simpul, perulangan biasa `while p ≠ nil and p.data ≠ key` melakukan n perbandingan kunci dan $n + 1$ uji akhir; versi *sentinel* melakukan $n + 1$ perbandingan kunci dan tidak ada uji akhir di dalam perulangan:

$$C_{\text{biasa}}(n) = 2n + 1, \qquad C_{\text{sentinel}}(n) = n + 1 \quad \text{uji seluruhnya.} \tag{9.4}$$

Banyaknya langkah tetap $\Theta(n)$; yang dibagi dua oleh *sentinel* adalah faktor konstantanya. Knuth menganalisis trik ini untuk pencarian sekuensial pada larik, dengan kunci ditaruh tepat sesudah elemen terakhir [3]; trik ini berlaku untuk struktur apa pun yang punya satu tempat cadangan di ujungnya, dan list sirkular dengan *sentinel* punya tepat satu.

9.7.4 Programnya

Kode 9.8 adalah *double linked list* sirkular dengan *sentinel*, sebagai kelas C++. *Sentinel*-nya adalah anggota objek, `Node nil_`, bukan simpul yang dialokasikan terpisah, sehingga ia hidup dan mati bersama list dan tidak pernah bocor. Penyalinan dilarang (seperti di Subbab 7.10.2), karena salinan objek akan berisi *sentinel* yang tetangganya menunjuk balik ke *sentinel asli*.

Kode 9.8 `cdll_sentinel.cpp`: *double linked list* sirkular dengan *sentinel*; `insertAfter` dan `deleteNode` tidak memuat kasus khusus, dan setiap operasi lain dibangun di atasnya.

```cpp
// cdll_sentinel.cpp -- a circular doubly linked list with a sentinel node, nil.
// nil.next is the first node and nil.prev the last; in an empty list both are
// &nil. Insertion and deletion have NO special cases at all.
#include <cstdio>

struct Node {
    int data;
    Node *prev;
    Node *next;
};

class SentinelList {
    Node nil_;  // the sentinel lives inside the object
public:
    // empty: nil is alone
    SentinelList() { nil_.prev = &nil_; nil_.next = &nil_; }
    SentinelList(const SentinelList &) = delete;
    SentinelList &operator=(const SentinelList &) = delete;
    ~SentinelList() { while (!isEmpty()) deleteNode(first()); }

    bool isEmpty() const { return nil_.next == &nil_; }
    Node *first() { return nil_.next; }   // &nil_ when empty
    Node *last() { return nil_.prev; }
    const Node *end() const { return &nil_; }

    // p is a node of the list or the sentinel
    Node *insertAfter(Node *p, int x) {
        // 1. n->prev = p   2. n->next = p->next
        Node *n = new Node{x, p, p->next};
        p->next->prev = n;  // 3. never NULL: at worst it is the sentinel
        p->next = n;                       // 4.
        return n;
    }
    void deleteNode(Node *p) {  // p is a real node, never the sentinel
        p->prev->next = p->next;
        p->next->prev = p->prev;
        delete p;
    }

    void frontInsertion(int x) { insertAfter(&nil_, x); }
    void backInsertion(int x) { insertAfter(nil_.prev, x); }
    void frontDeletion() { if (!isEmpty()) deleteNode(nil_.next); }
    void backDeletion() { if (!isEmpty()) deleteNode(nil_.prev); }
```

Kode 9.8 (Lanjutan) *cdll_sentinel.cpp: double linked list sirkular dengan sentinel; insertAfter dan deleteNode tidak memuat kasus khusus, dan setiap operasi lain dibangun di atasnya.*

```
    void insertSorted(int x) {  // stop on the first node >= x (or on nil)
        Node *p = nil_.next;
        while (p != &nil_ && p->data < x) p = p->next;
        insertAfter(p->prev, x);
    }

    Node *search(int key) {  // one test per step: the key is in nil too
        nil_.data = key;
        Node *p = nil_.next;
        while (p->data != key) p = p->next;
        return (p == &nil_) ? nullptr : p;
    }

    void show(const char *label) const {
        std::printf("%-24s", label);
        for (const Node *p = nil_.next; p != &nil_; p = p->next)
            std::printf("%d ", p->data);
        std::printf("| back: ");
        for (const Node *p = nil_.prev; p != &nil_; p = p->prev)
            std::printf("%d ", p->data);
        std::printf("\n");
    }
};

int main() {
    SentinelList L;
    L.show("empty:");
    L.frontInsertion(20);  // into an EMPTY list: the same four lines
    L.frontInsertion(10);
    L.backInsertion(40);
    L.show("10, 20, 40:");
    L.insertSorted(30);
    L.insertSorted(5);
    L.insertSorted(50);
    L.show("sorted 30, 5, 50:");

    Node *p = L.search(30);
    L.insertAfter(p, 35);
    L.show("insertAfter(30, 35):");
    L.deleteNode(L.search(35));
    L.show("delete 35:");
    std::printf("%-24s%s\n", "search(99):",
                L.search(99) == nullptr ? "not found" : "found");
```

Kode 9.8 (Lanjutan) *cdll_sentinel.cpp: double linked list sirkular dengan sentinel; insertAfter dan deleteNode tidak memuat kasus khusus, dan setiap operasi lain dibangun di atasnya.*

```
    L.frontDeletion();
    L.backDeletion();
    L.show("front + back del:");
    // the last deletion needs no special case
    while (!L.isEmpty()) L.frontDeletion();
    L.show("delete the rest:");
    std::printf("%-24s%s\n", "nil points at itself:",
                (L.first() == L.end() && L.last() == L.end()) ? "yes" : "no");
    return 0;
}
```

```
empty:                  | back:
10, 20, 40:             10 20 40 | back: 40 20 10
sorted 30, 5, 50:       5 10 20 30 40 50 | back: 50 40 30 20 10 5
insertAfter(30, 35):    5 10 20 30 35 40 50 | back: 50 40 35 30 20 10 5
delete 35:              5 10 20 30 40 50 | back: 50 40 30 20 10 5
search(99):             not found
front + back del:       10 20 30 40 | back: 40 30 20 10
delete the rest:        | back:
nil points at itself:   yes
```

Bacalah `insertAfter` dan `deleteNode` di listing: empat penugasan dan dua penugasan, tanpa satu `if` pun. Pemanggilan pertama, `L.frontInsertion(20)`, menyisipkan ke list kosong, dan pemanggilan terakhir di perulangan `while` menghapus satu-satunya simpul yang tersisa; keduanya memakai baris-baris yang persis sama dengan pemanggilan lain. Baris keluaran terakhir memastikan bahwa list kosong kembali ke bentuk awalnya, dengan *sentinel* menunjuk dirinya sendiri.

9.7.5 Kasus khusus, dihitung

Tabel 9.3 menghitung uji kasus khusus pada penyisipan sesudah simpul dan penghapusan simpul, dua operasi yang menjadi dasar semua operasi lain, pada setiap representasi dalam bab ini.

Tabel 9.3 Uji kasus khusus dalam dua operasi dasar, menurut representasinya. Hanya *sentinel* yang menghapus semuanya.

Representasi	Uji dalam *insert-after*	Uji dalam *delete-node*	Yang dijaga uji itu
non-sirkular, `head` dan `tail`, tanpa simpul kepala (Subbab 9.4)	1	2	ujung belakang saat penyisipan; kedua ujung saat penghapusan
non-sirkular dengan simpul kepala (Subbab 9.5)	1	1	ujung belakang saja
sirkular, lewat `head`, tanpa *sentinel* (Subbab 9.6)	1 (list kosong)	2 (satu-satunya simpul, simpul pertama)	list kosong; `head` yang berpindah
sirkular dengan *sentinel* (Subbab 9.7)	0	0	tidak ada

9.8 Deque untuk Meja Pengembalian

9.8.1 Apa itu deque

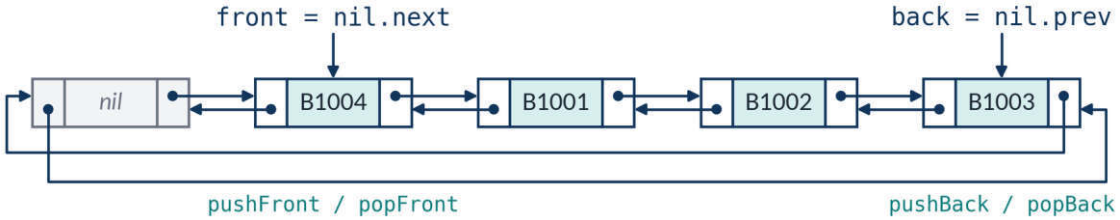
Bab 5 membangun *stack*, yang menambah dan membuang di satu ujung, dan *queue*, yang menambah di satu ujung dan membuang di ujung lain. *Deque* (*double-ended queue*, antrean berujung ganda, dilafalkan “dek”) memungkinkan keduanya di kedua ujung.

> **DEFINISI.** *Deque* adalah barisan yang mendukung `pushFront`, `pushBack`, `popFront`, `popBack`, `front`, `back`, `isEmpty`, dan `size`. *Deque* yang hanya dipakai lewat `pushFront` dan `popFront` adalah *stack*; yang dipakai hanya lewat `pushBack` dan `popFront` adalah *queue*.

Double linked list sirkular dengan *sentinel* mengimplementasikan setiap operasi *deque* dalam $O(1)$ menurut Persamaan (9.3), tanpa batas kapasitas. (Larik sirkular, seperti pada Soal 8.7 di Subbab 8.8.2, juga memberi $O(1)$ di kedua ujung, tetapi punya kapasitas, atau harus disalin ketika membesar.)

9.8.2 Meja pengembalian Pustaka Ceria

Buku yang dikembalikan di meja Pustaka Ceria ditaruh di troli dan dirak menurut urutan kedatangannya: itulah *queue*. Namun ada dua hal yang melanggar disiplin *queue*. Buku kembalian yang sudah dipesan pembaca yang menunggu harus ditangani *pertama*, jadi ia masuk ke depan troli. Dan ketika pustakawan menyadari bahwa buku yang terakhir dipindai ternyata salah pindai, buku itu diambil lagi dari *belakang* troli. Penambahan di kedua ujung dan pengambilan di kedua ujung: sebuah *deque*. **Gambar 9.16** memperlihatkan troli sesudah beberapa kejadian pertama.



Gambar 9.16 Meja pengembalian sebagai *deque* di atas *double linked list* sirkular dengan *sentinel*: depan adalah `nil.next`, belakang adalah `nil.prev`, dan keempat operasi bernilai $O(1)$.

Kode 9.9 mengimplementasikan *deque* sebagai kelas yang memegang *sentinel* yang dialokasikan di *heap*, lalu memutar ulang satu sore di meja pengembalian.

Kode 9.9 `deque.cpp`: *deque* berisi buku di atas *double linked list* sirkular dengan *sentinel*, dipakai di meja pengembalian Pustaka Ceria.

```cpp
// deque.cpp -- a deque (double-ended queue) built on a circular doubly linked
// list with a sentinel: the returns desk of the Pustaka Ceria library.
#include <iostream>
#include <stdexcept>
#include <string>

struct Book {
    std::string id;
    std::string title;
};

class Deque {
    struct Node {
        Book data;
        Node *prev;
        Node *next;
    };
    Node *nil_;  // sentinel: nil_->next = front, nil_->prev = back
    int size_ = 0;

    void linkAfter(Node *p, const Book &b) {
        Node *n = new Node{b, p, p->next};
        p->next->prev = n;
        p->next = n;
        ++size_;
    }
    Book unlink(Node *p) {
        if (p == nil_) throw std::underflow_error("pop from an empty deque");
        Book b = p->data;
        p->prev->next = p->next;
        p->next->prev = p->prev;
        delete p;
        --size_;
        return b;
    }
public:
    Deque() : nil_(new Node{Book{}, nullptr, nullptr}) {
        nil_->prev = nil_->next = nil_;       // empty: the sentinel is alone
    }
    Deque(const Deque &) = delete;
    Deque &operator=(const Deque &) = delete;
    ~Deque() {
        while (!isEmpty()) popFront();
        delete nil_;
    }
```

Kode 9.9 (Lanjutan) *deque.cpp: deque berisi buku di atas double linked list sirkular dengan sentinel, dipakai di meja pengembalian Pustaka Ceria.*

```
    bool isEmpty() const { return size_ == 0; }
    int size() const { return size_; }
    void pushFront(const Book &b) { linkAfter(nil_, b); }          // O(1)
    void pushBack(const Book &b) { linkAfter(nil_->prev, b); }     // O(1)
    Book popFront() { return unlink(nil_->next); }                 // O(1)
    Book popBack() { return unlink(nil_->prev); }                  // O(1)
    // precondition: !isEmpty()
    const Book &front() const { return nil_->next->data; }
    const Book &back() const { return nil_->prev->data; }

    void show() const {
        std::cout << "  front |";
        for (Node *p = nil_->next; p != nil_; p = p->next)
            std::cout << ' ' << p->data.id;
        std::cout << " | back   (size " << size_ << ")\n";
    }
};

int main() {
    Deque desk;
    std::cout << "Books returned at the desk go to the back:\n";
    desk.pushBack({"B1001", "Laskar Pelangi"});
    desk.pushBack({"B1002", "Bumi Manusia"});
    desk.pushBack({"B1003", "Filosofi Kopi"});
    desk.show();

    std::cout << "B1004 is reserved by a waiting reader, "
                 "so it jumps to the front:\n";
    desk.pushFront({"B1004", "Cantik Itu Luka"});
    desk.show();

    Book b = desk.popFront();
    std::cout << "shelved or handed out first: "
              << b.id << " " << b.title << '\n';
    b = desk.popBack();
    std::cout << "last scan was a mistake, taken back off: "
              << b.id << " " << b.title << '\n';
    desk.show();

    desk.pushBack({"B1005", "Ronggeng Dukuh Paruk"});
    std::cout << "front is " << desk.front().id
              << ", back is " << desk.back().id << '\n';
    desk.show();
```

Kode 9.9 (Lanjutan) *deque.cpp: deque berisi buku di atas double linked list sirkular dengan sentinel, dipakai di meja pengembalian Pustaka Ceria.*

```
    std::cout << "closing time, the desk is emptied from both ends:\n";
    bool fromFront = true;
    while (!desk.isEmpty()) {
        b = fromFront ? desk.popFront() : desk.popBack();
        std::cout << "  " << (fromFront ? "front: " : "back:  ")
                  << b.id << '\n';
        fromFront = !fromFront;
    }
    try {
        desk.popBack();
    } catch (const std::underflow_error &e) {
        std::cout << "one more popBack(): " << e.what() << '\n';
    }
    return 0;
}
```

```
Books returned at the desk go to the back:
  front | B1001 B1002 B1003 | back   (size 3)
B1004 is reserved by a waiting reader, so it jumps to the front:
  front | B1004 B1001 B1002 B1003 | back   (size 4)
shelved or handed out first: B1004 Cantik Itu Luka
last scan was a mistake, taken back off: B1003 Filosofi Kopi
  front | B1001 B1002 | back   (size 2)
front is B1001, back is B1005
  front | B1001 B1002 B1005 | back   (size 3)
closing time, the desk is emptied from both ends:
  front: B1001
  back:  B1005
  front: B1002
one more popBack(): pop from an empty deque
```

Setiap operasi hanya sedalam dua fungsi pembantu privat: `linkAfter` adalah **Pseudokode 9.10** dan `unlink` adalah **Pseudokode 9.11**. `unlink` juga menjadi tempat *deque* kosong dikenali: mengambil dari *deque* kosong berarti meminta pelepasan `nil_->next`, yaitu *sentinel* itu sendiri, dan kelas ini melempar `std::underflow_error` alih-alih menghancurkan *sentinel*-nya sendiri, seperti yang tampak di baris keluaran terakhir. Berbeda dengan *sentinel* pada **Kode 9.8**, yang satu ini dialokasikan dengan `new` (medan `Book`-nya menyimpan dua *string*), sehingga destruktor harus melepaskannya sesudah semua simpul nyata; Subbab 9.11 memperlihatkan kebocoran yang terjadi bila tidak. Versi Java dan Python, `DequeDesk.java` dan `deque_desk.py` di folder kode, mencetak keluaran yang sama.

9.9 Daftar “Baru Dipinjam”: Pola LRU

9.9.1 Masalahnya

Kios katalog Pustaka Ceria punya panel berjudul *Baru dipinjam*. Panel itu menampilkan k judul *berbeda* yang paling baru dipinjam, yang terbaru di atas; perpustakaan memakai $k = 4$. Ketika sebuah judul yang sudah ada di panel dipinjam, judul itu naik ke atas. Ketika judul baru dipinjam dan panel sudah penuh, judul yang *paling lama tidak dipinjam* jatuh dari bawah.

Inilah aplikasi *double linked list* yang paling terkenal di luar buku ajar: kebijakan *least recently used* (LRU, yang paling lama tidak dipakai), yang memutuskan apa yang dibuang dari sebuah *cache* (*cache* gambar peramban web, *cache* halaman basis data, tabel halaman sistem operasi), dan yang bekerja persis seperti panel ini. Setiap peminjaman harus melakukan tiga hal dengan cepat: mengetahui apakah judulnya sudah ada di panel, memindahkan sebuah judul ke atas, dan membuang judul di paling bawah.

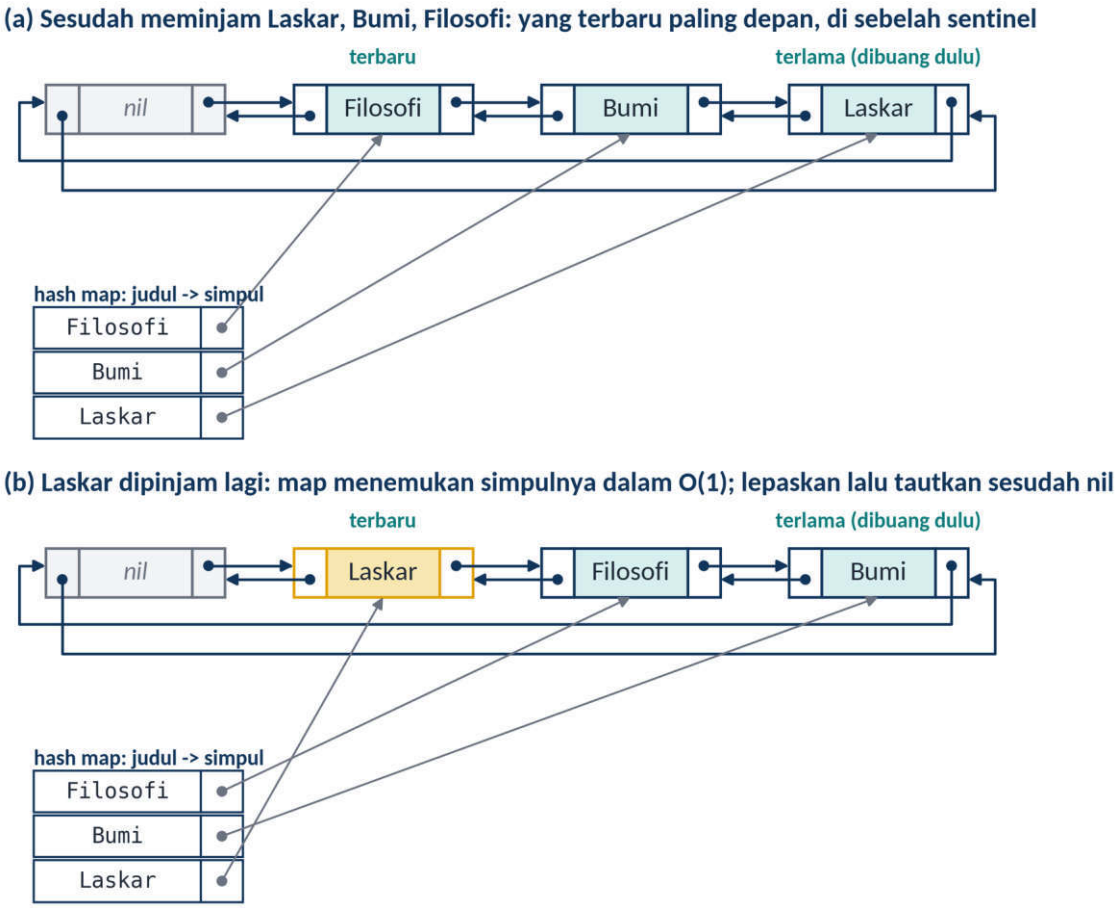
9.9.2 Dua struktur yang bekerja bersama

Tidak ada satu pun struktur dalam buku ini sejauh ini yang melakukan ketiganya dalam $O(1)$. List menjaga urutan, tetapi menemukan judul di dalamnya adalah pencarian $\Theta(k)$. *Hash map*, yang dipelajari Bab 15 (`std::unordered_map` di C++, `HashMap` di Java, `dict` di Python), menemukan kunci dalam waktu harapan $O(1)$ tetapi tidak menjaga urutan. Pola LRU memakai keduanya:

- *double linked list* sirkular dengan *sentinel* menyimpan judul-judul menurut kebaruannya: `nil.next` yang terbaru, `nil.prev` yang terlama;
- *hash map* memetakan setiap judul ke *alamat simpulnya* di dalam list.

Gambar 9.17 memperlihatkan panel sesudah tiga peminjaman, lalu sesudah judul pertama dipinjam lagi. *Map* menemukan simpul *Laskar* dalam $O(1)$; simpul itu dilepas dari tempatnya dengan **Pseudokode 9.11** (tanpa dibebaskan) lalu ditautkan lagi sesudah `nil` dengan

Pseudokode 9.10. Tidak satu pun dari kedua langkah itu perlu tahu di mana posisi simpul tadi di dalam list. Inilah inti seluruh rancangan, dan alasan mengapa list-nya harus *double linked*: *map* memberikan pointer ke simpul di tengah list, dan hanya simpul *double linked* yang dapat dilepas dari tengah tanpa mencari pendahulunya.



Gambar 9.17 Panel "baru dipinjam": *double linked list* sirkular dengan *sentinel* menyimpan judul dari yang terbaru sampai yang terlama, dan *hash map* menunjuk dari setiap judul ke simpulnya. (a) Sesudah tiga peminjaman; (b) sesudah *Laskar* dipinjam lagi dan dipindah ke depan; *map*-nya sendiri tidak berubah.

Pseudokode 9.13 adalah seluruh algoritmanya.

Pseudokode 9.13 `BORROW` pada panel "baru dipinjam": satu pencarian *hash map* ditambah sejumlah tetap penugasan pointer.

```
BORROW(title):
 1.  p ← map.find(title)
 2.  if p ≠ NIL                  // sudah ada di panel
 3.      lepaskan p              // dua penugasan (LIST-DELETE tanpa free)
 4.      tautkan p sesudah nil   // empat penugasan: kini p yang terbaru
 5.      return
 6.  n ← simpul baru untuk title;  tautkan n sesudah nil;  map[title] ← n
 7.  if map.size > k             // panel meluap
 8.      old ← nil.prev          // judul yang paling lama tidak dipinjam
 9.      lepaskan old;  map.erase(old.title);  free old
```

Setiap baris bernilai $O(1)$, dengan operasi *hash map* bernilai $O(1)$ secara rata-rata, sehingga

$$T_{\text{BORROW}} = O(1) \text{ harapan, berapa pun } k, \tag{9.5}$$

sedangkan panel yang sama bila disimpan dalam larik, atau dalam *single linked list* tanpa *map*, membutuhkan pencarian $\Theta(k)$ untuk setiap peminjaman, dan *single linked list* membutuhkan lagi perjalanan $\Theta(k)$ untuk menemukan pendahulu simpul yang dipindahkan.

9.9.3 Programnya

Kode 9.10 mengimplementasikan panel dengan `std::unordered_map` dari judul ke simpul dan memutar ulang delapan peminjaman dengan $k = 4$.

Kode 9.10 `recently_borrowed.cpp`: panel "baru dipinjam", daftar LRU yang dibangun dari *double linked list* sirkular dengan *sentinel* dan `std::unordered_map` dari judul ke simpul.

```
// recently_borrowed.cpp -- the "recently borrowed" panel of Pustaka Ceria: the
// k most recently borrowed distinct titles, newest first (an LRU list). A
// circular doubly linked list with a sentinel keeps the order; a hash map finds
// a title's node in O(1).
#include <iostream>
#include <string>
#include <unordered_map>

class RecentlyBorrowed {
    struct Node {
        std::string title;
        Node *prev;
        Node *next;
    };
    Node nil_;  // sentinel: nil_.next = newest, nil_.prev = oldest
    std::unordered_map<std::string, Node *> where_;   // title -> its node
    std::size_t capacity_;

    void unlink(Node *p) {
        p->prev->next = p->next;
        p->next->prev = p->prev;
    }
    void linkFront(Node *p) {             // put p right after the sentinel
        p->prev = &nil_;
        p->next = nil_.next;
        nil_.next->prev = p;
        nil_.next = p;
    }
public:
    explicit RecentlyBorrowed(std::size_t k)
        // empty: the sentinel is alone
        : nil_{"", &nil_, &nil_}, capacity_(k) {}
    RecentlyBorrowed(const RecentlyBorrowed &) = delete;
    RecentlyBorrowed &operator=(const RecentlyBorrowed &) = delete;
    ~RecentlyBorrowed() {
        while (nil_.next != &nil_) {
            Node *del = nil_.next;
            unlink(del);
            delete del;
        }
    }
```

Kode 9.10 (Lanjutan) *recently_borrowed.cpp: panel "baru dipinjam", daftar LRU yang dibangun dari double linked list sirkular dengan sentinel dan std::unordered_map dari judul ke simpul.*

```
    // Records one borrowing. Returns the title that fell off the panel, or "".
    std::string borrow(const std::string &title) {
        auto it = where_.find(title);
        if (it != where_.end()) {  // already on the panel: move it to the front
            unlink(it->second);
            linkFront(it->second);
            return "";
        }
        Node *n = new Node{title, nullptr, nullptr};
        linkFront(n);
        where_[title] = n;
        if (where_.size() <= capacity_) return "";
        Node *oldest = nil_.prev;  // over capacity: evict the least recent
        std::string gone = oldest->title;
        unlink(oldest);
        where_.erase(gone);
        delete oldest;
        return gone;
    }

    void show() const {
        std::cout << "   panel:";
        for (const Node *p = nil_.next; p != &nil_; p = p->next)
            std::cout << " [" << p->title << "]";
        std::cout << '\n';
    }
};

int main() {
    RecentlyBorrowed panel(4);
    const char *loans[] = {"Laskar Pelangi", "Bumi Manusia", "Filosofi Kopi",
                           "Laskar Pelangi", "Cantik Itu Luka",
                           "Ronggeng Dukuh Paruk", "Filosofi Kopi",
                           "Negeri 5 Menara"};
    for (const char *t : loans) {
        std::string gone = panel.borrow(t);
        std::cout << "borrow " << t;
        if (!gone.empty()) std::cout << "  (drops off: " << gone << ")";
        std::cout << '\n';
        panel.show();
    }
    return 0;
}
```

```
borrow Laskar Pelangi
   panel: [Laskar Pelangi]
borrow Bumi Manusia
   panel: [Bumi Manusia] [Laskar Pelangi]
borrow Filosofi Kopi
   panel: [Filosofi Kopi] [Bumi Manusia] [Laskar Pelangi]
borrow Laskar Pelangi
   panel: [Laskar Pelangi] [Filosofi Kopi] [Bumi Manusia]
borrow Cantik Itu Luka
   panel: [Cantik Itu Luka] [Laskar Pelangi] [Filosofi Kopi] [Bumi Manusia]
borrow Ronggeng Dukuh Paruk  (drops off: Bumi Manusia)
   panel: [Ronggeng Dukuh Paruk] [Cantik Itu Luka] [Laskar Pelangi] [Filosofi Kopi]
borrow Filosofi Kopi
   panel: [Filosofi Kopi] [Ronggeng Dukuh Paruk] [Cantik Itu Luka] [Laskar Pelangi]
borrow Negeri 5 Menara  (drops off: Laskar Pelangi)
   panel: [Negeri 5 Menara] [Filosofi Kopi] [Ronggeng Dukuh Paruk] [Cantik Itu Luka]
```

Telusuri kedelapan peminjaman itu. Tiga judul pertama mengisi panel dari atas. Peminjaman keempat adalah *Laskar Pelangi* lagi: ia ditemukan di *map* dan dipindahkan dari bawah ke atas,

sehingga tidak ada judul yang dibuang. *Cantik Itu Luka* mengisi tempat keempat. *Ronggeng Dukuh Paruk* membuat panel meluap, dan judul di paling bawah, *Bumi Manusia*, jatuh: ia dipinjam kedua, dan sejak itu tidak disentuh lagi. *Filosofi Kopi* naik ke atas, dan akhirnya *Negeri 5 Menara* mendorong keluar *Laskar Pelangi*, yang sudah menjadi yang paling lama tidak dipinjam. Versi Java dan Python, `RecentlyBorrowed.java` dan `recently_borrowed.py`, mencetak baris-baris yang sama; Subbab 9.12 memperlihatkan bahwa kedua bahasa itu juga menyediakan seluruh pola ini siap pakai.

9.10 Perbandingan Ongkos: Single Linked, Double Linked, Larik

9.10.1 Tabelnya

Tabel 9.4 membandingkan tiga struktur sekuensial dalam buku ini operasi demi operasi: larik dinamis (`std::vector`, `ArrayList` di Java, `list` di Python), *single linked list* dengan *head* dan *tail* dari Subbab 7.8, dan *double linked list* dengan *head* dan *tail* (atau, yang setara, dengan *sentinel*) dari bab ini. "Posisi yang dipegang" berarti program sudah punya pointer atau iterator ke simpulnya, atau indeks ke dalam larik.

Tabel 9.4 Larik dinamis, *single linked list*, dan *double linked list* dibandingkan, untuk n elemen.

Operasi	Larik dinamis	*Single linked* (*head*, *tail*)	*Double linked* (*head*, *tail* atau *sentinel*)
mengakses elemen pada indeks k	$O(1)$	$\Theta(k)$	$\Theta(\min(k, n-1-k))$
menyisipkan di depan	$\Theta(n)$	$O(1)$	$O(1)$
menghapus di depan	$\Theta(n)$	$O(1)$	$O(1)$
menyisipkan di belakang	$O(1)$ teramortisasi	$O(1)$	$O(1)$
menghapus di belakang	$O(1)$	$\Theta(n)$	$O(1)$
menyisipkan sesudah posisi yang dipegang	$\Theta(n)$ (geser)	$O(1)$	$O(1)$
menyisipkan sebelum posisi yang dipegang	$\Theta(n)$ (geser)	$\Theta(n)$	$O(1)$
menghapus posisi yang dipegang	$\Theta(n)$ (geser)	$\Theta(n)$	$O(1)$
mencari berdasarkan nilai, tak terurut	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
mencari berdasarkan nilai, terurut	$\Theta(\lg n)$ (pencarian biner)	$\Theta(n)$	$\Theta(n)$
menyusuri maju	$\Theta(n)$, ramah *cache*	$\Theta(n)$	$\Theta(n)$
menyusuri mundur	$\Theta(n)$	$\Theta(n^2)$, atau $\Theta(n)$ dengan *stack*	$\Theta(n)$
memori tambahan per elemen	tidak ada (ditambah kapasitas cadangan)	satu pointer	dua pointer
elemen lain berpindah saat sisip atau hapus	ya	tidak pernah	tidak pernah

Tiga baris layak dikomentari.

Akses berdasarkan indeks. *Double linked list* dengan `tail` dapat berjalan ke indeks k dari ujung yang lebih dekat. Dari depan dibutuhkan k langkah; dari belakang, $n-1-k$ langkah. Jadi ongkosnya

$$T_{\text{reach}}(k) = \min(k,\ n-1-k), \qquad \frac{1}{n}\sum_{k=0}^{n-1} \min\left(k,\ n-1-k\right) \approx \frac{n}{4}. \tag{9.6}$$

Rata-rata itu diperoleh karena nilai minimum naik dari 0 sampai sekitar $n/2$ pada separuh pertama indeks lalu turun kembali pada separuh kedua, sebuah segitiga yang tinggi rata-ratanya separuh puncaknya, $n/4$. Itu separuh rata-rata *single linked list*, $(n-1)/2$ menurut rumus Subbab 7.3.1, tetapi tetap $\Theta(n)$. `LinkedList.get(i)` milik Java melakukan persis ini: dokumentasinya menyatakan bahwa operasi berindeks menyusuri list "from the beginning or the end, whichever is closer to the specified index" (dari awal atau dari akhir, mana yang lebih dekat ke indeks yang diminta) [4].

Menghapus di belakang, dan di posisi yang dipegang. Kedua baris inilah alasan bab ini ada: dari $\Theta(n)$ pada *single linked list* menjadi $O(1)$ pada *double linked list*.

Memori tambahan dan elemen yang berpindah. Harganya satu pointer lagi per elemen (Persamaan (9.2)); hadiahnya, yang juga dimiliki *single linked list*, adalah bahwa menyisipkan atau menghapus tidak pernah memindahkan elemen lain, sehingga pointer ke simpul-simpul lain tetap sah. Untuk larik, kedua kolom itu terbalik.

9.10.2 Mengukur apa yang dibeli pointer kedua

Perbedaan pada baris penghapusan di belakang mudah diramalkan dengan tepat. Mengosongkan *single linked list* berisi n simpul dari belakang melakukan n penghapusan di belakang. Penghapusan yang mendapati $m \geq 2$ simpul harus berjalan $m-2$ langkah ke simpul sebelum `tail`, dan yang terakhir (satu simpul) tidak berjalan sama sekali, sehingga jumlah seluruh langkah berjalan adalah

$$\sum_{m=2}^{n} (m-2) = 0 + 1 + \cdots + (n-2) = \frac{(n-1)(n-2)}{2}, \tag{9.7}$$

yaitu $\Theta(n^2)$, dibandingkan tepat nol langkah berjalan (dan pekerjaan total $\Theta(n)$) untuk *double linked list*. Percobaan kedua menghapus separuh simpul, dipilih secara acak, ketika program sudah memegang pointer ke masing-masing: *single linked list* harus mencari pendahulu setiap korban dari `head`, *double linked list* melepasnya dalam $O(1)$.

Kode 9.11 menjalankan kedua percobaan untuk n dari 2.500 sampai 20.000, masing-masing tiga kali, dan melaporkan jalan tercepat.

Kode 9.11 `bench.cpp`: mengosongkan list dari belakang, dan menghapus simpul lewat pointer yang sudah dipegang program, untuk *single* dan *double linked list*.

```cpp
// bench.cpp -- what the prev pointer buys, measured.
// (1) Empty a list of n nodes from the back: singly linked with head and tail
//     (must walk to the node before tail) versus doubly linked (tail->prev).
// (2) Delete n/2 nodes in random order, given a pointer to each node: singly
//     linked (must search for the predecessor) versus doubly linked (O(1)
//     unlink).
#include <algorithm>
#include <chrono>
#include <cstdio>
#include <random>
#include <vector>

struct SNode { int data; SNode *next; };
struct DNode { int data; DNode *prev; DNode *next; };

using Clock = std::chrono::steady_clock;
double ms(Clock::time_point a, Clock::time_point b) {
    return std::chrono::duration<double, std::milli>(b - a).count();
}

double sllBackDeletions(int n) {
    SNode *head = nullptr, *tail = nullptr;
    for (int i = 0; i < n; ++i) {
        SNode *s = new SNode{i, nullptr};
        if (!head) head = s; else tail->next = s;
        tail = s;
    }
    auto t0 = Clock::now();
    while (head != nullptr) {
        if (head == tail) { delete head; head = tail = nullptr; break; }
        SNode *aux = head;
        while (aux->next != tail) aux = aux->next;   // the walk: Theta(n)
        delete tail;
        tail = aux;
        tail->next = nullptr;
    }
    return ms(t0, Clock::now());
}
```

Kode 9.11 (Lanjutan) *bench.cpp: mengosongkan list dari belakang, dan menghapus simpul lewat pointer yang sudah dipegang program, untuk single dan double linked list.*

```
double dllBackDeletions(int n) {
    DNode *head = nullptr, *tail = nullptr;
    for (int i = 0; i < n; ++i) {
        DNode *d = new DNode{i, tail, nullptr};
        if (!head) head = d; else tail->next = d;
        tail = d;
    }
    auto t0 = Clock::now();
    while (tail != nullptr) {
        DNode *del = tail;
        tail = tail->prev;                              // one step: O(1)
        if (tail) tail->next = nullptr; else head = nullptr;
        delete del;
    }
    return ms(t0, Clock::now());
}

template <typename N>
std::vector<N *> randomHalf(std::vector<N *> v, unsigned seed) {
    std::mt19937 rng(seed);
    std::shuffle(v.begin(), v.end(), rng);
    v.resize(v.size() / 2);
    return v;
}

double sllDeleteByPointer(int n) {
    std::vector<SNode *> all;
    SNode *head = nullptr, *tail = nullptr;
    for (int i = 0; i < n; ++i) {
        SNode *s = new SNode{i, nullptr};
        if (!head) head = s; else tail->next = s;
        tail = s;
        all.push_back(s);
    }
    std::vector<SNode *> victims = randomHalf(all, 42);
    auto t0 = Clock::now();
    for (SNode *p : victims) {
        if (p == head) { head = p->next; delete p; continue; }
        SNode *q = head;
        while (q->next != p) q = q->next;  // find the predecessor: O(n)
        q->next = p->next;
        delete p;
    }
    double t = ms(t0, Clock::now());
    while (head) { SNode *d = head; head = head->next; delete d; }
    return t;
```

Kode 9.11 (Lanjutan) *bench.cpp: mengosongkan list dari belakang, dan menghapus simpul lewat pointer yang sudah dipegang program, untuk single dan double linked list.*

```
}

double dllDeleteByPointer(int n) {
    std::vector<DNode *> all;
    DNode *head = nullptr, *tail = nullptr;
    for (int i = 0; i < n; ++i) {
        DNode *d = new DNode{i, tail, nullptr};
        if (!head) head = d; else tail->next = d;
        tail = d;
        all.push_back(d);
    }
    std::vector<DNode *> victims = randomHalf(all, 42);
    auto t0 = Clock::now();
    for (DNode *p : victims) {                        // unlink: O(1)
        if (p->prev) p->prev->next = p->next; else head = p->next;
        if (p->next) p->next->prev = p->prev; else tail = p->prev;
        delete p;
    }
    double t = ms(t0, Clock::now());
    while (head) { DNode *d = head; head = head->next; delete d; }
    return t;
}

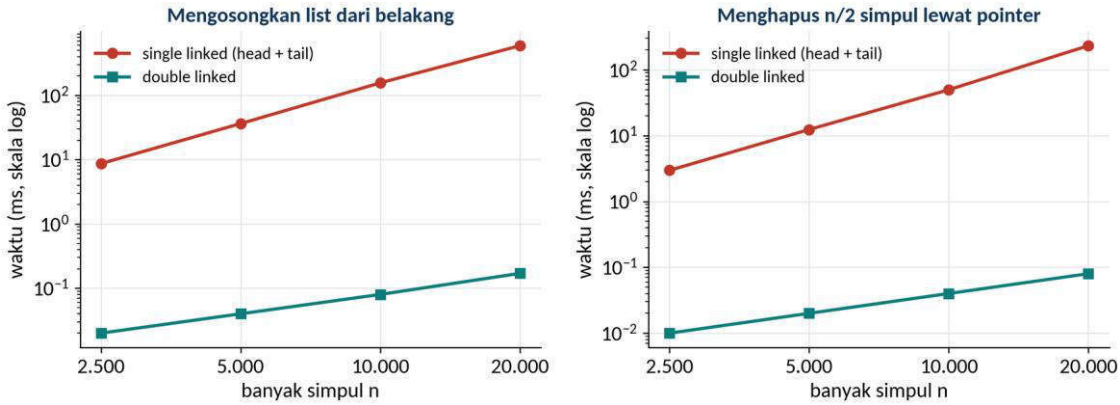
double best(double (*f)(int), int n) {           // the fastest of three runs
    double t = f(n);
    for (int r = 0; r < 2; ++r) t = std::min(t, f(n));
    return t;
}

int main() {
    std::printf("%8s | %14s %14s | %14s %14s\n", "n",
                "SLL back del", "DLL back del", "SLL by ptr", "DLL by ptr");
    for (int n = 2500; n <= 20000; n *= 2) {
        std::printf("%8d | %11.2f ms %11.2f ms | %11.2f ms %11.2f ms\n", n,
                    best(sllBackDeletions, n), best(dllBackDeletions, n),
                    best(sllDeleteByPointer, n), best(dllDeleteByPointer, n));
    }
    return 0;
}
```

Satu kali jalan di mesin yang dipakai untuk buku ini (dikompilasi dengan `-O2`) mencetak:

```
       n |   SLL back del   DLL back del |     SLL by ptr     DLL by ptr
    2500 |        8.68 ms        0.02 ms |        2.97 ms        0.01 ms
    5000 |       36.40 ms        0.04 ms |       12.38 ms        0.02 ms
   10000 |      156.41 ms        0.08 ms |       49.73 ms        0.04 ms
   20000 |      592.91 ms        0.17 ms |      232.24 ms        0.08 ms
```

Gambar 9.18 memplot angka-angka yang sama pada sumbu logaritmik. Setiap penggandaan n mengalikan waktu *single linked* sekitar 4 kali (8,68; 36,40; 156,41; 592,91 ms), ciri $\Theta(n^2)$ dari Persamaan (9.7), dan waktu *double linked* sekitar 2 kali (0,02; 0,04; 0,08; 0,17 ms), ciri $\Theta(n)$. Pada $n = 20.000$ *double linked list* mengosongkan dirinya dari belakang kira-kira 3.500 kali lebih cepat, dan menghapus 10.000 simpul yang dipegang kira-kira 2.900 kali lebih cepat (232,24 ms dibandingkan 0,08 ms). Pada 20.000 simpul, Persamaan (9.7) memberikan sekitar 2×10^8 langkah berjalan, yang diselesaikan dalam 0,59 detik, sekitar 3 ns per langkah: *single linked list* tidak lambat per langkah, ia hanya mengambil terlalu banyak langkah.



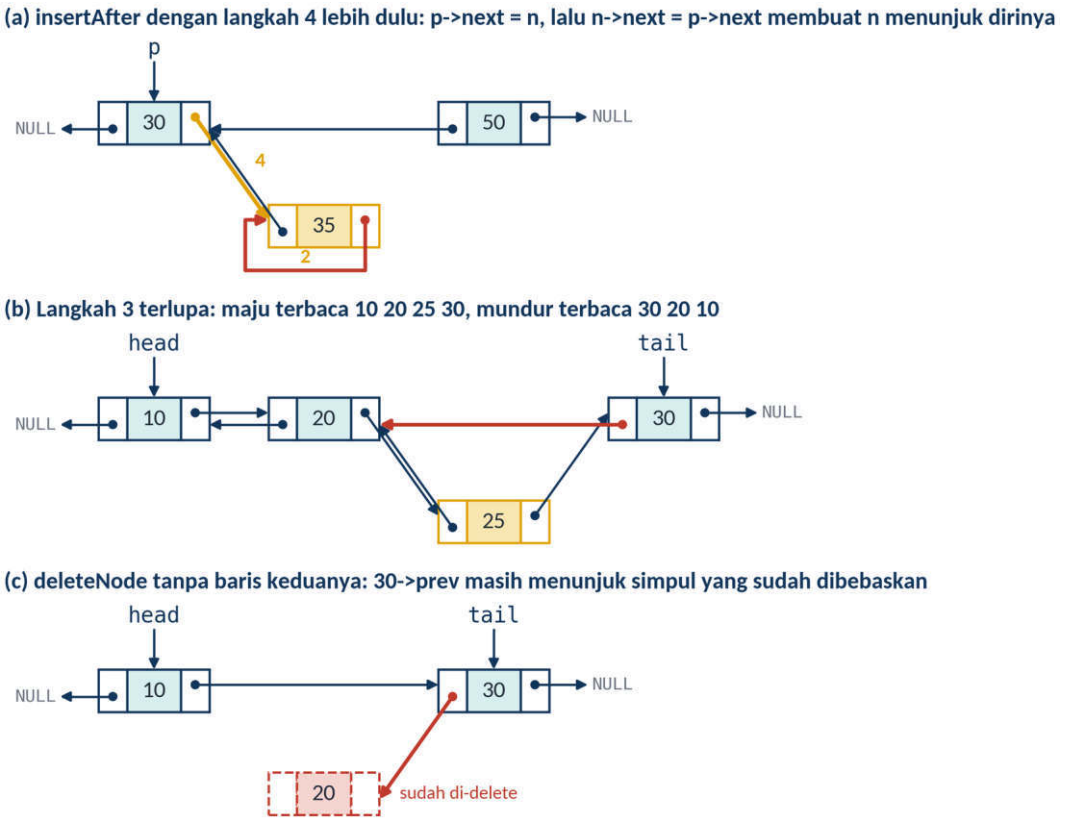
Gambar 9.18 Hasil pengukuran **Kode 9.11** pada sumbu logaritmik: waktu *single linked list* tumbuh empat kali per penggandaan n (kuadratik), waktu *double linked list* dua kali (linear).

Waktu-waktunya sendiri bergantung pada mesin; bentuknya tidak. Pengukuran ini juga mengingatkan pada Subbab 7.12.4: ketika posisi *tidak* sudah dipegang, pencarianlah yang mendominasi, dan larik biasanya menang. *Double linked list* adalah alat yang tepat ketika program menyimpan pointer ke tengah-tengah barisan dan perlu mengubahnya di sana.

9.11 Keamanan Memori untuk Tautan Dua Arah

9.11.1 Apa yang berubah dengan dua tautan

Setiap galat memori dari Subbab 7.11 (kebocoran, pemakaian sesudah dibebaskan, tautan menggantung, pembebasan ganda) dapat terjadi pula pada *double linked list*, dan alat-alat dari subbab itu, Valgrind [5] dan AddressSanitizer [6], menemukannya dengan cara yang sama. Dua tautan per simpul menambahkan keluarga galat baru, yang semuanya disebabkan oleh memperbarui satu arah dan tidak memperbarui arah lainnya. Galat ini sangat licik karena arah maju, arah yang dicetak setiap uji cepat, biasanya masih tampak benar. **Gambar 9.19** memperlihatkan tiga di antaranya.



Gambar 9.19 Tiga galat tautan dua arah: (a) `insertAfter` dengan urutan langkah yang salah membuat simpul baru menunjuk dirinya sendiri, (b) tautan mundur yang terlupa membuat list terbaca berbeda di kedua arah, (c) tautan mundur yang terlupa pada penghapusan meninggalkan pointer ke memori yang sudah dibebaskan.

- **Urutan yang salah** (panel a). Bila `insertAfter` menjalankan langkah 4 (`p->next = n`) sebelum langkah 2 (`n->next = p->next`), langkah 2 menyalin nilai baru `p->next`, yaitu `n`, dan simpul baru menunjuk dirinya sendiri. Rantai maju kini berakhir dalam perulangan tak berujung di 35, dan 50 tak terjangkau ke arah maju.
- **Separuh tautan pada penyisipan** (panel b). Bila `insertAfter` melupakan langkah 3 (`p->next->prev = n`), rantai majunya 10, 20, 25, 30 dan rantai mundurnya 30, 20, 10. Tidak terjadi galat memori: setiap pointer menunjuk simpul yang hidup. Hanya Persamaan (9.1), yang diperiksa `checkLinks` dari Subbab 9.4.5, yang mengungkap kerusakannya, dan kerusakan itu akan menimbulkan galat memori sungguhan kelak, saat pertama kali sebuah operasi memercayai `prev` milik 30.
- **Separuh tautan pada penghapusan** (panel c). Bila `deleteNode` melupakan baris keduanya (`p->next->prev = p->prev`), `prev` milik 30 masih menyimpan alamat simpul 20 yang sudah dihapus. Penyusuran mundur berikutnya membaca memori yang sudah dibebaskan.

Kode 9.12 memuat dua galat terakhir dan satu kebocoran *sentinel*, dipilih lewat argumen baris perintah.

Kode 9.12 `memory_bugs.cpp`: satu jalan yang benar dan tiga galat *double linked list* yang disengaja (`halflink`, `stale`, `leak`).

```cpp
// memory_bugs.cpp -- three classic doubly-linked-list bugs, one per argument.
//   ./memory_bugs           correct program
//   ./memory_bugs halflink  insertAfter forgets the backward link (wrong list)
//   ./memory_bugs stale     deleteNode forgets the backward link (freed memory)
//   ./memory_bugs leak      destroy frees the nodes but forgets the sentinel
#include <cstdio>
#include <cstring>

struct Node {
    int data;
    Node *prev;
    Node *next;
};

struct DList {
    Node *head = nullptr;
    Node *tail = nullptr;
};

void backInsertion(DList &L, int x) {
    Node *n = new Node{x, L.tail, nullptr};
    if (L.head == nullptr) L.head = n;
    else L.tail->next = n;
    L.tail = n;
}

// assumes p is not the last node
void insertAfterWrong(DList &L, Node *p, int x) {
    Node *n = new Node{x, p, p->next};
    p->next = n;                          // BUG: p->next->prev = n is missing
    (void)L;
}

void deleteNodeWrong(DList &L, Node *p) {  // assumes p is a middle node
    p->prev->next = p->next;  // BUG: p->next->prev = p->prev is missing
    delete p;
    (void)L;
}
```

Kode 9.12 (Lanjutan) *memory_bugs.cpp: satu jalan yang benar dan tiga galat double linked list yang disengaja (halflink, stale, leak).*

```
void show(const DList &L) {
    std::printf("forward : ");
    for (Node *p = L.head; p != nullptr; p = p->next)
        std::printf("%d ", p->data);
    std::printf("\nbackward: ");
    for (Node *p = L.tail; p != nullptr; p = p->prev)
        std::printf("%d ", p->data);
    std::printf("\n");
}

void clear(DList &L) {
    while (L.head != nullptr) {
        Node *del = L.head;
        L.head = L.head->next;
        delete del;
    }
    L.tail = nullptr;
}

void sentinelListLeak() {
    Node *nil = new Node{0, nullptr, nullptr};    // a heap-allocated sentinel
    nil->prev = nil->next = nil;
    for (int x = 1; x <= 3; ++x) {  // back insertion, no special case
        Node *n = new Node{x, nil->prev, nil};
        nil->prev->next = n;
        nil->prev = n;
    }
    Node *p = nil->next;
    while (p != nil) {  // frees the three real nodes ...
        Node *del = p;
        p = p->next;
        delete del;
    }
}  // BUG: ... but never the sentinel

int main(int argc, char *argv[]) {
    const char *mode = argc > 1 ? argv[1] : "ok";
    if (std::strcmp(mode, "leak") == 0) {
        sentinelListLeak();
        std::printf("done (leak)\n");
        return 0;
    }
    DList L;
    for (int x = 10; x <= 30; x += 10) backInsertion(L, x);
    if (std::strcmp(mode, "halflink") == 0) {
        insertAfterWrong(L, L.head->next, 25);        // between 20 and 30
```

Kode 9.12 (Lanjutan) *memory_bugs.cpp: satu jalan yang benar dan tiga galat double linked list yang disengaja (halflink, stale, leak).*

```
        show(L);  // forward has 25, backward does not
    } else if (std::strcmp(mode, "stale") == 0) {
        deleteNodeWrong(L, L.head->next);             // delete 20
        show(L);  // 30->prev still points at 20
    } else {
        show(L);
    }
    clear(L);
    std::printf("done (%s)\n", mode);
    return 0;
}
```

9.11.2 Apa kata alat-alatnya

Jalan `halflink`, di bawah Valgrind maupun AddressSanitizer, hanya mencetak ini:

```
forward : 10 20 25 30
backward: 30 20 10
done (halflink)
```

Kedua alat tidak melaporkan apa pun, karena tidak terjadi apa pun yang terlarang: program hanya membaca dan menulis memori miliknya. Ini batas penting untuk dipahami. Pemeriksa memori menemukan galat *memori*; invarian yang rusak adalah galat *logika*, dan hanya pemeriksaan invarian, atau uji yang menyusuri list ke dua arah, yang menemukannya. Itulah sebabnya program-program dalam bab ini mencetak setiap list ke dua arah.

Jalan `stale` adalah galat memori sungguhan, dan program mencetak keluaran yang salah tanpa mogok: baris mundur masih memperlihatkan 20, nilai yang sudah dihapus. Valgrind melaporkannya (kompilasi dengan `-g -O0`, seperti di Subbab 7.11.2):

```
==22996== Memcheck, a memory error detector
==22996== Command: ./memory_bugs stale
==22996==
==22996== Invalid read of size 4
==22996==    at 0x10938F: show(DList const&) (memory_bugs.cpp:46)
==22996==    by 0x109638: main (memory_bugs.cpp:89)
==22996==  Address 0x4e260e0 is 0 bytes inside a block of size 24 free'd
==22996==    at 0x484A61D: operator delete(void*, unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==22996==    by 0x109305: deleteNodeWrong(DList&, Node*) (memory_bugs.cpp:36)
==22996==    by 0x10962C: main (memory_bugs.cpp:88)
==22996==  Block was alloc'd at
==22996==    at 0x4846FA3: operator new(unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==22996==    by 0x109225: backInsertion(DList&, int) (memory_bugs.cpp:21)
==22996==    by 0x1095AD: main (memory_bugs.cpp:83)
==22996==
==22996== Invalid read of size 8
==22996==    at 0x1093AB: show(DList const&) (memory_bugs.cpp:45)
==22996==    by 0x109638: main (memory_bugs.cpp:89)
==22996==  Address 0x4e260e8 is 8 bytes inside a block of size 24 free'd
==22996==    at 0x484A61D: operator delete(void*, unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==22996==    by 0x109305: deleteNodeWrong(DList&, Node*) (memory_bugs.cpp:36)
==22996==    by 0x10962C: main (memory_bugs.cpp:88)
==22996==  Block was alloc'd at
==22996==    at 0x4846FA3: operator new(unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==22996==    by 0x109225: backInsertion(DList&, int) (memory_bugs.cpp:21)
==22996==    by 0x1095AD: main (memory_bugs.cpp:83)
==22996==
forward : 10 30
backward: 30 20 10
done (stale)
==22996==
==22996== HEAP SUMMARY:
==22996==     in use at exit: 0 bytes in 0 blocks
==22996==   total heap usage: 5 allocs, 5 frees, 77,896 bytes allocated
==22996==
==22996== All heap blocks were freed -- no leaks are possible
==22996==
==22996== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Laporannya berbentuk tiga bagian yang sudah kita kenal di Bab 7. Pembacaan tidak sah yang pertama terjadi di `show` pada baris 46, badan perulangan mundur, yang dipanggil dari `main` pada

baris 89. Bloknya telah dibebaskan oleh `deleteNodeWrong` pada baris 36, yang dipanggil dari baris 88. Blok itu dialokasikan oleh `backInsertion` pada baris 21. Galat pertama adalah pembacaan berukuran 4 pada posisi 0 dari blok 24 bita, medan `data` milik simpul 20 yang sudah dibebaskan; galat kedua, pada baris 45, adalah pembacaan berukuran 8 pada posisi 8, medan `prev`-nya, yang diikuti langkah perulangan `p = p->prev` untuk mencapai 10. Posisi 8 adalah persis tempat **Gambar 9.2** menaruh `prev`. AddressSanitizer melaporkan galat yang sama pada akses buruk pertamanya lalu menghentikan program:

```
=================================================================
==23023==ERROR: AddressSanitizer: heap-use-after-free on address 0x503000000070 at pc
0x55a6ee459b27 bp 0x7ffeb6f481d0 sp 0x7ffeb6f481c0
READ of size 4 at 0x503000000070 thread T0
    #0 0x55a6ee459b26 in show(DList const&) memory_bugs.cpp:46
    #1 0x55a6ee45a445 in main memory_bugs.cpp:89

0x503000000070 is located 0 bytes inside of 24-byte region [0x503000000070,0x503000000088)
freed by thread T0 here:
    #0 0x7f86376ff5e8 in operator delete(void*, unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:164
    #1 0x55a6ee45987b in deleteNodeWrong(DList&, Node*) memory_bugs.cpp:36
    #2 0x55a6ee45a43d in main memory_bugs.cpp:88

previously allocated by thread T0 here:
    #0 0x7f86376fe548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
    #1 0x55a6ee459389 in backInsertion(DList&, int) memory_bugs.cpp:21
    #2 0x55a6ee45a37f in main memory_bugs.cpp:83

SUMMARY: AddressSanitizer: heap-use-after-free memory_bugs.cpp:46 in show(DList const&)
```

Terakhir, jalan `leak` membebaskan ketiga simpul nyata dari sebuah list ber-*sentinel* tetapi melupakan *sentinel* yang dialokasikan di *heap*:

```
==22999== Memcheck, a memory error detector
==22999== Command: ./memory_bugs leak
==22999==
done (leak)
==22999==
==22999== HEAP SUMMARY:
==22999==     in use at exit: 24 bytes in 1 blocks
==22999==   total heap usage: 6 allocs, 5 frees, 77,920 bytes allocated
==22999==
==22999== 24 bytes in 1 blocks are definitely lost in loss record 1 of 1
==22999==    at 0x4846FA3: operator new(unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==22999==    by 0x10943C: sentinelListLeak() (memory_bugs.cpp:60)
==22999==    by 0x10956A: main (memory_bugs.cpp:78)
==22999==
==22999== LEAK SUMMARY:
==22999==    definitely lost: 24 bytes in 1 blocks
==22999==    indirectly lost: 0 bytes in 0 blocks
==22999==      possibly lost: 0 bytes in 0 blocks
==22999==    still reachable: 0 bytes in 0 blocks
==22999==         suppressed: 0 bytes in 0 blocks
==22999==
==22999== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Tepat satu blok 24 bita, satu simpul, *definitely lost* (pasti hilang), dialokasikan di `sentinelListLeak` pada baris 60. Bandingkan dengan list yang bocor di Subbab 7.11.2, yang memperlihatkan satu kebocoran langsung dan dua kebocoran tak langsung: di sini ketiga simpul nyata dilepaskan dengan benar, dan hanya *sentinel*, yang alamatnya tidak dipegang siapa pun selain variabel lokal `nil`, yang tertinggal. LeakSanitizer mengatakan hal yang sama:

```
=================================================================
==23028==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 24 byte(s) in 1 object(s) allocated from:
    #0 0x7f33f48fe548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
    #1 0x559d07462d5d in sentinelListLeak() memory_bugs.cpp:60
    #2 0x559d074632af in main memory_bugs.cpp:78

SUMMARY: AddressSanitizer: 24 byte(s) leaked in 1 allocation(s).
```

Dan sertifikat untuk program yang benar: **Kode 9.5** di bawah Valgrind diakhiri dengan

```
==23008== HEAP SUMMARY:
==23008==     in use at exit: 0 bytes in 0 blocks
==23008==   total heap usage: 11 allocs, 11 frees, 78,040 bytes allocated
==23008==
==23008== All heap blocks were freed -- no leaks are possible
==23008==
==23008== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Sebelas alokasi: sembilan simpul yang dibuat oleh jalannya program dan dua blok milik pustaka *run-time*, semuanya dibebaskan. Setiap program C++ dalam bab ini, kecuali program pengukur waktu, dijalankan di bawah Valgrind dengan `--leak-check=full` dan di bawah `-fsanitize=address,undefined`, dan masing-masing menghasilkan keluaran normalnya tanpa laporan apa pun.

9.11.3 Daftar periksa untuk tautan dua arah

1. **Empat saat menyisip, dua saat menghapus.** Penyisipan di antara dua simpul mengubah empat pointer; penghapusan mengubah dua. Hitunglah di setiap fungsi yang Anda tulis.
2. **Simpul baru dulu, tetangga kemudian, tautan penunjuk jalan paling akhir.** Pada `insertAfter(p)`, pasang `n->prev` dan `n->next`, lalu `p->next->prev`, dan `p->next` paling akhir (Subbab 9.4.2).
3. **Setiap tautan maju yang ditulis punya kembaran mundur.** Sesudah menulis `a->next = b`, tanyakan di mana `b->prev = a` ditulis. Sesudah melepas `p` ke arah maju, tanyakan di mana ia dilepas ke arah mundur.
4. **Kedua pointer luar.** Tanpa *sentinel*, penyisipan pertama ke list kosong harus mengisi `head` dan `tail`, dan menghapus satu-satunya simpul harus mengosongkan keduanya.
5. **Lepaskan *sentinel* atau simpul kepala.** Bila dialokasikan dengan `new`, ia harus di-`delete` sesudah simpul-simpul nyata; bila ia anggota objek list, penyalinan objek harus dilarang.
6. **Cetak ke dua arah, periksa invarian, jalankan alatnya.** List dua arah yang hanya diuji ke arah maju baru separuh teruji.

9.12 Double Linked List dalam Pustaka Standar

9.12.1 C++: `std::list`

`std::list<T>` adalah *double linked list* milik pustaka standar C++ [7]. Ia mendukung penyisipan dan penghapusan dalam waktu konstan di mana pun, bila diberikan iterator, dan “adding, removing and moving the elements within the list or across several lists does not invalidate the iterators or references” (menambah, membuang, dan memindahkan elemen di dalam list atau antarlist tidak membatalkan iterator maupun referensi) [8]. Implementasi GNU-nya persis

struktur Subbab 9.7: *double linked list* sirkular yang satu-satunya anggota datanya adalah simpul *header* yang berperan sebagai *sentinel*. Kode sumbernya menjelaskan bahwa list itu "actually circular" (sebenarnya sirkular), bahwa `end()` adalah simpul *header* itu sendiri, dan bahwa list kosong "when this member iterator's next/previous pointers refer to itself" (ketika pointer next/previous milik iterator anggota ini merujuk dirinya sendiri); *header* juga menyimpan banyaknya elemen, "so that `std::list::size()` is fast" (supaya `std::list::size()` cepat) [9]. **Kode 9.13** menjalankan operasi-operasi yang dibuat murah oleh *double linked list*.

Kode 9.13 `std_dll.cpp`: `std::list` disusuri ke dua arah, dengan penyisipan dan penghapusan $O(1)$ pada iterator, iterator yang stabil, dan `splice`.

```
// std_dll.cpp -- std::list, the C++ standard library's doubly linked list:
// both directions, O(1) insert/erase at an iterator, splice, stable iterators.
#include <iostream>
#include <iterator>
#include <list>
#include <string>

template <typename Seq>
void print(const char *label, const Seq &s) {
    std::cout << label;
    for (const auto &x : s) std::cout << ' ' << x;
    std::cout << '\n';
}

int main() {
    std::list<int> a = {10, 20, 30, 40};
    print("start:              ", a);
    a.push_front(5);                                  // O(1) at both ends
    a.push_back(50);
    print("push_front/back:    ", a);

    auto it = std::next(a.begin(), 3);  // walk to the node holding 30: O(k)
    auto kept = it;  // a second iterator to the same node
    a.insert(it, 25);                                 // O(1): new node before 30
    a.erase(std::prev(it, 2));  // O(1): remove the node holding 20
    print("insert 25, erase 20:", a);
    std::cout << "iterator to 30 still valid: " << *kept << '\n';

    std::cout << "backward:           ";
    // follows prev
    for (auto r = a.rbegin(); r != a.rend(); ++r) std::cout << ' ' << *r;
    std::cout << '\n';

    std::list<int> b = {7, 8, 9};
    // move the node holding 8: O(1)
    a.splice(a.begin(), b, std::next(b.begin()));
    print("after splice, a:    ", a);
    print("after splice, b:    ", b);
    a.splice(a.end(), b);  // move all of b to the back of a
    print("splice all of b:    ", a);
    std::cout << "b is now empty: " << (b.empty() ? "yes" : "no")
              << ", a.size() = " << a.size() << '\n';
```

Kode 9.13 (Lanjutan) *std_dll.cpp: std::list disusuri ke dua arah, dengan penyisipan dan penghapusan O(1) pada iterator, iterator yang stabil, dan splice.*

```
    std::list<std::string> recent = {"Filosofi Kopi", "Bumi Manusia",
                                     "Laskar Pelangi"};
    // "Laskar Pelangi" is borrowed again
    auto hit = std::next(recent.begin(), 2);
    // move it to the front, no copying
    recent.splice(recent.begin(), recent, hit);
    std::cout << "move-to-front:      ";
    for (const auto &t : recent) std::cout << " [" << t << ']';
    std::cout << '\n';
    return 0;
}
```

```
start:               10 20 30 40
push_front/back:     5 10 20 30 40 50
insert 25, erase 20: 5 10 25 30 40 50
iterator to 30 still valid: 30
backward:            50 40 30 25 10 5
after splice, a:     8 5 10 25 30 40 50
after splice, b:     7 9
splice all of b:     8 5 10 25 30 40 50 7 9
b is now empty: yes, a.size() = 9
move-to-front:       [Laskar Pelangi] [Filosofi Kopi] [Bumi Manusia]
```

Tiga ciri patut diperhatikan. Pertama, `insert(it, 25)` menyisipkan *sebelum* simpul iterator itu, sesuatu yang hanya dapat dilakukan `std::forward_list` yang *single linked* dengan `insert_after` pada simpul sebelumnya (Subbab 7.12.1); `std::prev(it, 2)` berjalan mundur menyusuri tautan `prev`. Kedua, iterator `kept` masih merujuk 30 sesudah penyisipan dan penghapusan di sebelahnya, karena tidak ada simpul yang berpindah. Ketiga, `splice` memindahkan simpul dari satu list ke list lain, atau ke posisi lain di list yang sama, dengan menautkan ulang, tanpa menyalin atau mengalokasikan: memindahkan satu simpul atau seluruh list bernilai $O(1)$ (memindahkan rentang di antara dua list yang berbeda linear terhadap panjang rentang, karena banyaknya elemen harus diperbarui). Baris keluaran terakhir adalah langkah pindah-ke-depan dari panel LRU, dikerjakan pustaka dalam satu pemanggilan.

9.12.2 Java: `LinkedList` dan `LinkedHashMap`

`java.util.LinkedList<E>` milik Java adalah "doubly-linked list implementation of the `List` and `Deque` interfaces" (implementasi *double linked list* dari antarmuka `List` dan `Deque`) [4]. `ListIterator`-nya dapat bergerak ke dua arah serta menyisipkan atau membuang di posisinya dalam $O(1)$, dan `descendingIterator()` menyusuri list dari elemen terakhir sampai yang pertama. Java juga menyediakan pola LRU dari Subbab 9.9 siap pakai: `LinkedHashMap` "maintains a doubly-linked list running through all of its entries" (memelihara *double linked list* yang melintasi semua entrinya), dan sebuah konstruktor khusus mengurutkan list itu menurut akses alih-alih menurut penyisipan, yang "is well-suited to building LRU caches" (sangat cocok untuk membangun *cache* LRU); menimpa `removeEldestEntry` menetapkan kapasitasnya [10]. **Kode 9.14** memakai keduanya.

Kode 9.14 `LibraryDll.java`: `LinkedList` disunting lewat `ListIterator` dan disusuri mundur, serta `LinkedHashMap` dalam urutan akses yang dipakai sebagai daftar LRU berkapasitas 3.

```java
// LibraryDll.java -- Java's doubly linked structures: LinkedList walked in both
// directions with a ListIterator, and LinkedHashMap in access order as an LRU
// list.
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.Map;

public class LibraryDll {
    public static void main(String[] args) {
        LinkedList<Integer> a = new LinkedList<>(List.of(10, 20, 30, 40));
        a.addFirst(5);                                  // O(1) at both ends
        a.addLast(50);
        System.out.println("addFirst/addLast:    " + a);

        // a cursor between two nodes
        ListIterator<Integer> it = a.listIterator();
        while (it.next() != 30) { }                     // walk forward to 30
        it.previous();                                  // step back over 30
        it.add(25);  // O(1): link a node before 30
        it.previous();                                  // back over 25 ...
        it.previous();                                  // ... and over 20
        it.remove();                                    // O(1): unlink 20
        System.out.println("add 25, remove 20:   " + a);

        StringBuilder sb = new StringBuilder("backward:           ");
        Iterator<Integer> d = a.descendingIterator();     // follows prev
        while (d.hasNext()) sb.append(' ').append(d.next());
        System.out.println(sb);
        // Java 21: no copy
        System.out.println("reversed() view:     " + a.reversed());

        // access order = true: every get() moves the entry to the most-recent
        // end
        LinkedHashMap<String, String> recent =
                new LinkedHashMap<>(16, 0.75f, true) {
            @Override
            protected boolean removeEldestEntry(
                    Map.Entry<String, String> eldest) {
                return size() > 3;  // keep only the three most recent
            }
        };
        recent.put("B1001", "Laskar Pelangi");
        recent.put("B1002", "Bumi Manusia");
```

Kode 9.14 (Lanjutan) *LibraryDll.java: LinkedList disunting lewat ListIterator dan disusuri mundur, serta LinkedHashMap dalam urutan akses yang dipakai sebagai daftar LRU berkapasitas 3.*

```java
        recent.put("B1003", "Filosofi Kopi");
        recent.get("B1001");  // borrowed again: now the newest
        recent.put("B1004", "Cantik Itu Luka");  // evicts the eldest, B1002
        System.out.println("LRU, oldest first:   " + recent.keySet());
    }
}
```

```
addFirst/addLast:    [5, 10, 20, 30, 40, 50]
add 25, remove 20:   [5, 10, 25, 30, 40, 50]
backward:            50 40 30 25 10 5
reversed() view:     [50, 40, 30, 25, 10, 5]
LRU, oldest first:   [B1003, B1001, B1004]
```

Operasi list menghasilkan list yang sama dengan **Kode 9.13**. Tampilan `reversed()`, yang ditambahkan antarmuka `SequencedCollection` sejak Java 21, menyusuri simpul-simpul yang

sama secara mundur tanpa menyalinnya. Pada *map*, `get("B1001")` menjadikan B1001 entri yang paling baru dipakai, sehingga menambahkan B1004 mengusir B1002, yang tertua: *map* mencetak kuncinya dari yang tertua sampai yang terbaru.

9.12.3 Python: `collections.deque`, `OrderedDict`, dan `lru_cache`

Python tidak punya kelas *linked list* serbaguna, karena alasan yang diberikan di Subbab 7.12.3, tetapi tiga bagian pustaka standarnya adalah *double linked list* di dalamnya. `collections.deque` menyimpan elemennya, menurut kode sumber CPython, dalam "a doubly-linked list of fixed length blocks" (*double linked list* berisi blok-blok berpanjang tetap) masing-masing 64 elemen, yang "assures that appends or pops never move any other data elements besides the one being appended or popped" (menjamin bahwa penambahan atau pengambilan tidak pernah memindahkan elemen data lain selain yang ditambahkan atau diambil) [11]; ia memberikan penambahan dan pengambilan $O(1)$ di kedua ujung [12]. `OrderedDict` versi Python murni menyimpan kuncinya dalam "a doubly linked list", sebuah "circular doubly linked list [that] starts and ends with a sentinel element" (*double linked list* sirkular yang dimulai dan diakhiri dengan elemen *sentinel*) [13], dan implementasi C yang dipakai CPython secara bawaan menyediakan operasi yang sama; `move_to_end` dan `popitem(last=False)` adalah persis langkah pindah-ke-depan dan usir-yang-tertua dari panel LRU. Dan dekorator `functools.lru_cache`, yang menyimpan hasil sebuah fungsi dalam *cache*, diimplementasikan dengan *double linked list* sirkular yang `root`-nya adalah *sentinel*: *cache hit* melepas entrinya lalu menautkannya lagi sebelum `root`, dan pengusiran mengambil entri sesudah `root` [14]. **Kode 9.15** memakai ketiganya.

Kode 9.15 `library_dll.py`: `collections.deque` (dengan `rotate` dan `maxlen`), `OrderedDict` yang dipakai sebagai daftar LRU, dan `functools.lru_cache`.

```python
# library_dll.py -- Python's built-in doubly linked structures:
# collections.deque (a doubly linked list of 64-slot blocks) and OrderedDict /
# lru_cache as LRU lists.
from collections import OrderedDict, deque
from functools import lru_cache


def main():
    d = deque([10, 20, 30, 40])
    d.appendleft(5)                             # O(1) at both ends
    d.append(50)
    print("appendleft/append:  ", list(d))
    d.rotate(2)                                 # the last two move to the front
    print("rotate(2):          ", list(d))
    d.rotate(-2)
    print("backward:           ", list(reversed(d)))

    recent = deque(maxlen=3)  # a bounded deque drops from the other end
    for book in ["B1001", "B1002", "B1003", "B1004"]:
        recent.appendleft(book)
    print("maxlen=3, newest first:", list(recent))

    lru = OrderedDict()  # a dict with a doubly linked order
    for k, v in [("B1001", "Laskar Pelangi"), ("B1002", "Bumi Manusia"),
                 ("B1003", "Filosofi Kopi")]:
        lru[k] = v
    lru.move_to_end("B1001")                    # borrowed again: now the newest
    lru["B1004"] = "Cantik Itu Luka"
    if len(lru) > 3:
        lru.popitem(last=False)                 # evict the oldest, B1002
    print("LRU, oldest first:  ", list(lru))

    @lru_cache(maxsize=2)
    def title_of(book_id):
        titles = {"B1001": "Laskar Pelangi", "B1002": "Bumi Manusia"}
        return titles.get(book_id, "?")

    for book_id in ["B1001", "B1002", "B1001", "B1003", "B1002"]:
        title_of(book_id)
    print("lru_cache:          ", title_of.cache_info())


if __name__ == "__main__":
    main()
```

```
appendleft/append:   [5, 10, 20, 30, 40, 50]
rotate(2):           [40, 50, 5, 10, 20, 30]
backward:            [50, 40, 30, 20, 10, 5]
maxlen=3, newest first: ['B1004', 'B1003', 'B1002']
LRU, oldest first:   ['B1003', 'B1001', 'B1004']
lru_cache:           CacheInfo(hits=1, misses=4, maxsize=2, currsize=2)
```

`rotate(2)` memindahkan dua elemen terakhir ke depan, sebuah rotasi lingkaran, jenis operasi yang wajar bagi list sirkular. *Deque* dengan `maxlen=3` diam-diam membuang satu elemen dari ujung seberang ketika elemen keempat ditambahkan, sehingga ia menjadi *buffer* "*k* butir terakhir" yang siap pakai. `OrderedDict` berakhir dengan kunci yang sama dengan `LinkedHashMap` milik Java. Untuk `lru_cache` yang menampung dua hasil, pemanggilan B1001, B1002, B1001, B1003, B1002 memberi satu *hit* (B1001 yang kedua) dan empat *miss*: B1003 mengusir B1002, yang paling lama tidak dipakai, sehingga pemanggilan terakhir, untuk B1002, meleset lagi.

Tabel 9.5 merangkum ketiga pustaka.

Tabel 9.5 Struktur *double linked* dalam pustaka standar C++, Java, dan Python.

	C++ `std::list`	Java `LinkedList`	Java `LinkedHashMap`	Python `collections.deque`	Python `OrderedDict`, `lru_cache`
Struktur	DLL sirkular dengan simpul *header* (*sentinel*)	DLL non-sirkular dengan `first`, `last`	tabel *hash* ditambah DLL yang melintasi entri	DLL berisi blok 64 slot	tabel *hash* ditambah DLL sirkular dengan *sentinel*
Kedua ujung	$O(1)$	$O(1)$	entri tertua dan terbaru, $O(1)$	$O(1)$	entri tertua dan terbaru, $O(1)$
Sisip atau hapus di suatu posisi	$O(1)$ lewat iterator	$O(1)$ lewat `ListIterator`	$O(1)$ berdasarkan kunci	$O(n)$	$O(1)$ berdasarkan kunci
Penyusuran mundur	`rbegin()`, `rend()`	`descendingIterator()`, `reversed()`	`reversed()` (Java 21)	`reversed()`	`reversed()`
Memindahkan simpul tanpa menyalin	`splice`	tidak	otomatis dalam urutan akses	`rotate` (seluruh *deque*)	`move_to_end`

Rancangan yang sama muncul jauh di luar pustaka standar. List serbaguna milik kernel Linux, `struct list_head`, adalah *double linked list* sirkular yang kepala list-nya berperan sebagai *sentinel*; dokumentasi kernel mencatat bahwa "to get from the head node to the tail, we can just travel one edge backwards" (untuk pergi dari simpul kepala ke ekor, kita cukup menempuh satu sisi ke belakang), dan juga memperingatkan bahwa, karena lokalitas datanya buruk, *linked list* adalah pilihan yang buruk bila kinerja penting [15]. Teknik *dancing links* dari Knuth memanfaatkan kenyataan bahwa simpul yang dilepas dari *double linked list* masih mengingat kedua tetangganya, sehingga `x.prev.next ← x; x.next.prev ← x` mengembalikannya tepat ke tempatnya semula; pencarian runut-balik (*backtracking*) memakai ini untuk membatalkan penghapusan dalam $O(1)$ [16]. Soal 9.12 meminta Anda memverifikasinya.

9.13 Profitina dalam Praktik

Profitina adalah platform visibilitas AI (GEO) yang mengukur seberapa siap situs web sebuah merek untuk dikutip oleh mesin penjawab AI, dan membantu memperbaiki apa yang ditemukannya [17]. Servernya adalah aplikasi Python yang dibangun di atas FastAPI, dan beberapa fiturnya membutuhkan persis operasi-operasi yang dibuat murah oleh bab ini. Kita melukiskan bentuk setiap masalah dan kelas struktur yang cocok untuknya.

- **Menjelajahi riwayat ke dua arah.** Dasbor analitik historis menampilkan skor sebuah merek minggu demi minggu. Pengguna yang membuka minggu terbaru lalu mundur ke minggu sebelum sebuah peringatan, lalu maju lagi untuk membandingkan, sedang menyusuri barisan *double linked* dengan kursor: setiap langkah adalah `cur = cur->prev` atau `cur = cur->next`, $O(1)$ berapa pun panjang riwayatnya. Subbab 7.13 mencatat bahwa riwayat yang hanya maju dari Bab 7 tidak dapat mendukung "mundur satu minggu" tanpa berjalan dari awal; pointer `prev`-lah yang membuat kedua tombol sama murahnya.
- **Menyimpan butir-butir terbaru.** Umpan komunitas *real-time* dan daftar notifikasi dasbor menyimpan butir-butir terbaru dan membuang yang tertua. Dalam Python ini adalah `collections.deque` dengan `maxlen`, yang blok-bloknya bertaut ganda ([11], Subbab 9.12.3).

- **Menyimpan hasil yang mahal.** Sebuah pemindaian membaca ulang halaman, dan mesin AI lokal menjawab pertanyaan; keduanya mahal, dan permintaan berulang sering terjadi. *Cache* berbatas dengan kebijakan LRU dari Subbab 9.9, seperti `functools.lru_cache` atau `OrderedDict` yang dipakai seperti pada **Kode 9.15**, menyimpan hasil yang baru dipakai dan mengusir yang paling lama tidak dipakai lebih dulu. Di dalamnya ada list ber-*sentinel* ditambah *hash map* seperti pada **Gambar 9.17**.

Studi kasus yang dapat dijalankan adalah Profitina Laundry, usaha binatu kecil yang mencatat berapa banyak pesanan yang diterimanya setiap minggu. **Kode 9.16** menyimpan total mingguan dalam *double linked list* dengan kursor, dibuka pada minggu terbaru, dan mengikuti serangkaian klik Back dan Forward, sambil melaporkan untuk setiap minggu perubahan dari minggu sebelumnya, yang dibacanya lewat `prev` dalam $O(1)$.

Kode 9.16 `laundry_weeks.cpp`: total pesanan mingguan Profitina Laundry dalam *double linked list*, dijelajahi satu minggu mundur atau maju dari minggu mana pun.

```
// laundry_weeks.cpp -- Profitina Laundry: browsing weekly order totals one week
// back or forward from any week, with a doubly linked list and a cursor.
#include <cstdio>
#include <string>

struct Week {
    std::string label;                  // e.g. "2026-W36"
    int orders;                         // orders taken that week
    Week *prev;
    Week *next;
};

struct History {
    Week *first = nullptr;
    Week *latest = nullptr;             // the newest week (the tail)
};

void addWeek(History &h, const std::string &label, int orders) {
    Week *w = new Week{label, orders, h.latest, nullptr};
    if (h.latest == nullptr) h.first = w;
    else h.latest->next = w;
    h.latest = w;  // O(1): weeks only ever arrive at the back
}

void report(const Week *w) {
    std::printf("  %s: %3d orders", w->label.c_str(), w->orders);
    if (w->prev != nullptr) {
        int d = w->orders - w->prev->orders;  // compare with the week before
        std::printf("  (%+d vs %s)", d, w->prev->label.c_str());
    }
    std::printf("\n");
}

void destroy(History &h) {
    while (h.first != nullptr) {
        Week *del = h.first;
        h.first = h.first->next;
        delete del;
    }
    h.latest = nullptr;
}
```

Kode 9.16 (Lanjutan) *laundry_weeks.cpp: total pesanan mingguan Profitina Laundry dalam double linked list, dijelajahi satu minggu mundur atau maju dari minggu mana pun.*

```
int main() {
    History h;
    const char *labels[] = {"2026-W32", "2026-W33", "2026-W34", "2026-W35",
                            "2026-W36", "2026-W37", "2026-W38"};
    const int orders[] = {118, 124, 97, 131, 140, 126, 152};
    for (int i = 0; i < 7; ++i) addWeek(h, labels[i], orders[i]);

    Week *cur = h.latest;  // the dashboard opens on the newest week
    std::printf("open:\n");
    report(cur);
    // b = one week back, f = one week forward
    const char *clicks = "bbbfbbbbb";
    for (const char *c = clicks; *c != '\0'; ++c) {
        // one step either way: O(1)
        Week *to = (*c == 'b') ? cur->prev : cur->next;
        if (to == nullptr) {
            std::printf("%s: no more weeks, stays on %s\n",
                        *c == 'b' ? "back" : "forward", cur->label.c_str());
            continue;
        }
        cur = to;
        std::printf("%s:\n", *c == 'b' ? "back" : "forward");
        report(cur);
    }
    destroy(h);
    return 0;
}
```

```
open:
  2026-W38: 152 orders  (+26 vs 2026-W37)
back:
  2026-W37: 126 orders  (-14 vs 2026-W36)
back:
  2026-W36: 140 orders  (+9 vs 2026-W35)
back:
  2026-W35: 131 orders  (+34 vs 2026-W34)
forward:
  2026-W36: 140 orders  (+9 vs 2026-W35)
back:
  2026-W35: 131 orders  (+34 vs 2026-W34)
back:
  2026-W34:  97 orders  (-27 vs 2026-W33)
back:
  2026-W33: 124 orders  (+6 vs 2026-W32)
back:
  2026-W32: 118 orders
back: no more weeks, stays on 2026-W32
```

Minggu baru hanya pernah ditambahkan di belakang, sehingga pointer *tail* membuat `addWeek` bernilai $O(1)$; setiap klik adalah satu langkah pointer; dan perbandingan dengan minggu sebelumnya adalah satu langkah lagi. Ketika kursor tiba di minggu tertua, `prev` bernilai null dan tombol Back tidak melakukan apa-apa. `LaundryWeeks.java` dan `laundry_weeks.py` mencetak laporan yang sama.

9.14 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Memperbarui satu arah saja. Setiap `a->next = b` membutuhkan `b->prev = a`-nya. List yang tercetak benar ke arah maju mungkin rusak ke arah mundur (**Gambar 9.19**, panel b). Cetaklah

setiap list uji ke dua arah dan periksa Persamaan (9.1) dengan fungsi seperti `checkLinks` (Subbab 9.4.5).

Menimpa tautan penunjuk jalan. Pada `insertAfter(p)`, `p->next` adalah satu-satunya jalan ke penerus; ubahlah paling akhir. Mengisi `p->next = n` lebih dulu membuat simpul baru menunjuk dirinya sendiri (**Gambar 9.19**, panel a).

Lupa mengosongkan `prev` di depan yang baru. Sesudah `frontDeletion`, `prev` milik simpul pertama baru harus menjadi null (Subbab 9.3.5); sesudah `backDeletion`, `next` milik simpul terakhir baru harus menjadi null. Bila tidak, list berujung di memori yang sudah dibebaskan.

Lupa pointer luar yang kedua. Pada list dengan `head` dan `tail`, penyisipan ke list kosong harus mengisi keduanya, dan penghapusan satu-satunya simpul harus mengosongkan keduanya (Subbab 9.4.1).

Menguji null pada list sirkular. Tidak ada pointer yang null pada list sirkular yang tidak kosong, sehingga `while (p != nullptr)` tidak pernah selesai. Pakailah `do-while` yang berhenti ketika kembali ke titik awal, atau, dengan *sentinel*, `for (p = nil.next; p != &nil; p = p->next)`.

Menghapus *sentinel*. `LIST-DELETE(nil)` menghancurkan jangkar list. Operasi *pop* harus memeriksa kekosongan lebih dulu (seperti yang dilakukan `Deque::unlink` di **Kode 9.9**).

Membocorkan simpul kepala atau *sentinel*. Simpul kepala atau *sentinel* yang dialokasikan di *heap* harus dilepaskan sesudah simpul-simpul nyata (Subbab 9.5.2 dan 9.11.2).

Menyalin objek list yang memuat *sentinel*-nya. Tetangga *sentinel* salinan masih menunjuk *sentinel* asli. Larang penyalinan atau implementasikan simpul demi simpul.

Mengira penghapusan $O(1)$ berarti penghapusan $O(1)$ berdasarkan nilai. `deleteNode(p)` bernilai $O(1)$; menemukan `p` berdasarkan nilai tetap pencarian $\Theta(n)$. *Double linked list* membayar ketika program sudah memegang simpulnya, seperti pada panel LRU di Subbab 9.9.

Memakai *double linked list* padahal larik sudah cukup. Untuk elemen kecil dan pembacaan yang kebanyakan berurutan, memori larik yang bersebelahan menang jauh (Subbab 7.12.4), dan simpul *double linked* membelanjakan 16 bita untuk pointer (Persamaan (9.2)).

9.15 Ringkasan Bab

- ***Double linked list*** memberi setiap simpul dua pointer, `next` dan `prev`. **Invarian dua arah** (Persamaan (9.1)) menyatakan bahwa setiap tautan maju punya tautan mundur pasangannya; setiap operasi harus menjaganya.
- Simpul *double linked* memakan satu pointer lebih banyak dari simpul *single linked*: 24 bita alih-alih 16 untuk `int` pada mesin 64-bit (**Gambar 9.2**, Persamaan (9.2)).
- **Dengan `head` saja**, penyisipan dan penghapusan di depan bernilai $O(1)$, penyisipan dan penghapusan di belakang $\Theta(n)$, dan penyusuran mundur membutuhkan dua perjalanan tanpa memori tambahan (**Tabel 9.1**).
- **Dengan `head` dan `tail`**, setiap operasi ujung bernilai $O(1)$, termasuk penghapusan di belakang, karena simpul sebelum yang terakhir adalah `tail->prev` (**Pseudokode 9.6**). Menyisipkan sebelum atau sesudah, dan menghapus, simpul yang dipegang program semuanya $O(1)$ (**Pseudokode 9.7**, **Pseudokode 9.9**, **Tabel 9.2**). Penyisipan mengubah empat pointer, penghapusan dua.

- **Simpul kepala** menghapus kasus khusus di depan; list **sirkular** mencapai kedua ujung lewat satu pointer (`head->prev` adalah simpul terakhir) dan membutuhkan tanda berhenti baru untuk penyusuran.
- ***Sentinel*** pada *double linked list* sirkular menghapus setiap kasus khusus: tidak ada pointer yang pernah null dan tidak ada pointer luar yang pernah berubah, sehingga penyisipan dan penghapusan ditulis sekali, tanpa uji (**Pseudokode 9.10**, **Pseudokode 9.11**, **Tabel 9.3**). Menyimpan kunci di *sentinel* membagi dua banyaknya uji pencarian sekuensial (Persamaan (9.4)).
- ***Deque*** di atas list ber-*sentinel* mengerjakan semua operasinya dalam $O(1)$; **daftar LRU**, list ber-*sentinel* ditambah *hash map* dari kunci ke simpul, mencatat setiap akses dalam waktu harapan $O(1)$ (Persamaan (9.5)).
- Dibandingkan *single linked list*, *double linked list* mengubah penghapusan di belakang dan penghapusan di posisi yang dipegang dari $\Theta(n)$ menjadi $O(1)$; terukur, mengosongkan 20.000 simpul dari belakang menjadi sekitar 3.500 kali lebih cepat (**Gambar 9.18**). Dibandingkan larik, ia kehilangan akses acak dan lokalitas *cache* (**Tabel 9.4**).
- Pemeriksa memori menemukan pemakaian memori terbebaskan yang disebabkan tautan yang separuh diperbarui, tetapi tidak menemukan invarian rusak yang tetap berada di memori hidup; cetaklah ke dua arah dan periksa invariannya.
- **Pustaka**: `std::list` di C++ adalah *double linked list* sirkular dengan *sentinel*; Java punya `LinkedList` dan `LinkedHashMap` yang siap untuk LRU; `collections.deque`, `OrderedDict`, dan `functools.lru_cache` di Python bertaut ganda di dalamnya (**Tabel 9.5**).

9.16 Soal Latihan

9.1 Mulai dari *double linked list* kosong dengan *head* dan *tail*, lakukan `frontInsertion(5)`, `backInsertion(8)`, `frontInsertion(2)`, `insertAfter(p, 6)` dengan `p` simpul yang berisi 5, `backDeletion()`, `deleteNode(p)` untuk `p` yang sama, dan `frontInsertion(9)`. Tuliskan list sesudah setiap operasi, dibaca maju dan mundur, dan sebutkan operasi mana yang mengubah `head` atau `tail`.

9.2 Untuk *double linked list* berisi n simpul, berikan waktu jalan dari: (a) penghapusan di belakang dengan `head` saja; (b) penghapusan di belakang dengan `head` dan `tail`; (c) menghapus simpul yang pointernya dipegang program, dibandingkan dengan *single linked list* dengan *head* dan *tail*; (d) mencetak list secara mundur dengan `head` saja (berikan banyaknya langkah pointer secara tepat); (e) mencapai simpul pada indeks $k = n/2$ bila `head` dan `tail` sama-sama tersedia (gunakan Persamaan (9.6)). Beri alasan setiap jawaban dalam satu kalimat.

9.3 Tulislah `void reverse(DList &L)`, yang membalik *double linked list* dengan *head* dan *tail* di tempat, tanpa membuat atau menghapus simpul, dalam satu lintasan. Apa bedanya dengan pembalikan iteratif *single linked list* di Subbab 7.6.6?

9.4 Tulislah `bool isPalindrome(const DList &L)`, yang memutuskan apakah list terbaca sama ke dua arah, dengan memori tambahan $O(1)$. Pastikan ia benar untuk list berpanjang ganjil dan genap serta untuk list kosong.

9.5 Tulislah `int deleteAll(DList &L, int key)`, yang menghapus setiap simpul berisi `key` dan mengembalikan banyaknya simpul yang dihapus. Apa yang harus disimpan perulangan sebelum menghapus sebuah simpul?

9.6 Tulislah `void concat(DList &a, DList &b)`, yang menambahkan semua simpul `b` ke ujung `a` dalam waktu $O(1)$ dan meninggalkan `b` kosong. Berapa ongkos operasi yang sama untuk dua *single linked list* dengan *head* dan *tail*, dan untuk dua larik?

9.7 Tulislah `Node *kthFromEnd(const DList &L, int k)`, yang mengembalikan simpul ke-k dari belakang ($k = 1$ adalah simpul terakhir) atau `nullptr` bila list berisi kurang dari k simpul, dalam waktu $O(k)$.

9.8 Tulislah `void moveToFront(DList &L, Node *p)`, yang memindahkan simpul `p` dari list dengan *head* dan *tail* ke depan dalam waktu $O(1)$. Tangani kasus ketika `p` sudah simpul pertama dan ketika ia simpul terakhir. Inilah langkah yang diulang panel LRU dari Subbab 9.9 pada setiap *hit*.

9.9 Pada *double linked list* sirkular dengan *sentinel*, tulislah `void rotateLeft(Node *nil, int k)` untuk $0 \le k < n$, yang menjadikan simpul ke-$(k+1)$ sebagai simpul pertama (sehingga 1, 2, 3, 4, 5, 6 yang diputar ke kiri sebanyak 2 menjadi 3, 4, 5, 6, 1, 2), dengan memindahkan *sentinel* itu sendiri. Berapa penugasan pointer yang dilakukannya, dan berapa waktu jalannya?

9.10 Pada **Pseudokode 9.7**, manakah dari keempat langkah yang boleh dipertukarkan tanpa merusak list, dan mana yang tidak? Untuk setiap pertukaran terlarang, lukiskan list rusak yang dihasilkannya.

9.11 Prosedur di bawah ini diusulkan sebagai `frontInsertion` untuk list Subbab 9.3, tanpa uji milik **Pseudokode 9.1**:

```
void frontInsertion(TNode *&head, int x) {
    TNode *n = new TNode{x, nullptr, head};
    head->prev = n;
    head = n;
}
```

Temukan masukan yang membuatnya gagal, katakan apa yang terjadi, dan jelaskan mengapa baris yang bersesuaian di **Pseudokode 9.10** tidak pernah dapat gagal dengan cara yang sama.

9.12 (*Dancing links.*) Pada *double linked list* sirkular dengan *sentinel*, simpul `x` dilepas dengan dua penugasan **Pseudokode 9.11** tetapi *tidak* dibebaskan, sehingga `x.prev` dan `x.next` masih menyimpan tetangga lamanya. Tunjukkan bahwa dua penugasan `x.prev.next ← x; x.next.prev ← x` mengembalikannya persis ke tempatnya semula. Lalu tunjukkan bahwa bila simpul `x` kemudian `y` dilepas dengan cara ini, keduanya harus dikembalikan dengan urutan terbalik, `y` lebih dulu, dan berikan list tiga simpul yang rusak bila pengembalian dilakukan dengan urutan yang sama.

9.13 *XOR linked list* menyimpan di setiap simpul satu medan $link = prev \oplus next$, yaitu *exclusive-or* bit demi bit dari kedua alamat, alih-alih dua pointer. (a) Jelaskan bagaimana penyusuran yang tahu alamat simpul sebelumnya dapat menghitung alamat simpul berikutnya. (b) Berapa memori yang dihematnya untuk n simpul pada mesin 64-bit, dibandingkan dengan Persamaan (9.2)? (c) Operasi mana dari Subbab 9.4 yang menjadi mustahil dalam $O(1)$, dan mengapa? (d) Mengapa struktur ini tidak dapat ditulis dalam Java atau Python?

9.14 Untuk setiap tugas Pustaka Ceria berikut, pilihlah antara larik, *single linked list*, dan *double linked list* (mungkin dengan *hash map*), dan beri alasan pilihan itu dengan baris-baris **Tabel 9.4**: (a) antrean ruang baca, tempat pembaca bergabung di belakang dan dilayani di depan; (b) riwayat *undo* dan *redo* suntingan katalog seorang pustakawan, tempat Undo mundur dan Redo maju dari suntingan sekarang; (c) panel "baru dipinjam" dari Subbab 9.9 untuk $k = 1.000$; (d)

daftar semua 50.000 catatan buku, terurut menurut nomor panggil dan dicari dengan pencarian biner.

9.17 Penyelesaian Soal Latihan

9.1 Sesudah setiap operasi (maju | mundur): `frontInsertion(5)`: 5 | 5, dan `head` serta `tail` sama-sama berubah, karena list tadinya kosong. `backInsertion(8)`: 5, 8 | 8, 5; `tail` berubah. `frontInsertion(2)`: 2, 5, 8 | 8, 5, 2; `head` berubah. `insertAfter(p, 6)`: 2, 5, 6, 8 | 8, 6, 5, 2; tidak ada yang berubah, karena `p` bukan simpul terakhir. `backDeletion()`: 2, 5, 6 | 6, 5, 2; `tail` mundur ke 6. `deleteNode(p)`: 2, 6 | 6, 2; tidak ada yang berubah, karena 5 simpul tengah. `frontInsertion(9)`: 9, 2, 6 | 6, 2, 9; `head` berubah.

9.2 (a) $\Theta(n)$: simpul terakhir hanya dapat dicapai dengan berjalan dari `head`, meskipun pendahulunya kemudian hanya selangkah jauhnya (Subbab 9.3.6). (b) $O(1)$: simpul terakhir baru adalah `tail->prev` (**Pseudokode 9.6**). (c) $O(1)$ untuk *double linked list* (**Pseudokode 9.9**); $\Theta(n)$ untuk *single linked list*, yang harus mencari pendahulunya dari `head`. (d) $n-1$ langkah maju ke simpul terakhir dan $n-1$ langkah mundur menyusuri `prev` (perulangan mengunjungi n simpul dan melakukan n penugasan, yang terakhir menghasilkan NIL), sekitar $2n$ seluruhnya: $\Theta(n)$. (e) Menurut Persamaan (9.6), $\min(n/2,\ n/2-1) = n/2-1$ langkah dari belakang, untuk n genap: bagian tengah adalah posisi terburuk, dan ongkosnya tetap $\Theta(n)$.

9.3 Program penyelesaian **Kode 9.17** memuat `reverse`. Membalik *double linked list* tidak mengubah simpul mana yang bertetangga, hanya arah mana yang disebut "maju", sehingga cukup menukar `prev` dan `next` di setiap simpul lalu menukar `head` dan `tail`. Perulangan harus maju ke `next` yang lama, yang sesudah penukaran tersimpan di `p->prev`; itulah sebabnya ia melangkah dengan `p = p->prev`. Ada satu lintasan dan tidak ada alokasi: waktu $\Theta(n)$ dan memori $O(1)$. Pembalikan *single linked* di Subbab 7.6.6 harus memegang tiga pointer (`prev`, `cur`, `nxt`) karena satu-satunya tautan setiap simpul harus dibalik sementara sisa list masih hanya dapat dicapai lewat tautan itu; di sini setiap simpul sudah menyimpan kedua arah.

Kode 9.17 `exercises.cpp`: penyelesaian Soal 9.3 sampai 9.9.

```cpp
// exercises.cpp -- solutions to the programming exercises of Chapter 9.
#include <cstdio>

struct Node {
    int data;
    Node *prev;
    Node *next;
};

struct DList {
    Node *head = nullptr;
    Node *tail = nullptr;
};

void backInsertion(DList &L, int x) {
    Node *n = new Node{x, L.tail, nullptr};
    if (L.head == nullptr) L.head = n;
    else L.tail->next = n;
    L.tail = n;
}

DList fromArray(const int *a, int n) {
    DList L;
    for (int i = 0; i < n; ++i) backInsertion(L, a[i]);
    return L;
}

void show(const char *label, const DList &L) {
    std::printf("%-22s", label);
    for (Node *p = L.head; p != nullptr; p = p->next)
        std::printf("%d ", p->data);
    std::printf("| back: ");
    for (Node *p = L.tail; p != nullptr; p = p->prev)
        std::printf("%d ", p->data);
    std::printf("\n");
}

void clear(DList &L) {
    while (L.head != nullptr) {
        Node *del = L.head;
        L.head = L.head->next;
        delete del;
    }
    L.tail = nullptr;
}
```

Kode 9.17 (Lanjutan) *exercises.cpp: penyelesaian Soal 9.3 sampai 9.9.*

```
// Exercise 9.3: reverse in place by swapping every node's two links.
void reverse(DList &L) {
    // p->prev is the OLD next
    for (Node *p = L.head; p != nullptr; p = p->prev) {
        Node *t = p->next;
        p->next = p->prev;
        p->prev = t;
    }
    Node *t = L.head;
    L.head = L.tail;
    L.tail = t;
}

// Exercise 9.4: does the list read the same in both directions?
bool isPalindrome(const DList &L) {
    Node *a = L.head, *b = L.tail;
    // stop when they meet or cross
    while (a != b && a != nullptr && a->prev != b) {
        if (a->data != b->data) return false;
        a = a->next;
        b = b->prev;
    }
    return true;
}

// Exercise 9.5: delete every node holding key; return how many were deleted.
int deleteAll(DList &L, int key) {
    int count = 0;
    Node *p = L.head;
    while (p != nullptr) {
        Node *nxt = p->next;  // save the way on before p may disappear
        if (p->data == key) {
            if (p->prev) p->prev->next = p->next; else L.head = p->next;
            if (p->next) p->next->prev = p->prev; else L.tail = p->prev;
            delete p;
            ++count;
        }
        p = nxt;
    }
    return count;
}
```

Kode 9.17 (Lanjutan) *exercises.cpp: penyelesaian Soal 9.3 sampai 9.9.*

```
// Exercise 9.6: append all of b to a in O(1); b becomes empty.
void concat(DList &a, DList &b) {
    if (b.head == nullptr) return;
    if (a.head == nullptr) {
        a = b;
    } else {
        a.tail->next = b.head;
        b.head->prev = a.tail;
        a.tail = b.tail;
    }
    b.head = b.tail = nullptr;
}

// Exercise 9.7: the k-th node from the back (k = 1 is the last), or nullptr.
Node *kthFromEnd(const DList &L, int k) {
    Node *p = L.tail;
    while (p != nullptr && k > 1) {
        p = p->prev;
        --k;
    }
    return (k >= 1) ? p : nullptr;
}

// Exercise 9.8: move node p to the front in O(1) (the step an LRU list
// repeats).
void moveToFront(DList &L, Node *p) {
    if (p == L.head) return;
    p->prev->next = p->next;              // unlink p (it has a predecessor)
    if (p->next) p->next->prev = p->prev; else L.tail = p->prev;
    p->prev = nullptr;                    // relink p before the old first node
    p->next = L.head;
    L.head->prev = p;
    L.head = p;
}

// Exercise 9.9: rotate a circular list with a sentinel left by k (0 <= k < n)
// by moving the sentinel itself: O(k) steps, 6 pointer assignments.
void rotateLeft(Node *nil, int k) {
    if (k == 0 || nil->next == nil) return;
    Node *q = nil;
    for (int i = 0; i < k; ++i) q = q->next;    // q = the k-th real node
    nil->prev->next = nil->next;          // take the sentinel out of the ring
    nil->next->prev = nil->prev;
    nil->prev = q;                        // and put it back right after q
    nil->next = q->next;
    q->next->prev = nil;
```

Kode 9.17 (Lanjutan) *exercises.cpp: penyelesaian Soal 9.3 sampai 9.9.*

```
    q->next = nil;
}

int main() {
    int a[] = {1, 2, 3, 4, 5};
    DList L = fromArray(a, 5);
    reverse(L);
    show("reverse(1..5):", L);

    int p1[] = {3, 8, 1, 8, 3}, p2[] = {3, 8, 8, 3}, p3[] = {3, 8, 1};
    DList P1 = fromArray(p1, 5), P2 = fromArray(p2, 4);
    DList P3 = fromArray(p3, 3), E;
    std::printf("isPalindrome: 38183 %s, 3883 %s, 381 %s, empty %s\n",
                isPalindrome(P1) ? "yes" : "no",
                isPalindrome(P2) ? "yes" : "no",
                isPalindrome(P3) ? "yes" : "no",
                isPalindrome(E) ? "yes" : "no");
    clear(P1); clear(P2); clear(P3);

    int d[] = {7, 2, 7, 7, 5, 7};
    DList D = fromArray(d, 6);
    int removed = deleteAll(D, 7);
    std::printf("deleteAll(7) removed %d\n", removed);
    show("after deleteAll:", D);

    int c[] = {6, 7, 8};
    DList B = fromArray(c, 3);
    concat(L, B);
    show("concat:", L);
    std::printf("b is empty: %s\n", B.head == nullptr ? "yes" : "no");
    Node *k2 = kthFromEnd(L, 2);
    std::printf("kthFromEnd(2) = %d, kthFromEnd(9) is %s\n", k2->data,
                kthFromEnd(L, 9) ? "a node" : "nullptr");
    moveToFront(L, kthFromEnd(L, 4));     // the node holding 1
    show("moveToFront(1):", L);
    moveToFront(L, L.tail);               // the node holding 8
    show("moveToFront(8):", L);
    clear(L);
    clear(D);
```

Kode 9.17 (Lanjutan) *exercises.cpp: penyelesaian Soal 9.3 sampai 9.9.*

```
    // a circular list with a sentinel on the stack
    Node nil{0, nullptr, nullptr};
    nil.prev = nil.next = &nil;
    for (int x = 1; x <= 6; ++x) {
        Node *n = new Node{x, nil.prev, &nil};
        nil.prev->next = n;
        nil.prev = n;
    }
    rotateLeft(&nil, 2);
    std::printf("%-22s", "rotateLeft(2):");
    for (Node *p = nil.next; p != &nil; p = p->next)
        std::printf("%d ", p->data);
    std::printf("| back: ");
    for (Node *p = nil.prev; p != &nil; p = p->prev)
        std::printf("%d ", p->data);
    std::printf("\n");
    while (nil.next != &nil) {
        Node *del = nil.next;
        nil.next = del->next;
        delete del;
    }
    return 0;
}
```

```
reverse(1..5):         5 4 3 2 1 | back: 1 2 3 4 5
isPalindrome: 38183 yes, 3883 yes, 381 no, empty yes
deleteAll(7) removed 4
after deleteAll:       2 5 | back: 5 2
concat:                5 4 3 2 1 6 7 8 | back: 8 7 6 1 2 3 4 5
b is empty: yes
kthFromEnd(2) = 7, kthFromEnd(9) is nullptr
moveToFront(1):        1 5 4 3 2 6 7 8 | back: 8 7 6 2 3 4 5 1
moveToFront(8):        8 1 5 4 3 2 6 7 | back: 7 6 2 3 4 5 1 8
rotateLeft(2):         3 4 5 6 1 2 | back: 2 1 6 5 4 3
```

Baris keluaran pertama memperlihatkan 1, 2, 3, 4, 5 yang dibalik, dibaca ke dua arah.

9.4 `isPalindrome` menggerakkan `a` maju dari `head` dan `b` mundur dari `tail`, membandingkan datanya, sampai keduanya bertemu (panjang ganjil: `a == b` pada simpul tengah) atau bersilang (panjang genap: `a->prev == b` sesudah pasangan terakhir). List kosong langsung memberi `a == b == nullptr` dan merupakan palindrom. Fungsi ini melakukan paling banyak $\lfloor n/2 \rfloor$ perbandingan dan memakai dua pointer. Pada keluaran, 3, 8, 1, 8, 3 dan 3, 8, 8, 3 adalah palindrom dan 3, 8, 1 bukan.

9.5 Perulangan harus menyimpan `p->next` di `nxt` *sebelum* menghapus `p`, karena sesudah `delete p` medan itu tidak boleh lagi dibaca. Setiap penghapusan adalah pelepasan **Pseudokode 9.9**, dengan dua ujinya untuk ujung-ujung list. Pada 7, 2, 7, 7, 5, 7 fungsi ini menghapus empat simpul, termasuk yang pertama dan yang terakhir, dan menyisakan 2, 5; bacaan mundur 5, 2 memastikan tautannya. $\Theta(n)$.

9.6 Tautkan simpul terakhir `a` dan simpul pertama `b` ke dua arah, pindahkan `a.tail` ke `b.tail`, dan kosongkan `b`; bila `a` kosong, `a` cukup mengambil alih kedua pointer milik `b`. Empat penugasan: $O(1)$. Untuk dua *single linked list* dengan *head* dan *tail* juga $O(1)$ (`a.tail->next = b.head; a.tail = b.tail;`), karena hanya ada tautan maju. Untuk dua larik umumnya $\Theta(m+n)$, atau $\Theta(n)$ teramortisasi bila larik kedua disalin ke ujung larik dinamis yang punya kapasitas cadangan. Pada keluaran, 6, 7, 8 ditambahkan ke list yang sudah dibalik.

9.7 Mulai di `tail` dan mundur $k-1$ kali; berhenti lebih awal dengan `nullptr` bila list habis. $O(k)$ langkah, berapa pun n. Dengan `tail` pada *single linked list*, simpul ke-k dari belakang memakan

$\Theta(n)$, karena orang hanya dapat berjalan dari depan. Pada keluaran, simpul kedua dari belakang 5, 4, 3, 2, 1, 6, 7, 8 adalah 7, dan list berisi 8 simpul tidak punya simpul ke-9 dari belakang.

9.8 Bila `p` sudah yang pertama, tidak ada yang berubah. Bila tidak, lepaskan `p` (ia punya pendahulu, sehingga `p->prev->next = p->next` tidak butuh uji; bila ia tadinya simpul terakhir, `tail` mundur ke `p->prev`), lalu tautkan ia sebelum simpul pertama lama: `p->prev = nullptr`, `p->next = head`, `head->prev = p`, `head = p`. Pekerjaannya konstan. Keluaran memindahkan 1 (simpul tengah) lalu 8 (simpul terakhir) ke depan; kedua bacaan tetap konsisten.

9.9 Jalankan `q` dari `nil` sebanyak k langkah, sehingga `q` adalah simpul nyata ke-k; keluarkan *sentinel* dari cincin dengan dua penugasan; kembalikan ia sesudah `q` dengan empat. Itu enam penugasan pointer, seperti kata komentar di listing, dan waktu $O(k)$ untuk perjalanannya. Tidak ada simpul data yang berpindah, dan tautan simpul data tidak berubah kecuali milik tetangga lama dan tetangga baru *sentinel*. Memutar 1, 2, 3, 4, 5, 6 ke kiri sebanyak 2 menghasilkan 3, 4, 5, 6, 1, 2, dan bacaan mundur 2, 1, 6, 5, 4, 3 memastikan tautannya.

9.10 Langkah 1 dan 2 (medan milik simpul baru sendiri) boleh dipertukarkan satu sama lain, dan boleh dikerjakan dalam urutan mana pun relatif terhadap langkah 3, karena tidak satu pun menimpa medan yang dibaca langkah lain: langkah 3 membaca `p.next`, yang hanya ditulis langkah 4. Langkah 3 dan 4 tidak boleh dipertukarkan: bila langkah 4 (`p.next ← n`) datang sebelum langkah 3, langkah 3 akan menjalankan `n.prev ← n`, menjadikan simpul baru pendahulunya sendiri, dan penerus lama tetap menunjuk balik ke `p`, sehingga rantai mundur melompati simpul baru. Langkah 4 juga tidak boleh datang sebelum langkah 2: langkah 2 lalu mengisi `n.next ← n`, dan rantai maju berputar selamanya di simpul baru, dengan segala sesudahnya tak terjangkau (**Gambar 9.19**, panel a). Singkatnya, langkah 4 harus terakhir.

9.11 Prosedur itu gagal pada list kosong: `head` bernilai null, sehingga `head->prev = n` mendereferensi pointer null, yang merupakan perilaku tak terdefinisi dan biasanya membuat program mogok. Uji di **Pseudokode 9.1** ada tepat untuk kasus ini. Di **Pseudokode 9.10** baris yang bersesuaian adalah `p.next.prev ← n`, dan `p.next` tidak pernah NIL, karena pada list sirkular dengan *sentinel* setiap pointer menyimpan alamat sebuah simpul; pada list kosong `p.next` adalah *sentinel* itu sendiri, dan penugasan itu dengan benar mengisi `nil.prev` dengan simpul baru.

9.12 Sesudah `x.prev.next ← x.next; x.next.prev ← x.prev`, kedua tetangga melompati `x`, tetapi `x.prev` dan `x.next` masih menyebut mereka, sehingga `x.prev.next ← x; x.next.prev ← x` membuat kedua tetangga menunjuk `x` lagi, yaitu keadaan sebelum pelepasan. Pengembalian ini benar hanya bila kedua tetangga itu masih bertetangga satu sama lain. Ambil list a, x, y, b. Melepas x menautkan $a \leftrightarrow y$; melepas y lalu menautkan $a \leftrightarrow b$, dan y mengingat a dan b. Mengembalikan y lebih dulu menautkan $a \leftrightarrow y \leftrightarrow b$, lalu mengembalikan x (yang mengingat a dan y) menghasilkan a, x, y, b: benar. Mengembalikan x lebih dulu justru mengisi $a.next \leftarrow x$ dan $y.prev \leftarrow x$ sementara y tidak berada di dalam list, sehingga $a.next$ adalah x tetapi $x.next$ adalah y, simpul yang sudah dilepas, dan $b.prev$ masih a: rantai majunya a, x, y, b dan rantai mundurnya b, a. Invariannya rusak. Algoritma runut-balik Knuth mengandalkan persis disiplin masuk-terakhir-keluar-pertama ini, yang disediakan secara alami oleh pencarian rekursif [16].

9.13 (a) Saat berjalan maju, penyusuran memegang alamat simpul sebelumnya q dan simpul sekarang p; karena $p.link = q \oplus next$ dan $q \oplus q = 0$, alamat berikutnya adalah $p.link \oplus q$. Rumus yang sama berjalan mundur bila q adalah simpul sesudahnya. Di ujung-ujung list, tetangga yang tidak ada dianggap beralamat 0. (b) Satu pointer per simpul, $8n$ bita: memori *single linked list*. (c) Menghapus (atau menyisipkan di samping) simpul yang *hanya* diberikan pointernya: tanpa alamat salah satu tetangga, tetangga yang lain tidak dapat dihitung, sehingga program

harus berjalan ke simpul itu dari salah satu ujung. (d) Java dan Python tidak mengizinkan program melihat atau menggabungkan alamat: referensi tidak dapat diubah menjadi bilangan lalu dikembalikan, dan *garbage collector* harus dapat mengenali setiap referensi, yang mustahil dilakukannya di dalam hasil *exclusive-or*.

9.14 (a) Sebuah *queue*: *single linked list* dengan *head* dan *tail* (atau larik sirkular) sudah cukup, karena yang dibutuhkan hanya penghapusan di depan dan penyisipan di belakang, keduanya $O(1)$; pointer kedua milik *double linked list* tidak akan terpakai. (b) *Double linked list* dengan kursor: Undo adalah `cur = cur->prev` dan Redo adalah `cur = cur->next`, keduanya $O(1)$; suntingan baru sesudah beberapa *undo* menghapus segala sesudah kursor lalu menambahkan di belakang. (Dua *stack* adalah alternatif yang sama baiknya.) (c) *Double linked list* dengan *sentinel* ditambah *hash map*, seperti di Subbab 9.9: setiap peminjaman memakan waktu harapan $O(1)$, sedangkan larik atau list tanpa *map* akan mencari di antara 1.000 entri pada setiap peminjaman. (d) Larik: pencarian biner membutuhkan akses $O(1)$ berdasarkan indeks, yang tidak ditawarkan *linked list* mana pun; daftar itu diurutkan sekali dan dibaca berkali-kali, sehingga ongkos penyisipan $\Theta(n)$ pada larik jarang berarti.

Daftar Pustaka

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, sec. 2.2.5 (doubly linked lists).

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, sec. 10.2 (linked lists and sentinels).

[3] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998, sec. 6.1 (sequential searching).

[4] Oracle, "Class LinkedList," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/LinkedList.html. [Accessed: 24-Sep-2026].

[5] Valgrind Developers, "Valgrind: Current releases" (Valgrind 3.27.1, 20 May 2026). [Online]. Available: https://valgrind.org/downloads/current.html. [Accessed: 24-Sep-2026].

[6] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. 2012 USENIX Annual Technical Conference*, Boston, MA, USA, Jun. 2012, pp. 309–318.

[7] ISO/IEC 14882:2017, *Programming Languages — C++*, 5th ed. Geneva, Switzerland: ISO, 2017, sec. 26.3.10 (class template `list`).

[8] cppreference.com, "std::list." [Online]. Available: https://en.cppreference.com/w/cpp/container/list. [Accessed: 24-Sep-2026].

[9] GNU Project, "libstdc++-v3/include/bits/stl_list.h," GCC source repository. [Online]. Available: https://gcc.gnu.org/git/?p=gcc.git;a=blob;f=libstdc%2B%2B-v3/include/bits/stl_list.h. [Accessed: 24-Sep-2026].

[10] Oracle, "Class LinkedHashMap<K,V>," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/LinkedHashMap.html. [Accessed: 24-Sep-2026].

[11] Python Software Foundation, "Modules/_collectionsmodule.c," CPython source repository. [Online]. Available: https://github.com/python/cpython/blob/main/Modules/_collectionsmodule.c. [Accessed: 24-Sep-2026].

[12] Python Software Foundation, "collections — Container datatypes," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/collections.html. [Accessed: 24-Sep-2026].

[13] Python Software Foundation, “Lib/collections/**init**.py,” CPython source repository. [Online]. Available: https://github.com/python/cpython/blob/main/Lib/collections/**init**.py. [Accessed: 24-Sep-2026].

[14] Python Software Foundation, “Lib/functools.py,” CPython source repository. [Online]. Available: https://github.com/python/cpython/blob/main/Lib/functools.py. [Accessed: 24-Sep-2026].

[15] The Linux Kernel Documentation, “Linked Lists in Linux.” [Online]. Available: https://docs.kernel.org/core-api/list.html. [Accessed: 24-Sep-2026].

[16] D. E. Knuth, “Dancing links,” in *Millennial Perspectives in Computer Science*, J. Davies, B. Roscoe, and J. Woodcock, Eds. Basingstoke, U.K.: Palgrave, 2000, pp. 187–214. [Online]. Available: https://arxiv.org/abs/cs/0011047

[17] Profitina, “Profitina — AI Visibility (GEO) platform.” [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

Bab 10
Rekursi

Di akhir bab ini Anda akan sanggup menyatakan dengan tepat apa itu definisi rekursif dan fungsi rekursif, serta mengenali dua bagian yang dimiliki setiap definisi atau fungsi rekursif yang benar: *base case* (kasus dasar) yang langsung memberi jawaban, dan *recursive case* (kasus rekursif) yang menyerahkan masalah yang lebih kecil kepada fungsi yang sama; membuktikan kebenaran sebuah fungsi rekursif dengan induksi matematika, dan menemukan ongkosnya dengan menuliskan lalu menyelesaikan sebuah relasi rekurens; menggambar *call stack* sebuah rekursi yang sedang berjalan, *frame* demi *frame*, dan membaca pohon rekursi; menulis faktorial dengan gelung maupun dengan rekursi, dan menjelaskan ongkos masing-masing; menghitung bilangan Fibonacci dari kisah kelinci yang terkenal itu dengan enam cara (rekursi naif, rekursi dengan memo, tabel *bottom-up*, dua variabel, rekursi ekor, dan *fast doubling*) serta dengan rumus tertutup Binet, dan menjelaskan, berdasarkan jumlah panggilan dan waktu yang diukur, mengapa cara naif tidak berguna lagi begitu n melewati beberapa puluh; menghitung faktor persekutuan terbesar dengan algoritma Euklides dan membuktikan bahwa langkah yang diperlukannya hanya sebanyak logaritma bilangannya; memangkatkan sebuah bilangan dengan pangkat yang sangat besar hanya dengan segelintir perkalian; menulis penyelesai rekursif Menara Hanoi yang dijanjikan di Bab 5 dan membuktikan bahwa $2^n - 1$ langkahnya adalah yang paling sedikit; melakukan rekursi atas larik dan *linked list*; menjelaskan apa itu *tail call* (panggilan ekor) dan apa yang sesungguhnya dilakukan GCC, mesin virtual Java, dan CPython terhadapnya, diperlihatkan lewat kode *assembly* dan *crash* yang nyata; mengetahui seberapa dalam rekursi boleh berjalan di C++, Java, dan Python serta cara menaikkan setiap batasnya; dan mengubah rekursi apa pun menjadi gelung dengan *stack* buatan sendiri. Setiap program disajikan dalam C++, Java, dan Python, dikompilasi, dijalankan, dan, bila membangun struktur berantai, diperiksa keamanan memorinya.

10.1 Dari Mana Kita Melanjutkan

Rekursi sudah lama menunggu di balik layar buku ini. Subbab 2.5.5 menulis *binary search* sebagai fungsi yang memanggil dirinya sendiri pada separuh larik, dan berargumen, secara informal, bahwa fungsi itu benar karena ia benar pada setiap interval yang lebih pendek. Bab 3 memakai rekursi tanpa banyak basa-basi dalam *merge sort* dan *quicksort* (Subbab 3.8 dan 3.9), dan berjanji bahwa pohon rekursi yang digambar di sana akan digambar dan diselesaikan secara umum kelak. Bab 5 menuturkan kisah Menara Hanoi (Subbab 5.2), menemukan bahwa n cakram memerlukan $2^n - 1$ langkah, menyelesaikan teka-teki itu dengan tiga *stack* dan sebuah gelung, lalu menyisakan penyelesai rekursifnya untuk bab ini. Subbab 5.10 kemudian memperlihatkan *stack* terpenting di setiap komputer, yaitu *call stack*, dan ditutup dengan fungsi kecil `countdown` yang tiga *frame*-nya berada di *stack* pada saat yang sama. Bab 7 menghitung panjang sebuah *linked list* dan membaliknya secara rekursif (Subbab 7.6.2 dan 7.6.7), dan mengingatkan bahwa fungsi seperti itu memerlukan satu *stack frame* per simpul. Bab 9 ditutup dengan `functools.lru_cache`, tembolok hasil fungsi yang disimpan dalam *doubly linked list* sirkular, yang pemakaiannya paling terkenal kita jumpai di Subbab 10.5.

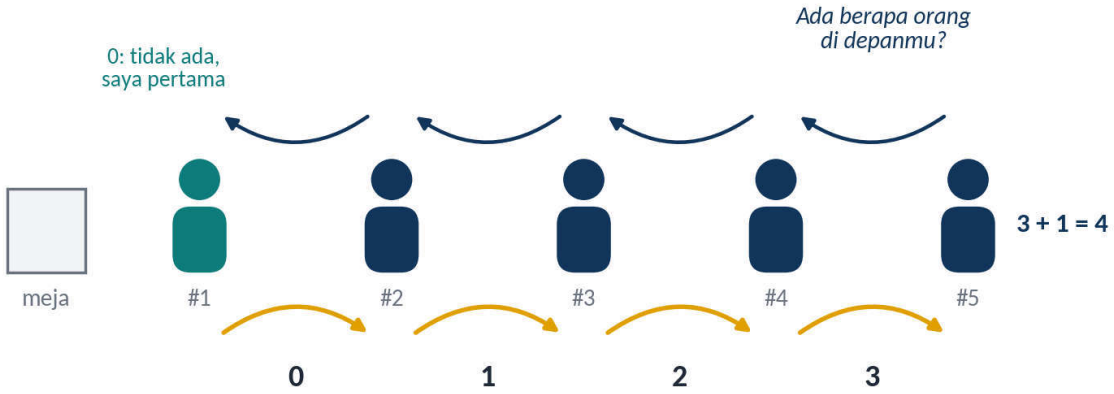
Bab ini mengumpulkan semua benang itu dan menjadikan rekursi alat yang dapat Anda percayai. Kita mulai dengan sebuah kisah dan sebuah definisi (Subbab 10.2), lalu mengamati mesin menjalankan panggilan rekursif, *frame* demi *frame* (Subbab 10.3). Lima masalah klasik menyusul, masing-masing mengajarkan satu gagasan baru: faktorial (rekursi paling sederhana, Subbab 10.4), Fibonacci (bagaimana rekursi dapat meledak, dan empat cara menjinakkannya, Subbab 10.5), faktor persekutuan terbesar Euklides (rekursi yang menyusut sangat cepat, Subbab 10.6), pemangkatan cepat (membagi dua alih-alih mengurangi satu, Subbab 10.7), dan Menara Hanoi (masalah yang penyelesaian rekursifnya optimal namun tetap eksponensial, Subbab 10.8). Subbab 10.9 menerapkan rekursi pada larik dan *linked list*, Subbab 10.10 menggambar pohon rekursi dan menyelesaikan rekurens yang digambarkannya, dan Subbab 10.11 sampai 10.13 menjawab pertanyaan praktis: apa yang dilakukan *compiler* terhadap *tail call*, seberapa dalam rekursi boleh berjalan di setiap bahasa, dan bagaimana mengganti rekursi dengan gelung dan *stack* bila rekursinya terlalu dalam. Bab 11, tentang pohon, akan memakai semua isi bab ini hampir di setiap halamannya, sebab pohon adalah struktur rekursif: sebuah akar dengan pohon-pohon yang lebih kecil di bawahnya.

10.2 Berpikir Secara Rekursif

10.2.1 Pertanyaan yang diteruskan sepanjang antrean

Bayangkan antrean di meja peminjaman perpustakaan Pustaka Ceria pada Sabtu pagi yang ramai. Anda berada di tengah antrean, antreannya berbelok mengitari sebuah rak buku, dan Anda tidak dapat melihat meja. Anda ingin tahu ada berapa orang di depan Anda. Anda bisa keluar dari antrean dan menghitung, tetapi tempat Anda bisa diambil orang. Ada cara yang lebih malas. Anda menepuk bahu orang di depan Anda dan bertanya: "Ada berapa orang di depan*mu*?" Berapa pun jawabannya, Anda menambahkan satu (untuk dia), dan Anda pun tahu jawabannya.

Tentu orang di depan Anda juga tidak tahu. Tetapi ia dapat berbuat persis seperti Anda: bertanya kepada orang di depannya, lalu menambahkan satu pada jawabannya. Pertanyaan itu merambat ke depan sepanjang antrean sampai tiba pada orang yang berdiri di meja. Orang itu tidak perlu bertanya kepada siapa pun: tidak ada orang di depannya, jadi ia langsung menjawab "nol". Lalu jawaban-jawaban merambat kembali: orang kedua mendengar 0 dan berkata 1, orang ketiga mendengar 1 dan berkata 2, dan seterusnya, sampai jawabannya kembali kepada Anda. **Gambar 10.1** memperlihatkan seluruh percakapan itu untuk antrean lima orang.



Gambar 10.1 Menghitung orang di depan dengan bertanya: setiap pertanyaan maju satu tempat (biru tua), orang pertama menjawab 0 tanpa bertanya kepada siapa pun, dan setiap jawaban kembali dengan tambahan satu (kuning).

Tiga ciri prosedur kecil ini adalah seluruh isi rekursi.

- **Semua orang menjalankan prosedur yang sama.** Tidak ada yang memerlukan petunjuk berbeda. "Bila tidak ada orang di depanku, jawab 0; bila ada, tanyai orang di depan dan jawab satu lebih banyak daripada jawabannya."
- **Setiap pertanyaan lebih kecil.** Orang yang Anda tanyai memiliki lebih sedikit orang di depannya daripada Anda. Pertanyaan itu menyusut satu orang setiap kali diteruskan.
- **Ada yang menjawab tanpa bertanya.** Orang di meja menjawab secara langsung. Tanpa orang itu, pertanyaannya akan merambat selamanya (atau, dalam antrean yang melingkari sebuah tiang, berputar-putar).

Perhatikan pula apa yang terjadi *selama* pertanyaan itu merambat: setiap orang yang sudah bertanya berdiri menunggu sambil mengingat bahwa ia harus menambahkan satu pada jawaban yang nanti kembali. Pada saat terdalam, lima orang sedang menunggu. Dalam program, orang-orang yang menunggu itu adalah *stack frame*, dan tempat mereka menunggu adalah *call stack* dari Subbab 5.10. Subbab 10.3 mengamati mereka.

10.2.2 Definisi rekursif

Para matematikawan sudah memakai rekursi jauh sebelum ada komputer. Sebuah *definisi rekursif* mendefinisikan sesuatu sebagian berdasarkan contoh yang lebih kecil dari sesuatu itu sendiri. Berikut jenis yang paling sederhana, sebuah barisan bilangan:

$$f(0) = 3, \qquad f(n) = 2f(n-1) + 3 \quad \text{untuk } n \geq 1. \tag{10.1}$$

Persamaan (10.1) tidak memberi tahu $f(4)$ secara langsung. Ia memberi tahu cara memperoleh $f(4)$ dari $f(3)$, $f(3)$ dari $f(2)$, dan seterusnya sampai $f(0)$, yang dinyatakannya secara langsung. Bila diuraikan:

$$f(0) = 3$$

$$f(1) = 2f(0) + 3 = 2 \cdot 3 + 3 = 9$$

$$f(2) = 2f(1) + 3 = 2 \cdot 9 + 3 = 21$$

$$f(3) = 2f(2) + 3 = 2 \cdot 21 + 3 = 45$$

$$f(4) = 2f(3) + 3 = 2 \cdot 45 + 3 = 93$$

Setiap definisi rekursif memiliki dua bagian yang sama, dan keduanya memiliki nama yang terus dipakai di sepanjang bab ini.

> **DEFINISI.** Sebuah *definisi rekursif* mendefinisikan suatu objek berdasarkan contoh-contoh yang lebih kecil dari objek itu sendiri. *Base case*-nya (disebut juga *jangkar* atau *aturan berhenti*) memberikan jawaban untuk contoh terkecil secara langsung, tanpa merujuk pada objek itu. *Recursive case*-nya (disebut juga *langkah induktif*) memberikan jawaban untuk contoh yang lebih besar berdasarkan satu atau lebih contoh yang benar-benar lebih kecil. Sebuah *fungsi rekursif* adalah fungsi yang badannya memanggil fungsi itu sendiri, secara langsung atau melalui fungsi lain; *base case*-nya kembali tanpa panggilan seperti itu.

Dalam Persamaan (10.1), *base case*-nya adalah $f(0) = 3$ dan *recursive case*-nya $f(n) = 2f(n - 1) + 3$. Kata "benar-benar lebih kecil" itu penting. Definisi seperti $g(n) = g(n) + 1$ merujuk pada contoh yang sama sekali tidak lebih kecil, dan tidak mendefinisikan apa pun. Definisi seperti $h(n) = h(n + 1) - 1$ merujuk pada contoh yang *lebih besar*, dan penguraiannya tidak pernah mencapai *base case*.

10.2.3 Membuktikan definisi rekursif: induksi

Nilai-nilai 3, 9, 21, 45, 93 tampak selalu tiga kurangnya dari kelipatan enam: $3 = 6 - 3$, $9 = 12 - 3$, $21 = 24 - 3$, $45 = 48 - 3$, $93 = 96 - 3$. Kelipatannya adalah $6 \cdot 1$,$6 \cdot 2$,$6 \cdot 4$,$6 \cdot 8$,$6 \cdot 16$, sehingga dugaannya

$$f(n) = 6 \cdot 2^n - 3 \quad \text{untuk setiap } n \geq 0. \tag{10.2}$$

Rumus seperti Persamaan (10.2), yang menghitung $f(n)$ tanpa menghitung nilai sebelumnya, disebut *bentuk tertutup* (*closed form*). Lima nilai yang cocok belum membuktikannya; nilai keenam bisa saja meleset. Alat untuk membuktikan pernyataan tentang definisi rekursif adalah *induksi matematika*, dan alat itu memiliki dua bagian yang persis sama dengan definisinya sendiri.

- **Base case.** Periksa pernyataannya untuk n terkecil. Untuk $n = 0$: $6 \cdot 2^0 - 3 = 6 - 3 = 3 = f(0)$. Benar.
- **Langkah induktif.** Anggap pernyataan itu berlaku untuk $n - 1$ (anggapan ini disebut *hipotesis induksi*), lalu tunjukkan bahwa ia juga berlaku untuk n. Misalkan $f(n - 1) = 6 \cdot 2^{n-1} - 3$. Maka, menurut *recursive case* Persamaan (10.1),

$$f(n) = 2f(n - 1) + 3 = 2(6 \cdot 2^{n-1} - 3) + 3 = 6 \cdot 2^n - 6 + 3 = 6 \cdot 2^n - 3.$$

Mengapa ini membuktikan rumusnya untuk *setiap* n? *Base case* memberi $n = 0$. Langkahnya, bila diterapkan dengan $n = 1$, mengubah kebenaran untuk 0 menjadi kebenaran untuk 1; diterapkan dengan $n = 2$, mengubah kebenaran untuk 1 menjadi kebenaran untuk 2; dan seterusnya. Setiap n tercapai setelah berhingga banyak langkah, persis seperti penguraian definisinya mencapai $f(0)$ setelah berhingga banyak langkah. Induksi dan rekursi adalah satu gagasan yang dilihat dari dua sisi: rekursi *menghitung* dengan menurunkan ke kasus yang lebih kecil, sedangkan induksi *membuktikan* dengan membangun dari kasus yang lebih kecil.

> **WAWASAN KUNCI:** Untuk meyakinkan diri bahwa sebuah fungsi rekursif benar, jangan menelusuri setiap panggilannya. Periksa *base case*-nya, lalu *anggaplah* panggilan-panggilan rekursifnya mengembalikan jawaban yang benar untuk masukan yang lebih kecil, dan periksa apakah fungsi itu menggabungkan jawaban-jawaban tersebut dengan benar. Itulah bukti induksi, dan hanya dengan cara itu kita dapat bernalar tentang rekursi yang dalamnya seribu panggilan. Para pemrogram menyebut sikap ini "memercayai rekursinya", kadang *lompatan iman rekursif* (*recursive leap of faith*); bukti induksilah yang membuat kepercayaan itu beralasan.

10.2.4 Dari definisi ke program

Sebuah definisi rekursif berubah menjadi fungsi rekursif hampir kata demi kata. Bentuk umumnya adalah

```
return_type function_name(parameters) {
    if (the problem is small enough)        // base case
        return the answer directly;
    ... function_name(smaller parameters) ...   // recursive case
    return an answer built from the smaller answers;
}
```

Kode 10.1 menuliskan Persamaan (10.1) dengan cara ini dalam C++, dua kali: sekali secara polos, sebagai `f`, dan sekali sebagai `fTraced`, yang melakukan pekerjaan yang sama tetapi mencetak satu baris ketika dipanggil dan satu baris lagi ketika kembali, menjorok empat spasi untuk setiap tingkat kedalaman. Fungsi `main` menelusuri $f(3)$ lalu memeriksa fungsi rekursifnya terhadap bentuk tertutup Persamaan (10.2) untuk $n = 0,1,\ldots,10$.

Kode 10.1 `01_recursive_f.cpp`: definisi rekursif Persamaan (10.1) sebagai fungsi C++, ditelusuri, dan diperiksa terhadap bentuk tertutupnya.

```cpp
// 01_recursive_f.cpp -- a first recursive function, f(0) = 3, f(n) = 2 f(n-1) + 3,
// traced call by call, then checked against the closed form 6 * 2^n - 3.
#include <cstdio>

long long f(int n) {
    if (n == 0) return 3;     // base case
    return 2 * f(n - 1) + 3;  // recursive case: a smaller n
}

long long fTraced(int n, int depth) {  // the same function, talking as it works
    std::printf("%*scall f(%d)\n", 4 * depth, "", n);
    long long result;
    if (n == 0) {
        result = 3;
    } else {
        long long smaller = fTraced(n - 1, depth + 1);
        result = 2 * smaller + 3;
    }
    std::printf("%*sf(%d) returns %lld\n", 4 * depth, "", n, result);
    return result;
}

int main() {
    std::printf("Tracing f(3):\n");
    fTraced(3, 0);

    std::printf("\n n      f(n)   6*2^n-3\n");
    for (int n = 0; n <= 10; ++n) {
        long long closed = 6LL * (1LL << n) - 3;
        std::printf("%2d %9lld %9lld%s\n", n, f(n), closed,
                    f(n) == closed ? "" : "  MISMATCH");
    }
    return 0;
}
```

```
Tracing f(3):
call f(3)
    call f(2)
        call f(1)
            call f(0)
            f(0) returns 3
        f(1) returns 9
    f(2) returns 21
f(3) returns 45

 n      f(n)   6*2^n-3
 0         3         3
 1         9         9
 2        21        21
 3        45        45
 4        93        93
 5       189       189
 6       381       381
 7       765       765
 8      1533      1533
 9      3069      3069
10      6141      6141
```

Bacalah jejaknya perlahan-lahan, sebab itulah antrean **Gambar 10.1** sekali lagi. `f(3)` dipanggil dan, sebelum dapat mengembalikan apa pun, memanggil `f(2)`, yang memanggil `f(1)`, yang memanggil `f(0)`. Kini empat panggilan belum selesai pada saat yang sama, masing-masing menjorok satu langkah lebih jauh. Hanya `f(0)`, sang *base case*, yang dapat langsung menjawab. Lalu panggilan-panggilan itu selesai dalam urutan yang *berkebalikan* dengan urutan dimulainya: `f(0)` mengembalikan 3, `f(1)` memakainya untuk mengembalikan 9, `f(2)` mengembalikan 21, dan `f(3)` mengembalikan 45. Tabelnya menegaskan bukti induksi tadi: fungsi rekursif dan bentuk tertutupnya sepakat di setiap baris. Versi Java dan Python di folder Code (`RecursiveF.java`, `01_recursive_f.py`) mencetak baris-baris yang persis sama.

10.2.5 Tiga aturan, dan sepatah kata jujur tentang ongkos

Sebuah fungsi rekursif bekerja bila tiga aturan berikut dipenuhi.

1. **Ada *base case* dan ia diuji lebih dahulu.** Setiap rantai panggilan harus berakhir pada kasus yang kembali tanpa rekursi.
2. **Setiap panggilan rekursif maju menuju *base case*.** Argumennya harus benar-benar mengecil menurut suatu ukuran yang tidak dapat berkurang selamanya: bilangan bulat tak negatif yang turun, interval yang memendek, list yang kehilangan satu simpul. Pada `f`, n berkurang satu dan berhenti di 0. Subbab 10.3.4 memperlihatkan apa yang terjadi bila aturan ini dilanggar.
3. **Percayai panggilan rekursifnya, dan gabungkan dengan benar.** Anggaplah panggilan pada masalah yang lebih kecil itu benar (hipotesis induksi), dan pastikan jawaban untuk seluruh masalah dibangun dengan benar darinya.

Rekursi juga ada harganya, dan lebih baik harga itu disebutkan sekarang daripada ditemukan belakangan. Setiap panggilan yang masih menunggu menempati satu *stack frame*, sehingga rekursi sedalam n tingkat memakai memori sebanding dengan n di samping apa pun yang dibutuhkan masalahnya, dan *call stack* itu kecil (Subbab 10.12 mengukurnya). Setiap panggilan juga memakan waktu untuk pembukuannya, biasanya beberapa nanodetik di C++, sehingga fungsi rekursif jarang lebih cepat daripada gelung yang setara. Yang terburuk, sebuah definisi rekursif yang dituliskan secara naif bisa menyelesaikan submasalah yang sama berulang-ulang; Subbab 10.5 memperlihatkan fungsi yang melakukan 331 juta panggilan untuk menghasilkan satu bilangan yang dapat ditemukan sebuah gelung dengan 90 penjumlahan. Ada aturan praktis lama

yang berbunyi “bila suatu masalah mudah diselesaikan dengan gelung, pakailah gelung”. Aturan itu sehat, dan bab ini memperlihatkan di mana ia berlaku (faktorial, Fibonacci, Euklides) dan di mana tidak: Menara Hanoi, pohon (Bab 11), graf (Bab 13), pengurutan *divide and conquer* (Bab 3), dan setiap struktur yang bersarang sampai kedalaman yang tidak diketahui, tempat program rekursifnya pendek, jernih, dan benar, sedangkan versi gelungnya harus membangun dengan tangan *stack* yang disediakan rekursi secara cuma-cuma.

Gagasan ini sudah tua. Definisi rekursif barisan bilangan sudah dipakai berabad-abad, dan bukti dengan induksi dibuat cermat pada abad kesembilan belas. Dalam pemrograman, rekursi datang bersama LISP, yang deskripsinya oleh John McCarthy pada 1960 berjudul “Recursive functions of symbolic expressions and their computation by machine” [1], dan bersama ALGOL 60, yang laporannya mengizinkan sebuah prosedur memanggil dirinya sendiri. Pada tahun yang sama Edsger Dijkstra menjelaskan cara mengimplementasikan panggilan semacam itu dengan *stack* berisi *activation record*, mekanisme yang masih dipakai setiap bahasa hingga kini [2]. Cormen dkk. [3] dan Knuth [4] membahas rekursi, rekurens, dan induksi dengan kecermatan yang menjadi sasaran bab ini.

10.3 Call Stack, Frame demi Frame

10.3.1 Frame, sekali lagi

Subbab 5.10 menjelaskan apa yang terjadi ketika sebuah fungsi memanggil fungsi lain. Prosesor harus mengingat di mana harus melanjutkan di fungsi pemanggil, dan fungsi yang dipanggil memerlukan tempat untuk parameter dan variabel lokalnya. Karena itu program yang sedang berjalan memelihara sebuah *call stack*, dan setiap panggilan melakukan *push* sebuah *stack frame* (*activation record*) yang berisi alamat kembali, parameter, variabel lokal, dan beberapa register yang disimpan; setiap *return* melakukan *pop* terhadap *frame* itu dan menyerahkan nilai kembaliannya ke *frame* di bawahnya.

Tidak ada yang istimewa bagi rekursi dalam mekanisme ini, dan justru itulah intinya. Ketika `f(3)` memanggil `f(2)`, program melakukan persis apa yang dilakukannya ketika `main` memanggil `printf`: ia melakukan *push* sebuah *frame* baru. *Frame* baru itu milik *fungsi* yang sama tetapi merupakan *aktivasi* yang berbeda, dengan salinan `n`-nya sendiri. `n` milik `f(3)` bernilai 3 dan tetap 3 selama `f(2)`, `f(1)`, dan `f(0)` berjalan di atasnya dengan `n` mereka sendiri, yaitu 2, 1, dan 0. Itulah sebabnya rekursi dapat bekerja: *frame-frame* itu memisahkan panggilan-panggilan yang sedang menunggu.

10.3.2 Faktorial, frame demi frame

Faktorial sebuah bilangan bulat tak negatif n, ditulis $n!$, adalah hasil kali bilangan-bilangan bulat dari 1 sampai n, dengan $0! = 1$ (hasil kali kosong):

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1.$$

Misalnya $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$ dan $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$. Letakkan kedua hasil kali itu berdampingan: $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot 4!$. Faktorial sebuah bilangan adalah bilangan itu dikali faktorial bilangan sebelumnya, dan itu adalah definisi rekursif:

$$n! = \begin{cases} 1 & \text{jika } n = 0, \\ n \cdot (n-1)! & \text{jika } n \geq 1. \end{cases} \tag{10.3}$$

Base case Persamaan (10.3) adalah $0! = 1$ dan *recursive case*-nya $n! = n \cdot (n-1)!$. Bila diuraikan untuk nilai-nilai kecil, satu langkah per baris:

$$0! = 1$$

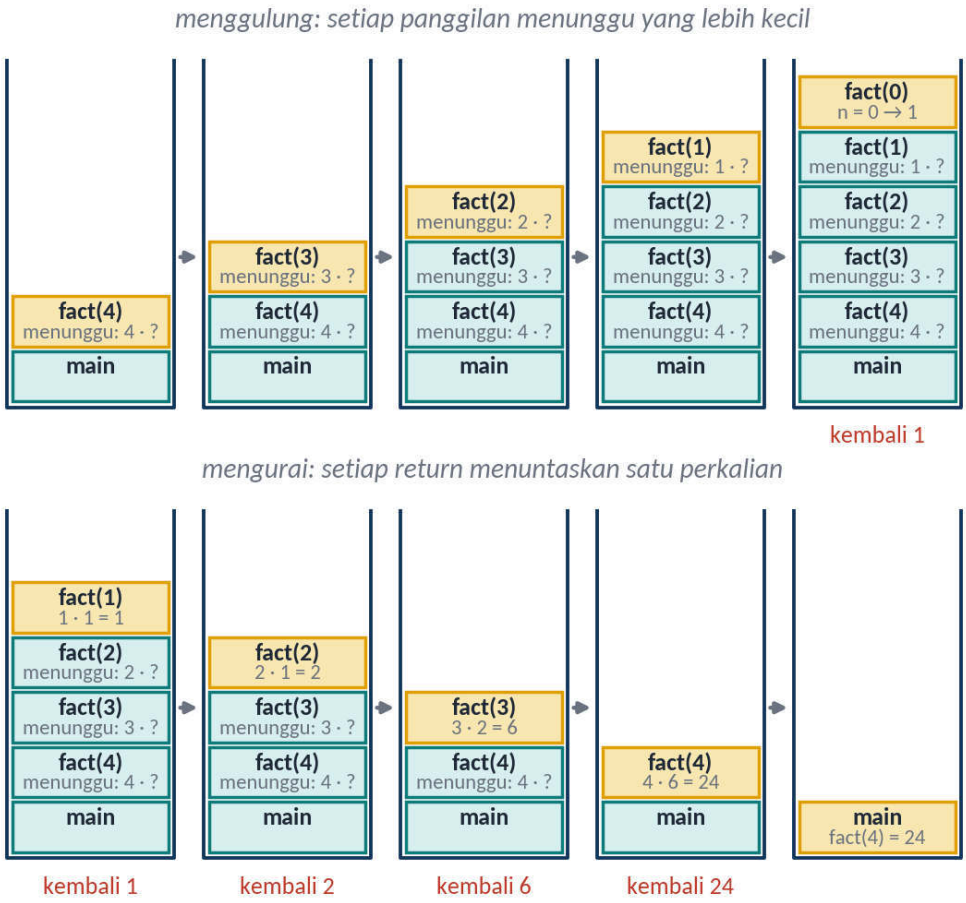
$$1! = 1 \cdot 0! = 1 \cdot 1 = 1$$

$$2! = 2 \cdot 1! = 2 \cdot 1 = 2$$

$$3! = 3 \cdot 2! = 3 \cdot 2 = 6$$

$$4! = 4 \cdot 3! = 4 \cdot 6 = 24$$

Gambar 10.2 memperlihatkan isi *call stack* selama fungsi rekursif `factRec(4)` dari subbab berikutnya menghitung 4!. Pada baris atas, *stack* itu *menggulung*: `main` memanggil `fact(4)`, yang belum dapat mengalikan karena memerlukan `fact(3)`; `fact(3)` pada gilirannya menunggu `fact(2)`, dan seterusnya sampai `fact(0)` berada di puncak, enam *frame* tingginya bila `main` ikut dihitung. `fact(0)` adalah *base case* dan mengembalikan 1 tanpa memanggil apa pun. Pada baris bawah, *stack* itu *mengurai*: setiap *return* melakukan *pop* satu *frame*, dan *frame* di bawahnya, yang sejak tadi menunggu dengan "$k \cdot$?", akhirnya mendapat bilangan yang kurang dan melakukan satu-satunya perkaliannya: $1 \cdot 1 = 1$, $2 \cdot 1 = 2$, $3 \cdot 2 = 6$, dan $4 \cdot 6 = 24$.



Gambar 10.2 *Call stack* selama `fact(4)` berjalan. Baris atas: setiap panggilan menunggu panggilan yang lebih kecil sampai *base case* `fact(0)` berada di puncak. Baris bawah: setiap *return* melakukan *pop* satu *frame* dan menuntaskan satu perkalian di *frame* di bawahnya.

Dua fakta dapat dibaca langsung dari gambar itu. Pertama, *tidak ada perkalian dalam perjalanan turun*; keempat perkalian terjadi dalam perjalanan naik kembali, dengan urutan $1 \cdot 1$, $2 \cdot 1$, $3 \cdot 2$, $4 \cdot 6$, kebalikan dari urutan panggilannya. Kedua, tinggi *stack* mencapai $n + 2$ *frame* untuk $n!$ (sebanyak $n + 1$ panggilan `fact(n)` sampai `fact(0)`, ditambah `main`). Waktunya sebanding dengan banyaknya panggilan, $n + 1$, demikian pula memori tambahannya.

10.3.3 Jejak itu sebagai program

Kode 10.2 menghitung $n!$ dengan kedua cara, dengan gelung (`factIter`) dan dengan rekursi (`factRec`), dan menelusuri `fact(4)` dengan trik indentasi yang sama seperti **Kode 10.1**. Program itu lalu mencetak kedua versi berdampingan untuk $n = 0$ sampai 20 dan akhirnya meminta 21!.

Kode 10.2 `02_factorial.cpp`: faktorial dengan gelung dan dengan rekursi, jejak `fact(4)`, dan batas aritmetika 64 bit.

```cpp
// 02_factorial.cpp -- n! computed iteratively and recursively, one recursive call
// traced frame by frame, and the limit of a 64-bit integer.
#include <cstdio>
#include <string>

unsigned long long factIter(int n) {
    unsigned long long result = 1;
    for (int i = 2; i <= n; ++i) result *= i;  // 1 * 2 * 3 * ... * n
    return result;
}

unsigned long long factRec(int n) {
    if (n == 0) return 1;       // base case: 0! = 1
    return n * factRec(n - 1);  // recursive case: n! = n * (n-1)!
}

unsigned long long factTraced(int n, int depth) {
    std::printf("%*sfact(%d): ", 4 * depth, "", n);
    if (n == 0) {
        std::printf("base case, returns 1\n");
        return 1;
    }
    std::printf("waits for fact(%d)\n", n - 1);
    unsigned long long below = factTraced(n - 1, depth + 1);
    unsigned long long result = n * below;
    std::printf("%*sfact(%d): %d * %llu = %llu\n", 4 * depth, "", n, n, below,
                result);
    return result;
}

std::string times(const std::string &decimal, int k) {  // exact decimal * small k
    std::string out;
    int carry = 0;
    for (int i = (int)decimal.size() - 1; i >= 0; --i) {
        int d = (decimal[i] - '0') * k + carry;
        out.insert(out.begin(), char('0' + d % 10));
        carry = d / 10;
    }
    while (carry > 0) {
        out.insert(out.begin(), char('0' + carry % 10));
        carry /= 10;
    }
    return out;
}
```

Kode 10.2 (Lanjutan) *02_factorial.cpp: faktorial dengan gelung dan dengan rekursi, jejak fact(4), dan batas aritmetika 64 bit.*

```cpp
int main() {
    std::printf("Tracing fact(4):\n");
    factTraced(4, 0);

    std::printf("\n n                    n! (loop)          n! (recursion)\n");
    for (int n = 0; n <= 20; ++n)
        std::printf("%2d %24llu %24llu\n", n, factIter(n), factRec(n));

    std::string exact = "1";
    for (int k = 2; k <= 21; ++k) exact = times(exact, k);
    std::printf("\n21! in 64 bits: %llu (wrong: it wrapped around)\n", factRec(21));
    std::printf("21! exactly   : %s\n", exact.c_str());
    return 0;
}
```

```
Tracing fact(4):
fact(4): waits for fact(3)
    fact(3): waits for fact(2)
        fact(2): waits for fact(1)
            fact(1): waits for fact(0)
                fact(0): base case, returns 1
            fact(1): 1 * 1 = 1
        fact(2): 2 * 1 = 2
    fact(3): 3 * 2 = 6
fact(4): 4 * 6 = 24

 n                  n! (loop)          n! (recursion)
 0                          1                       1
 1                          1                       1
 2                          2                       2
 3                          6                       6
 4                         24                      24
 5                        120                     120
 6                        720                     720
 7                       5040                    5040
 8                      40320                   40320
 9                     362880                  362880
10                    3628800                 3628800
11                   39916800                39916800
12                  479001600               479001600
13                 6227020800              6227020800
14                87178291200             87178291200
15              1307674368000           1307674368000
16             20922789888000          20922789888000
17            355687428096000         355687428096000
18           6402373705728000        6402373705728000
19         121645100408832000      121645100408832000
20        2432902008176640000     2432902008176640000

21! in 64 bits: 14197454024290336768 (wrong: it wrapped around)
21! exactly   : 51090942171709440000
```

Jejak itu adalah **Gambar 10.2** dalam kata-kata: empat baris “waits for” dalam perjalanan turun, *base case*-nya, lalu empat perkalian dalam perjalanan naik. Tabelnya memperlihatkan kedua versi sepakat sampai $20! = 2{,}432{,}902{,}008{,}176{,}640{,}000$, faktorial terbesar yang muat dalam 64 bit: `unsigned long long` terbesar adalah $2^{64} - 1 \approx 1.8 \times 10^{19}$, sedangkan $21! \approx 5.1 \times 10^{19}$ hampir tiga kali lebih besar. C++ tidak melaporkan luapan (*overflow*) itu; aritmetika *unsigned* diam-diam hanya menyimpan 64 bit terendah dari hasil kali sebenarnya, yaitu $21! \bmod 2^{64} = 14{,}197{,}454{,}024{,}290{,}336{,}768$, bilangan yang tampak wajar tetapi sama sekali salah. Program itu mencetak nilai eksaknya di sebelahnya, yang dihitung digit demi digit dengan fungsi pembantu kecil `times` yang mengalikan string desimal dengan sebuah bilangan kecil, seperti kita mengalikan bersusun dengan tangan.

Versi Java dan Python mencetak keluaran yang sama. **Kode 10.3** memakai `long`, yang meluap dengan cara yang persis sama (`long` di Java bertanda, sehingga program menampilkan 64 bit itu sebagai bilangan tak bertanda dengan `Long.toUnsignedString` agar cocok dengan C++), dan `BigInteger` untuk nilai eksaknya.

Kode 10.3 `Factorial.java`: program yang sama dalam Java; `BigInteger` memberi 21! yang eksak.

```java
// Factorial.java -- n! computed iteratively and recursively, one recursive call
// traced frame by frame, and the limit of a 64-bit integer.
import java.math.BigInteger;

public class Factorial {
    static long factIter(int n) {
        long result = 1;
        for (int i = 2; i <= n; ++i) result *= i;  // 1 * 2 * 3 * ... * n
        return result;
    }

    static long factRec(int n) {
        if (n == 0) return 1;       // base case: 0! = 1
        return n * factRec(n - 1);  // recursive case: n! = n * (n-1)!
    }

    static long factTraced(int n, int depth) {
        String pad = " ".repeat(4 * depth);
        System.out.printf("%sfact(%d): ", pad, n);
        if (n == 0) {
            System.out.println("base case, returns 1");
            return 1;
        }
        System.out.printf("waits for fact(%d)%n", n - 1);
        long below = factTraced(n - 1, depth + 1);
        long result = n * below;
        System.out.printf("%sfact(%d): %d * %d = %d%n", pad, n, n, below, result);
        return result;
    }

    public static void main(String[] args) {
        System.out.println("Tracing fact(4):");
        factTraced(4, 0);

        System.out.println(
            "\n n                    n! (loop)          n! (recursion)");
        for (int n = 0; n <= 20; ++n)
            System.out.printf("%2d %24d %24d%n", n, factIter(n), factRec(n));
```

Kode 10.3 (Lanjutan) *Factorial.java: program yang sama dalam Java; BigInteger memberi 21! yang eksak.*

```java
        BigInteger exact = BigInteger.ONE;
        for (int k = 2; k <= 21; ++k) exact = exact.multiply(BigInteger.valueOf(k));
        // a long has 64 bits; read them as an unsigned number, as C++ does
        System.out.printf("%n21! in 64 bits: %s (wrong: it wrapped around)%n",
                          Long.toUnsignedString(factRec(21)));
        System.out.printf("21! exactly   : %s%n", exact);
    }
}
```

Bilangan bulat Python tidak pernah meluap: ia tumbuh menjadi sebanyak digit yang diperlukan. Karena itu **Kode 10.4** menghitung 21! secara eksak dengan fungsi rekursif biasa dan meniru hasil 64 bit dengan `% 2**64`, sisa bagi yang tersimpan dalam 64 bit.

Kode 10.4 `02_factorial.py`: program yang sama dalam Python, yang bilangan bulatnya tumbuh sesuai kebutuhan.

```python
# 02_factorial.py -- n! computed iteratively and recursively, one recursive call
# traced frame by frame, and the limit of a 64-bit integer.


def fact_iter(n):
    result = 1
    for i in range(2, n + 1):  # 1 * 2 * 3 * ... * n
        result *= i
    return result


def fact_rec(n):
    if n == 0:
        return 1                # base case: 0! = 1
    return n * fact_rec(n - 1)  # recursive case: n! = n * (n-1)!


def fact_traced(n, depth):
    pad = " " * (4 * depth)
    print(f"{pad}fact({n}): ", end="")
    if n == 0:
        print("base case, returns 1")
        return 1
    print(f"waits for fact({n - 1})")
    below = fact_traced(n - 1, depth + 1)
    result = n * below
    print(f"{pad}fact({n}): {n} * {below} = {result}")
    return result


def main():
    print("Tracing fact(4):")
    fact_traced(4, 0)

    print("\n n                    n! (loop)          n! (recursion)")
    for n in range(21):
        print(f"{n:2d} {fact_iter(n):24d} {fact_rec(n):24d}")

    exact = fact_rec(21)     # Python integers never overflow
    wrapped = exact % 2**64  # what 64 bits keep of it
    print(f"\n21! in 64 bits: {wrapped} (wrong: it wrapped around)")
    print(f"21! exactly   : {exact}")


if __name__ == "__main__":
    main()
```

10.3.4 Bila base case hilang

Langgarlah aturan 2 dari Subbab 10.2.5 dan lihat apa yang terjadi. Misalkan *recursive case*-nya kita tulis `return n * factRec(n + 1);` karena salah ketik. Argumennya kini membesar: `factRec(4)` memanggil `factRec(5)`, yang memanggil `factRec(6)`, dan *base case* `n == 0` tidak pernah tercapai. Program itu tidak berputar diam-diam selamanya seperti `while (true)`; setiap panggilan melakukan *push* sebuah *frame*, *frame-frame* itu menumpuk, dan setelah beberapa puluh ribu panggilan *stack*-nya penuh. Apa yang terjadi berikutnya bergantung pada bahasanya.

- **C++** membaca atau menulis memori tepat di luar ujung *stack*, tempat sistem operasi memasang halaman penjaga (*guard page*) yang bukan milik siapa pun, dan program langsung dimatikan dengan *segmentation fault* (di Linux dan macOS; Windows melaporkan eksepsi *stack overflow*). Tidak ada pesan yang memberi tahu fungsi mana penyebabnya; *debugger* atau AddressSanitizer dapat memberitahukannya.

- **Java** memeriksa *stack* pada setiap panggilan dan melempar `StackOverflowError`, dengan *stack trace* yang memperlihatkan metode yang sama berulang seribu kali.
- **Python** menghitung kedalaman *stack* interpreternya dan melempar `RecursionError: maximum recursion depth exceeded` bila kedalaman itu melampaui batas, secara bawaan 1.000.

Kegagalan ini memiliki nama yang kemudian menjadi nama sebuah situs web terkenal bagi para pemrogram: *stack overflow*. Subbab 10.12 mengukur dengan tepat di mana hal itu terjadi di setiap bahasa, dan Subbab 10.13 memperlihatkan cara menghindarinya untuk rekursi yang memang sah dalamnya.

> **PERINGATAN:** Rekursi yang tidak maju menuju *base case*-nya bukanlah program yang lambat; ia adalah *crash*, biasanya dalam sepersekian detik. Dua penyebab yang lazim adalah *base case* yang dapat dilompati argumennya (menguji `n == 0` padahal `n` bisa berpindah dari 1 ke -1, misalnya pada fungsi yang mengurangkan 2) dan argumen yang tidak mengecil (melewatkan `n` alih-alih `n - 1`). Ujilah masukan terkecil lebih dahulu: 0, 1, dan 2 menangkap sebagian besar *bug* semacam itu.

10.4 Faktorial: Iteratif dan Rekursif

10.4.1 Dua algoritma untuk satu fungsi

Metode *iteratif* menghitung $n!$ dengan gelung yang mengalikan setiap bilangan ke dalam hasil kali berjalan, mengikuti definisi $n! = n(n-1)(n-2)\cdots 1$ dari kiri ke kanan atau, seperti `factIter`, dari kanan ke kiri. Metode *rekursif* mengikuti Persamaan (10.3). **Pseudokode 10.1** menuliskan keduanya dalam pseudokode.

Pseudokode 10.1 Faktorial dengan gelung dan dengan rekursi.

```
FACTORIAL-ITERATIVE(n)
 1.  result ← 1
 2.  for i ← 2 to n
 3.      result ← result · i
 4.  return result

FACTORIAL-RECURSIVE(n)
 1.  if n = 0
 2.      return 1                        // base case: 0! = 1
 3.  return n · FACTORIAL-RECURSIVE(n − 1)
```

Keduanya benar, dan bukti-buktinya berbentuk sama. Untuk gelung, *loop invariant* (invarian gelung) "pada awal iterasi dengan indeks i, result $= (i-1)!$" berlaku sebelum iterasi pertama ($1 = 1!$) dan dipertahankan oleh baris 3 ($(i-1)! \cdot i = i!$), sehingga setelah iterasi terakhir result $= n!$. Untuk versi rekursif, induksi pada n: *base case* mengembalikan $1 = 0!$, dan bila panggilan pada $n-1$ mengembalikan $(n-1)!$, baris 3 mengembalikan $n \cdot (n-1)! = n!$. Invarian gelung adalah induksi pada banyaknya iterasi; induksi pada n adalah argumen yang sama untuk rekursinya.

10.4.2 Ongkos masing-masing

Misalkan $T(n)$ waktu jalan `FACTORIAL-RECURSIVE(n)`. *Base case* memakan suatu konstanta c_0. Untuk $n \geq 1$, fungsi itu melakukan kerja konstan c (satu perbandingan, satu perkalian, satu panggilan, dan satu *return*) ditambah kerja panggilan rekursif pada $n-1$:

$$T(0) = c_0, \qquad T(n) = T(n-1) + c \quad (n \geq 1). \tag{10.4}$$

Persamaan (10.4) adalah sebuah *rekurens*, persamaan yang mendefinisikan suatu fungsi berdasarkan nilai-nilainya sendiri yang lebih kecil, sebagaimana Persamaan (10.1). Bila diuraikan selangkah demi selangkah:

$$T(n) = T(n-1) + c = T(n-2) + 2c = T(n-3) + 3c = \cdots = T(0) + nc \\ = c_0 + cn = \Theta(n).$$

Gelungnya melakukan $n - 1$ iterasi berongkos konstan, yang juga $\Theta(n)$. Keduanya berbeda dalam memori. Gelung memerlukan dua variabel berapa pun n-nya: memori tambahan $O(1)$. Rekursi menyimpan $n + 1$ *frame* di *stack* pada titik terdalamnya: memori tambahan $\Theta(n)$. Untuk faktorial, perbedaan ini dalam praktik tidak pernah berarti, karena n tidak mungkin berguna melampaui 20 dalam aritmetika 64 bit, dan bahkan dengan bilangan bulat besar, hasil kali sejuta digit dibatasi oleh perkaliannya, bukan oleh *stack*. Tetapi perbedaan itu nyata, dan Subbab 10.12 memperlihatkan betapa cepat ia menjadi penentu bagi rekursi yang lebih dalam.

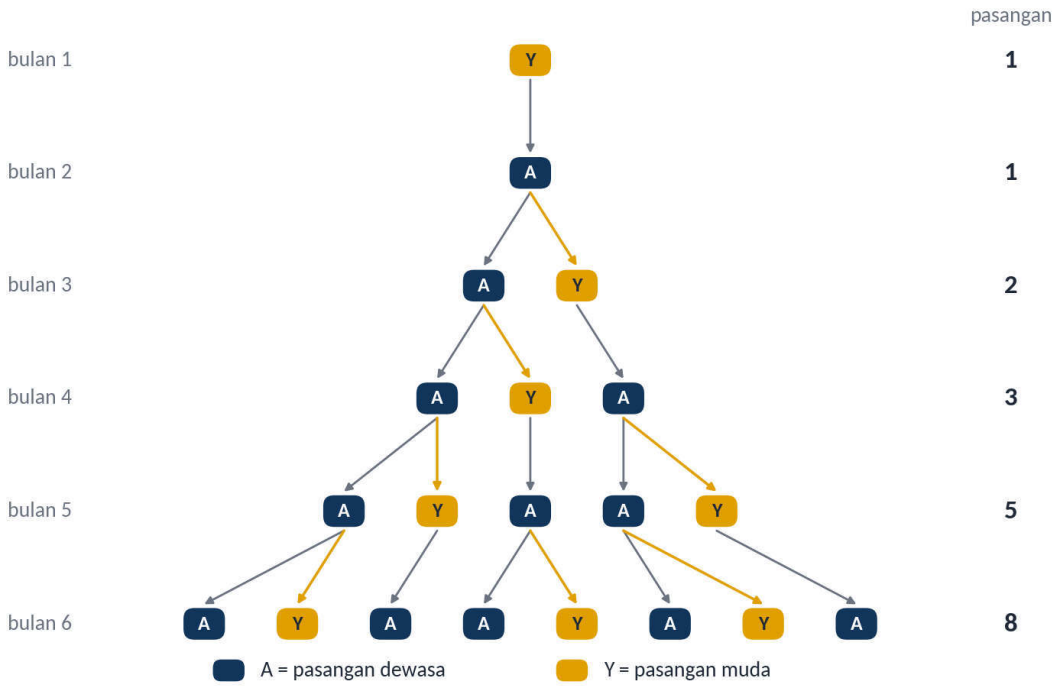
> **WAWASAN KUNCI:** *Waktu* sebuah rekursi adalah total kerja semua panggilannya. *Memori tambahan* sebuah rekursi adalah banyaknya *frame* terbesar yang berada di *stack pada saat yang sama*, dikali ukuran sebuah *frame*. Untuk faktorial keduanya sebanding dengan n. Subbab 10.5 memperlihatkan rekursi yang waktunya eksponensial sementara memorinya hanya linear, dan Subbab 10.7 rekursi yang waktu dan memorinya sama-sama logaritmik.

10.5 Fibonacci dan Kelinci-Kelincinya

10.5.1 Masalah kelinci

Pada 1202 Leonardo dari Pisa, yang kini dikenal sebagai Fibonacci, menerbitkan *Liber Abaci*, buku yang membawa angka Hindu–Arab beserta aritmetikanya kepada para pedagang Eropa. Di antara ratusan soal praktis di dalamnya ada satu soal tentang kelinci [5]. Seseorang menaruh sepasang kelinci yang baru lahir, seekor jantan dan seekor betina, di suatu tempat berpagar. Sepasang kelinci yang baru lahir memerlukan satu bulan untuk menjadi dewasa, dan sejak itu setiap pasangan dewasa melahirkan satu pasangan baru setiap bulan. Tidak ada kelinci yang mati. Ada berapa pasang kelinci pada akhir tahun?

Hitunglah bulan demi bulan, seperti pada **Gambar 10.3**, pada awal setiap bulan. Pada bulan 1 ada satu pasangan muda (Y). Pada bulan 2 pasangan itu telah menjadi dewasa (A), tetapi belum melahirkan: tetap satu pasang. Pada bulan 3 pasangan dewasa itu melahirkan satu pasangan muda: dua pasang. Pada bulan 4 pasangan pertama melahirkan lagi, dan pasangan yang lahir pada bulan 3 menjadi dewasa: tiga pasang. Pada bulan 5 kedua pasangan dewasa melahirkan: lima pasang.



Gambar 10.3 Enam bulan pertama kelinci Fibonacci. Panah abu-abu menunjukkan sepasang kelinci yang terus hidup ke bulan berikutnya; panah kuning menunjukkan pasangan dewasa yang melahirkan pasangan muda baru.

Aturan di balik gambar itu adalah sebagai berikut: pasangan yang hidup pada bulan n adalah semua pasangan yang hidup pada bulan $n-1$ (tidak ada yang mati) ditambah satu pasangan baru untuk setiap pasangan yang sudah dewasa pada bulan $n-1$, dan pasangan dewasa pada bulan $n-1$ tepat sama dengan pasangan-pasangan yang hidup pada bulan $n-2$. Jadi banyaknya pasangan dalam suatu bulan adalah jumlah banyaknya pasangan pada dua bulan sebelumnya. Bagian pertama **Kode 10.5** di bawah ini mengikuti pasangan dewasa dan muda sepanjang tahun dan mencapai 144 pasang pada bulan 12, jawaban Fibonacci. Dalam notasi modern, dengan $F(0) = 0$ ditambahkan di depan demi kemudahan,

$$F(0) = 0, \qquad F(1) = 1, \qquad F(n) = F(n-1) + F(n-2) \quad (n \geq 2), \tag{10.5}$$

yang menghasilkan *barisan Fibonacci* 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, … Banyaknya pasangan pada bulan n adalah $F(n)$. Barisan itu dinamai menurut Fibonacci pada abad kesembilan belas oleh Édouard Lucas, orang yang sama yang menciptakan Menara Hanoi. Barisan ini muncul pada spiral kepala bunga matahari dan buah pinus, dalam analisis algoritma Euklides (Subbab 10.6.4), dalam pohon pencarian seimbang yang akan dipelajari kemudian, dan, seperti yang akan segera kita lihat, dalam ongkos perhitungan naifnya sendiri.

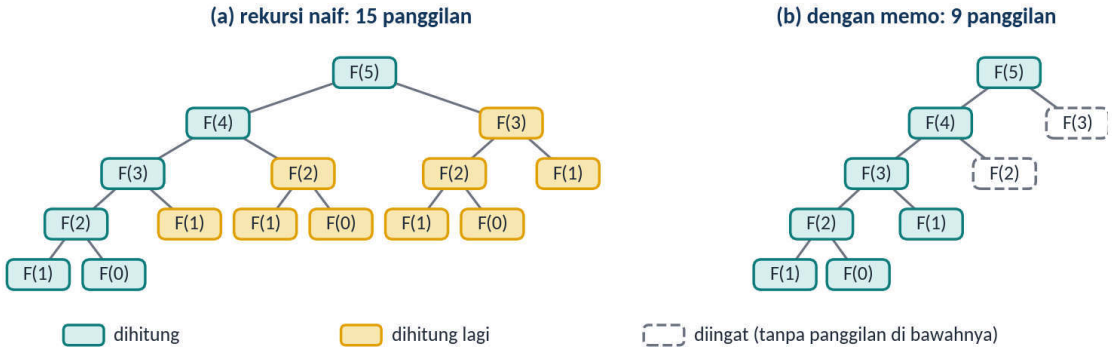
10.5.2 Rekursi naif, dan mengapa ia meledak

Persamaan (10.5) langsung dapat diterjemahkan menjadi kode:

```
long long fibNaive(int n) {
    if (n <= 1) return n;                          // F(0) = 0, F(1) = 1
    return fibNaive(n - 1) + fibNaive(n - 2);
}
```

Fungsi ini memiliki dua *base case* dan satu *recursive case* yang melakukan *dua* panggilan rekursif. Fungsi ini benar; induksi pada n, dengan dua *base case* dan hipotesis bahwa kedua panggilan yang lebih kecil benar, membuktikannya dalam dua baris. Namun sebagai cara menghitung bilangan Fibonacci, fungsi ini termasuk algoritma terburuk dalam buku ini. **Gambar 10.4**(a)

memperlihatkan sebabnya. Panggilan-panggilan `fibNaive(5)` membentuk sebuah pohon: `F(5)` memanggil `F(4)` dan `F(3)`; `F(4)` memanggil `F(3)` dan `F(2)`; dan seterusnya sampai daun-daun `F(1)` dan `F(0)`. Pohon itu memiliki 15 simpul, satu per panggilan, dan banyak di antaranya mengulangi pekerjaan yang sudah dilakukan: `F(3)` dihitung dua kali, `F(2)` tiga kali, `F(1)` lima kali. Semua simpul kuning adalah usaha yang terbuang.



Gambar 10.4 Panggilan-panggilan untuk menghitung $F(5)$. (a) Rekursi naif: 15 panggilan, sebagian besar menghitung ulang nilai yang sudah diketahui. (b) Dengan memoisasi: 9 panggilan; nilai yang sudah ada di tabel dikembalikan tanpa panggilan apa pun di bawahnya.

Berapa banyak panggilan yang dilakukan `fibNaive(n)` secara umum? Misalkan banyaknya $C(n)$. Panggilan dengan $n \le 1$ tidak melakukan panggilan lebih lanjut, jadi $C(0) = C(1) = 1$. Panggilan dengan $n \ge 2$ adalah satu panggilan ditambah semua panggilan yang dilakukan kedua anaknya:

$$C(0) = C(1) = 1, \qquad C(n) = C(n-1) + C(n-2) + 1 \quad (n \ge 2). \tag{10.6}$$

Persamaan (10.6) hampir sama dengan rekurens Fibonacci itu sendiri, dan penyelesaiannya adalah

$$C(n) = 2F(n+1) - 1. \tag{10.7}$$

Buktinya dengan induksi. Untuk $n = 0$: $2F(1) - 1 = 1$; untuk $n = 1$: $2F(2) - 1 = 2 - 1 = 1$. Untuk $n \ge 2$, dengan menganggap Persamaan (10.7) berlaku untuk $n-1$ dan $n-2$:

$$\begin{aligned} C(n) &= (2F(n) - 1) + (2F(n-1) - 1) + 1 = 2(F(n) + F(n-1)) - 1 \\ &= 2F(n+1) - 1. \end{aligned}$$

Jadi fungsi naif itu melakukan panggilan kira-kira dua kali sebanyak jawaban yang dihitungnya. Itu bencana, karena bilangan Fibonacci tumbuh secara eksponensial. Misalkan $\varphi = (1 + \sqrt{5})/2 \approx 1.618$, *rasio emas*, akar positif dari $x^2 = x + 1$. Maka

$$F(n) \ge \varphi^{n-2} \quad \text{untuk setiap } n \ge 1. \tag{10.8}$$

Bukti dengan induksi: $F(1) = 1 \ge \varphi^{-1} \approx 0.618$ dan $F(2) = 1 \ge \varphi^0 = 1$; dan untuk $n \ge 3$, $F(n) = F(n-1) + F(n-2) \ge \varphi^{n-3} + \varphi^{n-4} = \varphi^{n-4}(\varphi + 1) = \varphi^{n-4}\varphi^2 = \varphi^{n-2}$, dengan memakai $\varphi + 1 = \varphi^2$. (Argumen yang sama dengan $\le$ dan pangkat $n-1$ menghasilkan $F(n) \le \varphi^{n-1}$.) Jadi $C(n) = 2F(n+1) - 1 = \Theta(\varphi^n)$: setiap kenaikan n sebesar satu mengalikan kerjanya dengan kira-kira 1,618, dan setiap kenaikan sebesar lima mengalikannya dengan kira-kira $\varphi^5 \approx 11$.

Bagian kedua **Kode 10.5** menghitung panggilan-panggilan itu secara nyata dan mencetak $2F(n+1) - 1$ di sampingnya; bagian ketiga menghitung berapa kali setiap $F(k)$ dihitung di dalam

`fibNaive(6)`. Bilangan-bilangan itu, 1, 1, 2, 3, 5, 8 untuk $F(6)$ sampai $F(1)$, sendiri adalah bilangan Fibonacci: rekursi naif menghitung $F(k)$ tepat $F(n-k+1)$ kali untuk $k \geq 1$.

10.5.3 Memoisasi: ingat apa yang sudah dihitung

Obat bagi penghitungan ulang adalah ingatan. Simpanlah sebuah tabel `memo` dengan satu entri untuk setiap nilai n, yang semula bertanda "belum diketahui". Sebelum menghitung $F(n)$, lihat tabelnya; bila nilainya ada, kembalikan nilai itu; bila tidak, hitunglah secara rekursif, *simpan*, lalu kembalikan. Teknik ini disebut *memoisasi* (*memoization*, dari "memo", catatan untuk diri sendiri), dan rekursi dengan memoisasi disebut *dynamic programming top-down* (pemrograman dinamis dari atas ke bawah). **Pseudokode 10.2** menerapkannya pada Persamaan (10.5).

Pseudokode 10.2 Bilangan Fibonacci dengan rekursi bermemo.

```
FIB-MEMO(n, memo)                    // memo[0..n] initialised to "unknown"
 1.  if n ≤ 1
 2.      return n
 3.  if memo[n] is known
 4.      return memo[n]               // remembered: no recursion below
 5.  memo[n] ← FIB-MEMO(n − 1, memo) + FIB-MEMO(n − 2, memo)
 6.  return memo[n]
```

Gambar 10.4(b) memperlihatkan dampaknya pada $F(5)$. Tulang punggung kiri $F(5), F(4), F(3), F(2), F(1)$ menghitung setiap nilai untuk pertama kalinya; setiap anak kanan menemukan nilainya sudah ada di tabel (simpul bergaris putus-putus) dan langsung kembali. Setiap $F(k)$ dengan $2 \leq k \leq n$ dihitung tepat sekali, dengan ongkos dua panggilan ke anak-anaknya, sehingga total panggilannya $1 + 2(n-1) = 2n - 1$ untuk $n \geq 1$: 9 untuk $n = 5$, 179 untuk $n = 90$. Waktunya $\Theta(n)$, demikian pula memorinya (tabel dan *stack*). Rekursinya tetap persis Persamaan (10.5); hanya ingatannya yang baru.

> **WAWASAN KUNCI:** Sebuah rekursi eksponensial sering kali eksponensial hanya karena ia menyelesaikan submasalah yang sama berulang-ulang. Bila banyaknya submasalah yang *berbeda* kecil (di sini, $n + 1$), menyimpan setiap jawaban saat pertama kali ditemukan mengubah algoritma eksponensial menjadi algoritma polinomial. Itulah seluruh gagasan *dynamic programming*.

Python menyediakan memoisasi siap pakai: hiasi fungsi naif itu dengan `@functools.lru_cache(maxsize=None)` (atau `@functools.cache`) dan setiap hasilnya otomatis diingat, di dalam *doubly linked list* sirkular dengan *sentinel* yang diuraikan di Subbab 9.12.3.

10.5.4 Bottom-up: tanpa rekursi sama sekali

Memoisasi menghitung nilai-nilai dalam urutan yang diminta rekursinya, yang untuk Fibonacci adalah $F(1), F(2), \ldots, F(n)$. Sekalian saja kita isi tabelnya dalam urutan itu secara langsung, dengan gelung, dimulai dari *base case*-nya:

```
F[0] = 0;  F[1] = 1;
for (int i = 2; i <= n; ++i) F[i] = F[i - 1] + F[i - 2];
```

Inilah *dynamic programming bottom-up* (dari bawah ke atas): tanpa rekursi, tanpa *stack*, waktu $\Theta(n)$. Lihat sekali lagi badan gelung itu: ia hanya pernah membaca dua entri terakhir tabel. Jadi tabelnya tidak diperlukan; dua variabel, $a = F(i)$ dan $b = F(i+1)$, sudah cukup, dan setiap langkah mengganti pasangan (a, b) dengan $(b, a + b)$. Itulah `fibTwoVars` di **Kode 10.5**: waktu $\Theta(n)$ dan memori $O(1)$.

10.5.5 Rekursi ekor: gelung yang ditulis sebagai rekursi

Gelung yang memperbarui pasangan itu dapat juga ditulis sebagai rekursi, dengan melewatkan pasangannya sebagai parameter:

```
long long fibAux(long long x, long long y, int n) {   // x = F(i), y = F(i+1)
    if (n == 0) return x;
    return fibAux(y, x + y, n - 1);
}
long long fibTail(int n) { return fibAux(0, 1, n); }
```

Setiap panggilan menggeser pasangan itu satu langkah sepanjang barisan dan mengurangi banyaknya langkah yang tersisa; setelah n langkah, `x` berisi $F(n)$. Panggilan rekursifnya adalah *hal terakhir* yang dilakukan `fibAux`: hasilnya dikembalikan tanpa diubah, tanpa penjumlahan atau perkalian yang menunggunya. Panggilan semacam itu disebut *tail call* (panggilan ekor), dan fungsi yang semua panggilan rekursifnya adalah *tail call* disebut *tail recursive* (rekursif ekor). `fibNaive` tidak rekursif ekor (ia menjumlahkan hasil kedua panggilannya), begitu pula `factRec` (ia mengalikan). `fibAux` melakukan $n+1$ panggilan, linear seperti gelungnya, dan Subbab 10.11 memperlihatkan bahwa *compiler* dapat mengubahnya menjadi persis gelung itu.

10.5.6 Fast doubling: $O(\log n)$

Kita dapat berbuat lebih baik daripada linear. Bilangan Fibonacci memenuhi sepasang *identitas penggandaan* (*doubling identities*):

$$F(2k) = F(k)(2F(k+1) - F(k)), \qquad F(2k+1) = F(k)^2 + F(k+1)^2. \tag{10.9}$$

Berikut satu cara melihat asal-usulnya. Langkah $(F(k), F(k+1)) \to (F(k+1), F(k+2))$ adalah pemetaan linear, sehingga dapat ditulis dengan matriks 2×2 $Q = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, dan dengan induksi pada n

$$Q^n = \begin{pmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{pmatrix} \quad (n \geq 1).$$

Karena $Q^{2k} = Q^k Q^k$, kalikan ruas kanan dengan dirinya sendiri untuk $n = k$ dan bandingkan entri-entrinya: entri kanan atas memberi $F(2k) = F(k+1)F(k) + F(k)F(k-1) = F(k)(F(k+1) + F(k-1))$, dan dengan menyubstitusikan $F(k-1) = F(k+1) - F(k)$ kita peroleh identitas pertama; entri kiri atas memberi $F(2k+1) = F(k+1)^2 + F(k)^2$, identitas kedua. (Periksa dengan $k = 3$: $F(6) = 2 \cdot (2 \cdot 3 - 2) = 8$ dan $F(7) = 2^2 + 3^2 = 13$. Benar.)

Identitas-identitas itu memberikan $F(n)$ dan $F(n+1)$ dari $F(k)$ dan $F(k+1)$ dengan $k = \lfloor n/2 \rfloor$: bila $n = 2k$, pasangan $(F(2k), F(2k+1))$ langsung diperoleh dari Persamaan (10.9); bila $n = 2k+1$, pasangannya $(F(2k+1), F(2k) + F(2k+1))$. Jadi `fibPair(n)` di **Kode 10.5** melakukan satu panggilan rekursif pada $\lfloor n/2 \rfloor$ dan beberapa perkalian, dan argumennya terbagi dua di setiap tingkat. Fungsi itu melakukan $\lfloor \lg n \rfloor + 2$ panggilan untuk $n \geq 1$ (8 untuk $n = 90$), dan berjalan dalam waktu dan *stack* $\Theta(\log n)$, bila setiap perkalian bilangan yang muat dalam satu *word* mesin dihitung sebagai satu langkah [6].

10.5.7 Bentuk tertutup Binet

Akhirnya, ada sebuah bentuk tertutup, yang diterbitkan Jacques Binet pada 1843 dan sudah dikenal sebelumnya oleh de Moivre dan Euler [7]. Dengan $\varphi = (1+\sqrt{5})/2$ dan $\psi = (1-\sqrt{5})/2 \approx -0.618$, dua akar dari $x^2 = x + 1$,

$$F(n) = \frac{\varphi^n - \psi^n}{\sqrt{5}}. \tag{10.10}$$

Buktinya lagi-lagi induksi dua baris. Untuk $n = 0$ ruas kanannya $0/\sqrt{5} = 0$, dan untuk $n = 1$ ruas kanannya $\sqrt{5}/\sqrt{5} = 1$. Bila rumus itu berlaku untuk $n - 1$ dan $n - 2$, maka, karena $\varphi^n = \varphi^{n-1} + \varphi^{n-2}$ (kalikan $\varphi^2 = \varphi + 1$ dengan φ^{n-2}) dan hal serupa berlaku untuk ψ, ruas kanan untuk n sama dengan jumlah ruas kanan untuk $n - 1$ dan $n - 2$, yaitu $F(n-1) + F(n-2) = F(n)$. Karena $\psi^n/\sqrt{5}$ terletak di antara -0.5 dan 0.5 untuk setiap $n \geq 0$, $F(n)$ tidak lain adalah $\varphi^n/\sqrt{5}$ *yang dibulatkan ke bilangan bulat terdekat*, dan itulah yang dihitung `fibBinet` dengan satu panggilan `pow`.

Metode ini yang tercepat, dan metode ini salah. Sebuah `double` membawa 53 bit signifikan, kira-kira 16 digit desimal, dan φ sendiri sudah dibulatkan. Galat φ^n bertambah besar seiring n, dan begitu galat itu melampaui setengah, pembulatannya jatuh pada bilangan bulat yang salah. Bagian terakhir **Kode 10.5** mencari n pertama tempat hal itu terjadi.

10.5.8 Keenam cara dalam satu program

Kode 10.5 menyatukan semuanya: tabel kelinci, rekursi naif beserta banyaknya panggilan, banyaknya submasalah yang diulang, keenam cara menghitung $F(90)$, dan pencarian kegagalan pertama rumus Binet.

Kode 10.5 `03_fibonacci.cpp`: kelinci-kelinci, dan $F(n)$ dengan rekursi naif, memoisasi, tabel *bottom-up*, dua variabel, rekursi ekor, *fast doubling*, dan rumus Binet.

```cpp
// 03_fibonacci.cpp -- Fibonacci's rabbits, and F(n) computed six ways:
// naive recursion, memoized recursion, bottom-up table, two variables,
// tail recursion, fast doubling, and Binet's closed form.
#include <cmath>
#include <cstdio>
#include <utility>
#include <vector>
using namespace std;
typedef long long ll;

ll naiveCalls = 0;
ll fibNaive(int n) {
    ++naiveCalls;
    if (n <= 1) return n;  // F(0) = 0, F(1) = 1
    return fibNaive(n - 1) + fibNaive(n - 2);
}

vector<int> computed;  // computed[k]: times F(k) was asked
ll fibCounted(int n) {
    ++computed[n];
    if (n <= 1) return n;
    return fibCounted(n - 1) + fibCounted(n - 2);
}

ll memoCalls = 0;
ll fibMemo(int n, vector<ll> &memo) {  // memo[k] == -1: not known yet
    ++memoCalls;
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];  // remembered: no recursion
    memo[n] = fibMemo(n - 1, memo) + fibMemo(n - 2, memo);
    return memo[n];
}

ll fibTable(int n) {  // bottom-up, the whole table
    if (n <= 1) return n;
    vector<ll> F(n + 1);
    F[0] = 0;
    F[1] = 1;
    for (int i = 2; i <= n; ++i) F[i] = F[i - 1] + F[i - 2];
    return F[n];
}
```

Kode 10.5 (Lanjutan) *03_fibonacci.cpp: kelinci-kelinci, dan F(n) dengan rekursi naif, memoisasi, tabel bottom-up, dua variabel, rekursi ekor, fast doubling, dan rumus Binet.*

```
ll fibTwoVars(int n) {  // bottom-up, only two values kept
    ll a = 0, b = 1;    // a = F(i), b = F(i+1)
    for (int i = 0; i < n; ++i) {
        ll next = a + b;
        a = b;
        b = next;
    }
    return a;
}

ll tailCalls = 0;
ll fibAux(ll x, ll y, int n) {  // x = F(i), y = F(i+1), n steps left
    ++tailCalls;
    if (n == 0) return x;
    return fibAux(y, x + y, n - 1);  // tail call: nothing left to do
}
ll fibTail(int n) { return fibAux(0, 1, n); }

ll doublingCalls = 0;
pair<ll, ll> fibPair(int n) {  // returns (F(n), F(n+1))
    ++doublingCalls;
    if (n == 0) return {0, 1};
    pair<ll, ll> h = fibPair(n / 2);  // (F(k), F(k+1)), k = n / 2
    ll a = h.first, b = h.second;
    ll c = a * (2 * b - a);  // F(2k)   = F(k) (2F(k+1) - F(k))
    ll d = a * a + b * b;    // F(2k+1) = F(k)^2 + F(k+1)^2
    if (n % 2 == 0) return {c, d};
    return {d, c + d};
}
ll fibDoubling(int n) { return fibPair(n).first; }

ll fibBinet(int n) {  // round(phi^n / sqrt 5)
    double phi = (1 + sqrt(5.0)) / 2;
    return llround(pow(phi, n) / sqrt(5.0));
}
```

Kode 10.5 (Lanjutan) *03_fibonacci.cpp: kelinci-kelinci, dan F(n) dengan rekursi naif, memoisasi, tabel bottom-up, dua variabel, rekursi ekor, fast doubling, dan rumus Binet.*

```
int main() {
    printf("Fibonacci's rabbits (pairs at the start of each month)\n");
    printf("month  adult  young  total\n");
    ll adult = 0, young = 1;
    for (int month = 1; month <= 12; ++month) {
        printf("%5d %6lld %6lld %6lld\n", month, adult, young, adult + young);
        ll born = adult;  // every adult pair has a pair
        adult += young;   // last month's young grow up
        young = born;
    }

    printf("\nNaive recursion: calls made to compute F(n)\n");
    printf(" n       F(n)      calls  2F(n+1)-1\n");
    int ns[] = {0, 1, 2, 3, 4, 5, 6, 10, 15, 20, 25, 30};
    for (int n : ns) {
        naiveCalls = 0;
        ll v = fibNaive(n);
        printf("%2d %10lld %10lld %10lld\n", n, v, naiveCalls,
               2 * fibTwoVars(n + 1) - 1);
    }

    computed.assign(7, 0);
    fibCounted(6);
    printf("\nInside fibNaive(6), F(k) is computed this many times:\n");
    for (int k = 6; k >= 0; --k) printf("  F(%d): %d\n", k, computed[k]);

    int n = 90;
    vector<ll> memo(n + 1, -1);
    ll m = fibMemo(n, memo);
    ll t = fibTail(n);
    ll d = fibDoubling(n);
    printf("\nF(%d) six ways\n", n);
    printf("  memoized recursion : %lld  (%lld calls)\n", m, memoCalls);
    printf("  bottom-up table    : %lld\n", fibTable(n));
    printf("  two variables      : %lld\n", fibTwoVars(n));
    printf("  tail recursion     : %lld  (%lld calls)\n", t, tailCalls);
    printf("  fast doubling      : %lld  (%lld calls)\n", d, doublingCalls);
    printf("  Binet, double      : %lld\n", fibBinet(n));

    int k = 0;
    while (fibBinet(k) == fibTwoVars(k)) ++k;
    printf("\nBinet's formula in double precision is exact up to F(%d);\n", k - 1);
    printf("F(%d) = %lld, but it gives %lld\n", k, fibTwoVars(k), fibBinet(k));
    return 0;
}
```

```
Fibonacci's rabbits (pairs at the start of each month)
month  adult  young  total
    1      0      1      1
    2      1      0      1
    3      1      1      2
    4      2      1      3
    5      3      2      5
    6      5      3      8
    7      8      5     13
    8     13      8     21
    9     21     13     34
   10     34     21     55
   11     55     34     89
   12     89     55    144

Naive recursion: calls made to compute F(n)
 n       F(n)      calls  2F(n+1)-1
 0          0          1          1
 1          1          1          1
 2          1          3          3
 3          2          5          5
 4          3          9          9
 5          5         15         15
 6          8         25         25
10         55        177        177
15        610       1973       1973
20       6765      21891      21891
25      75025     242785     242785
30     832040    2692537    2692537

Inside fibNaive(6), F(k) is computed this many times:
  F(6): 1
  F(5): 1
  F(4): 2
  F(3): 3
  F(2): 5
  F(1): 8
  F(0): 5

F(90) six ways
  memoized recursion : 2880067194370816120  (179 calls)
  bottom-up table    : 2880067194370816120
  two variables      : 2880067194370816120
  tail recursion     : 2880067194370816120  (91 calls)
  fast doubling      : 2880067194370816120  (8 calls)
  Binet, double      : 2880067194370824704

Binet's formula in double precision is exact up to F(70);
F(71) = 308061521170129, but it gives 308061521170130
```

Setiap klaim dalam subbab ini terdapat pada keluaran itu. Tabel kelinci mencapai 144 pada bulan 12, dengan $F(n-1)$ pasangan dewasa dan $F(n-2)$ pasangan muda pada bulan n. Banyaknya panggilan rekursi naif yang terukur cocok dengan Persamaan (10.7) di setiap baris, sampai 2.692.537 panggilan untuk $F(30) = 832{,}040$. Di dalam `fibNaive(6)`, banyaknya penghitungan ulang sendiri adalah bilangan Fibonacci. Kelima metode eksak sepakat bahwa $F(90) = 2{,}880{,}067{,}194{,}370{,}816{,}120$, dengan 179 panggilan untuk memoisasi $(2 \cdot 90 - 1)$, 91 untuk rekursi ekor, dan 8 untuk *fast doubling*. Rumus Binet dengan presisi `double` meleset 8.584 pada $n = 90$ dan hanya eksak sampai $F(70)$: pada $n = 71$ ia menjawab 308.061.521.170.130 alih-alih 308.061.521.170.129. (Program Java `Fibonacci.java` dan program Python `03_fibonacci.py` mencetak baris-baris yang sama, termasuk kegagalan yang sama pada 71: ketiganya memakai aritmetika `double` IEEE 754 yang sama dan `pow` yang berperilaku benar.) Pelajarannya umum:

bentuk tertutup yang dievaluasi dalam bilangan titik mengambang hanyalah hampiran, dan jawaban bilangan bulat yang eksak memerlukan aritmetika bilangan bulat yang eksak.

10.5.9 Diukur: panggilan dan waktu

Berapa lama semua ini berlangsung? Program `04_fib_bench.cpp`, `FibBench.java`, dan `04_fib_bench.py` di folder Code mengukur waktu rekursi naif untuk $n = 20,25, \ldots ,40$ dan metode-metode lainnya untuk $n = 90$ (dirata-ratakan atas banyak kali jalan, karena satu kali jalan terlalu singkat untuk diukur). Bilangan-bilangan di bawah ini berasal dari satu kali jalan nyata pada mesin yang dipakai untuk buku ini, sebuah server Linux dua inti yang dipakai bersama; hasil Anda akan berbeda, tetapi bentuknya tidak. Pertama C++ (g++ 13.3 dengan `-O2`):

```
Naive recursion
  n           F(n)        calls     time (ms)
 20           6765        21891         0.016
 25          75025       242785         0.150
 30         832040      2692537         1.502
 35        9227465     29860703        16.958
 40      102334155    331160281       196.942

F(90), average of 100000 runs
  memoized recursion :    888.4 ns
  two variables      :     47.2 ns
  tail recursion     :     28.0 ns
  fast doubling      :     16.1 ns
  Binet, double      :     15.5 ns
```

lalu Java (javac 21, JIT HotSpot, dengan satu putaran pemanasan sebelum metode-metode cepat diukur):

```
Naive recursion
  n           F(n)        calls     time (ms)
 20           6765        21891         0.316
 25          75025       242785         0.369
 30         832040      2692537         3.964
 35        9227465     29860703        44.404
 40      102334155    331160281       523.244

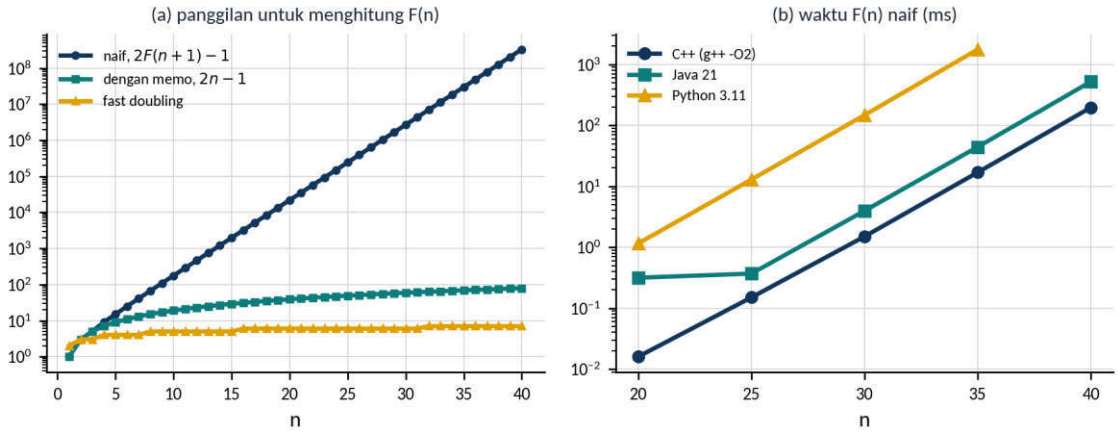
F(90), average of 100000 runs
  memoized recursion :    582.2 ns
  two variables      :     13.9 ns
  tail recursion     :    246.9 ns
  fast doubling      :     52.0 ns
  Binet, double      :     14.6 ns
```

dan Python 3.11 (rekursi naif berhenti pada $n = 35$, karena $n = 40$ akan memakan waktu sekitar 20 detik):

```
Naive recursion
  n           F(n)        calls     time (ms)
 20           6765        21891         1.156
 25          75025       242785        12.928
 30         832040      2692537       147.673
 35        9227465     29860703      1769.737

F(90), average of 20000 runs
  memoized recursion :  11122.2 ns
  two variables      :   2074.0 ns
  tail recursion     :   4318.0 ns
  fast doubling      :   1116.2 ns
  Binet, double      :    184.2 ns
```

Gambar 10.5(a) memetakan ketiga banyaknya panggilan pada skala logaritmik: banyaknya panggilan naif, $2F(n+1)-1$, berupa garis lurus, yang merupakan rupa pertumbuhan eksponensial pada skala seperti itu, sementara $2n-1$ dan $\lfloor \lg n \rfloor + 2$ menempel di dasar. **Gambar 10.5**(b) memetakan waktu naif yang diukur: tiga garis lurus yang hampir sejajar, masing-masing naik dengan faktor sekitar 11 setiap lima langkah n, tepat φ^5. Ketiga bahasa berbeda dengan faktor konstan (di sini C++ kira-kira 2,5 kali lebih cepat daripada Java dan kira-kira 100 kali lebih cepat daripada CPython), tetapi *kemiringan*-nya, yang ditentukan oleh algoritmanya, sama untuk ketiganya.



Gambar 10.5 (a) Banyaknya panggilan untuk menghitung $F(n)$ dengan ketiga metode rekursif, skala logaritmik. (b) Waktu rekursi naif yang diukur dalam C++, Java, dan Python, September 2026, skala logaritmik.

Tabel 10.1 merangkum metode-metode itu. Waktu yang diukur untuk $F(90)$ layak dicermati, karena mengandung tiga kejutan kecil. Memoisasi adalah metode linear eksak yang paling lambat di ketiga bahasa, padahal ia hanya melakukan 179 panggilan: sebagian besar waktunya habis untuk mengalokasikan dan mengisi tabel. Rekursi ekor di C++ setidaknya secepat gelungnya, karena (Subbab 10.11) GCC mengubahnya *menjadi* gelung; di Java ia kira-kira 17 kali lebih lambat daripada gelung dan di Python kira-kira dua kali lebih lambat, karena baik JVM maupun CPython tidak melakukannya. Dan di C++, *fast doubling* dan rumus Binet sama-sama memakan sekitar 16 ns: pada $n = 90$ metode logaritmik hanya memerlukan 8 panggilan, dan `pow` pun tidak gratis. Analisis asimtotik memberi tahu metode mana yang menang ketika n membesar; hanya pengukuran yang memberi tahu apa yang terjadi pada n tertentu.

Tabel 10.1 Tujuh cara menghitung $F(n)$, dengan setiap operasi pada *word* mesin dihitung satu langkah.

Metode	Panggilan atau langkah	Waktu	Memori tambahan	Eksak?
Rekursi naif	$2F(n+1)-1$ panggilan	$\Theta(\varphi^n)$	*stack* $\Theta(n)$	ya
Rekursi bermemo	$2n-1$ panggilan	$\Theta(n)$	tabel dan *stack* $\Theta(n)$	ya
Tabel *bottom-up*	$n-1$ penjumlahan	$\Theta(n)$	tabel $\Theta(n)$	ya
Dua variabel	n penjumlahan	$\Theta(n)$	$O(1)$	ya
Rekursi ekor	$n+1$ panggilan	$\Theta(n)$	*stack* $\Theta(n)$, $O(1)$ bila *compiler* menghapus *tail call*-nya	ya
Fast doubling	$\lfloor \lg n \rfloor + 2$ panggilan	$\Theta(\log n)$	*stack* $\Theta(\log n)$	ya
Binet, `double`	satu `pow`	$O(1)$	$O(1)$	hanya sampai $F(70)$

Satu catatan lagi: nilai-nilai dalam subbab ini berhenti pada $F(92) = 7{,}540{,}113{,}804{,}746{,}346{,}429$, bilangan Fibonacci terbesar yang muat dalam bilangan bulat bertanda 64 bit. Di luar itu, semua metode eksak memerlukan bilangan bulat besar (`BigInteger` di Java, `int` di Python), dan ongkos sebuah perkalian lalu tumbuh seiring banyaknya digit; itulah sebabnya pustaka-pustaka yang serius memakai *fast doubling* dengan perkalian cepat [6].

10.6 Faktor Persekutuan Terbesar Euklides

10.6.1 Masalah dan gagasannya

Faktor persekutuan terbesar (FPB, *greatest common divisor*) dua bilangan bulat tak negatif a dan b, yang tidak keduanya nol, ditulis $\gcd(a,b)$, adalah bilangan bulat terbesar yang membagi keduanya. Misalnya, pembagi 228 adalah 1, 2, 3, 4, 6, 12, 19, 38, 57, 76, 114, dan 228, pembagi 90 adalah 1, 2, 3, 5, 6, 9, 10, 15, 18, 30, 45, dan 90, dan bilangan terbesar yang ada di kedua daftar adalah 6, jadi $\gcd(228,90) = 6$. Mendaftar pembagi itu lambat untuk bilangan besar. Sekitar 300 SM, *Elemen* karya Euklides (Buku VII, Proposisi 1 dan 2) menguraikan metode yang jauh lebih baik [8], yang oleh Knuth disebut "kakek semua algoritma, karena ia algoritma nontrivial tertua yang bertahan sampai hari ini" [9]. Dalam bentuk modern, metode itu bertumpu pada satu persamaan:

$$\gcd(a,0) = a, \qquad \gcd(a,b) = \gcd(b,\ a \bmod b) \quad (b > 0), \tag{10.11}$$

dengan $a \bmod b$ adalah sisa pembagian a oleh b (`a % b` di ketiga bahasa kita, untuk operan tak negatif).

Mengapa Persamaan (10.11) **benar?** Tulis $a = qb + r$ dengan hasil bagi $q = \lfloor a/b \rfloor$ dan sisa $r = a \bmod b$, $0 \le r < b$. Bila sebuah bilangan d membagi a dan b, ia membagi $a - qb = r$; jadi setiap pembagi persekutuan a dan b adalah pembagi persekutuan b dan r. Sebaliknya, bila d membagi b dan r, ia membagi $qb + r = a$; jadi setiap pembagi persekutuan b dan r adalah pembagi persekutuan a dan b. Kedua pasangan itu memiliki pembagi persekutuan yang *persis sama*, sehingga yang terbesar pun sama. Dan $\gcd(a,0) = a$ karena setiap bilangan membagi 0, sehingga pembagi persekutuan a dan 0 tidak lain adalah pembagi a, yang terbesar adalah a sendiri.

Mengapa ia berhenti? Argumen kedua berubah dari b menjadi $a \bmod b$, yang benar-benar lebih kecil daripada b dan tidak pernah negatif. Sebuah bilangan bulat tak negatif tidak dapat

berkurang selamanya, sehingga setelah berhingga banyak langkah argumen kedua bernilai 0 dan *base case*-nya menjawab. Itulah aturan 2 dari Subbab 10.2.5, dengan ukuran "argumen kedua".

10.6.2 Dengan tangan, dan sebagai gambar

Berikut gcd(228,90), satu pembagian per baris:

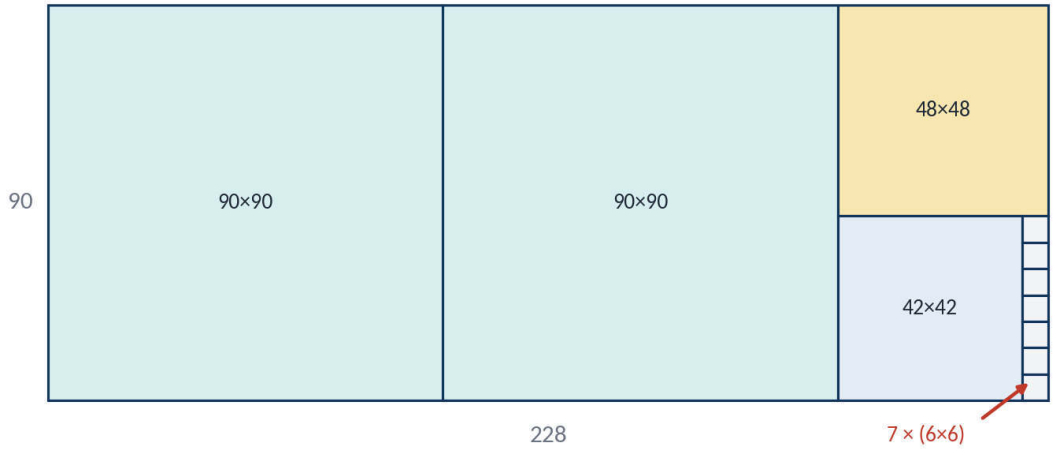
$$228 = 2 \cdot 90 + 48 \qquad \Rightarrow \gcd(228,90) = \gcd(90,48)$$

$$90 = 1 \cdot 48 + 42 \qquad \Rightarrow \gcd(90,48) = \gcd(48,42)$$

$$48 = 1 \cdot 42 + 6 \qquad \Rightarrow \gcd(48,42) = \gcd(42,6)$$

$$42 = 7 \cdot 6 + 0 \qquad \Rightarrow \gcd(42,6) = \gcd(6,0) = 6$$

Jawabannya adalah sisa terakhir yang bukan nol, 6, setelah empat pembagian. **Gambar 10.6** memperlihatkan perhitungan yang sama sebagai geometri, yang lebih dekat dengan cara Euklides memikirkannya. Ambil persegi panjang 228×90 dan potong persegi-persegi terbesar yang mungkin: dua persegi 90×90, menyisakan jalur 48×90. Dari jalur itu potong satu persegi 48×48, menyisakan 48×42; lalu satu persegi 42×42, menyisakan 6×42; dan jalur itu tepat tertutup oleh tujuh persegi 6×6. Sisi persegi terkecil, 6, mengukur kedua sisi persegi panjang semula sebanyak bilangan bulat kali ($228 = 38 \cdot 6$, $90 = 15 \cdot 6$), dan tidak ada persegi yang lebih besar yang mampu.



Gambar 10.6 Algoritma Euklides pada 228 dan 90 sebagai pemotongan persegi dari persegi panjang: setiap pembagian membuang sebanyak mungkin persegi bersisi pendek; persegi terakhir, 6×6, tepat menutup sisanya, jadi $\gcd(228,90) = 6$.

10.6.3 Programnya

Pseudokode 10.3 memberikan bentuk rekursif dan iteratifnya. Panggilan rekursifnya adalah *tail call*: hasilnya dikembalikan tanpa diubah. Karena itulah bentuk iteratifnya begitu mudah ditulis: ia mengganti "panggil dirimu dengan $(b, a \bmod b)$" dengan "isikan $(a, b) \leftarrow (b, a \bmod b)$ dan kembali ke atas".

Pseudokode 10.3 Algoritma Euklides, rekursif dan iteratif.

```
GCD-RECURSIVE(a, b)
 1.  if b = 0
 2.      return a
 3.  return GCD-RECURSIVE(b, a mod b)     // a tail call

GCD-ITERATIVE(a, b)
 1.  while b ≠ 0
 2.      (a, b) ← (b, a mod b)
 3.  return a
```

Kode 10.6 menelusuri dua perhitungan, membandingkan kedua bentuk pada beberapa pasangan, mencari pasangan di bawah 100 yang memerlukan pembagian terbanyak, dan menjalankan algoritmanya pada bilangan-bilangan Fibonacci yang berurutan.

Kode 10.6 `05_gcd.cpp`: algoritma Euklides secara rekursif dan iteratif, ditelusuri, beserta kasus terburuknya.

```cpp
// 05_gcd.cpp -- Euclid's algorithm, recursive and iterative, a traced run,
// and Lame's worst case: consecutive Fibonacci numbers.
#include <cstdio>

long gcdRec(long a, long b) {
    if (b == 0) return a;     // gcd(a, 0) = a
    return gcdRec(b, a % b);  // gcd(a, b) = gcd(b, a mod b)
}

long gcdIter(long a, long b) {
    while (b != 0) {
        long r = a % b;
        a = b;
        b = r;
    }
    return a;
}

long gcdTraced(long a, long b) {
    if (b == 0) {
        std::printf("  gcd(%ld, 0): b = 0, the answer is %ld\n", a, a);
        return a;
    }
    std::printf("  gcd(%ld, %ld): %ld = %ld * %ld + %ld\n", a, b, a, a / b, b,
                a % b);
    return gcdTraced(b, a % b);
}

int steps(long a, long b) {  // number of divisions performed
    return b == 0 ? 0 : 1 + steps(b, a % b);
}

int main() {
    std::printf("Traced:\n");
    gcdTraced(228, 90);
    gcdTraced(1071, 462);

    long pairs[][2] = {{228, 90}, {1071, 462}, {90, 228},
                       {17, 5},    {12, 0},     {144, 89}};
    std::printf("\n     a     b   recursive  iterative  divisions\n");
    for (auto &p : pairs)
        std::printf("%6ld %5ld %11ld %10ld %10d\n", p[0], p[1], gcdRec(p[0], p[1]),
                    gcdIter(p[0], p[1]), steps(p[0], p[1]));
}
```

Kode 10.6 (Lanjutan) *05_gcd.cpp: algoritma Euklides secara rekursif dan iteratif, ditelusuri, beserta kasus terburuknya.*

```
    int best = 0;
    long bestA = 0, bestB = 0;
    for (long a = 1; a <= 100; ++a)
        for (long b = 1; b < a; ++b)
            if (steps(a, b) > best) {
                best = steps(a, b);
                bestA = a;
                bestB = b;
            }
    std::printf("\nAmong all 1 <= b < a <= 100, the most divisions is %d, "
                "first for gcd(%ld, %ld)\n", best, bestA, bestB);

    std::printf("\nConsecutive Fibonacci numbers:\n");
    long x = 1, y = 2;  // F(2), F(3)
    for (int k = 3; k <= 12; ++k) {
        std::printf("  gcd(F(%d), F(%d)) = gcd(%ld, %ld) = %ld "
                    "after %d divisions\n",
                    k + 1, k, x + y, y, gcdRec(x + y, y), steps(x + y, y));
        long z = x + y;
        x = y;
        y = z;
    }
    return 0;
}
```

```
Traced:
  gcd(228, 90): 228 = 2 * 90 + 48
  gcd(90, 48): 90 = 1 * 48 + 42
  gcd(48, 42): 48 = 1 * 42 + 6
  gcd(42, 6): 42 = 7 * 6 + 0
  gcd(6, 0): b = 0, the answer is 6
  gcd(1071, 462): 1071 = 2 * 462 + 147
  gcd(462, 147): 462 = 3 * 147 + 21
  gcd(147, 21): 147 = 7 * 21 + 0
  gcd(21, 0): b = 0, the answer is 21

     a     b   recursive  iterative  divisions
   228    90           6          6          4
  1071   462          21         21          3
    90   228           6          6          5
    17     5           1          1          3
    12     0          12         12          0
   144    89           1          1         10

Among all 1 <= b < a <= 100, the most divisions is 9, first for gcd(89, 55)

Consecutive Fibonacci numbers:
  gcd(F(4), F(3)) = gcd(3, 2) = 1 after 2 divisions
  gcd(F(5), F(4)) = gcd(5, 3) = 1 after 3 divisions
  gcd(F(6), F(5)) = gcd(8, 5) = 1 after 4 divisions
  gcd(F(7), F(6)) = gcd(13, 8) = 1 after 5 divisions
  gcd(F(8), F(7)) = gcd(21, 13) = 1 after 6 divisions
  gcd(F(9), F(8)) = gcd(34, 21) = 1 after 7 divisions
  gcd(F(10), F(9)) = gcd(55, 34) = 1 after 8 divisions
  gcd(F(11), F(10)) = gcd(89, 55) = 1 after 9 divisions
  gcd(F(12), F(11)) = gcd(144, 89) = 1 after 10 divisions
  gcd(F(13), F(12)) = gcd(233, 144) = 1 after 11 divisions
```

Perhatikan baris ketiga tabel: dengan argumen yang urutannya "terbalik", gcd(90,228), pembagian pertamanya $90 = 0 \cdot 228 + 90$, yang sekadar menukar keduanya, dan algoritma berlanjut seperti sebelumnya dengan satu pembagian tambahan. Argumennya tidak perlu diurutkan lebih dahulu. Baris $\gcd(12,0) = 12$ hanyalah *base case*-nya.

10.6.4 Berapa banyak pembagian? Teorema Lamé

Pencarian atas semua pasangan di bawah 100 menemukan bahwa pembagian terbanyak, 9, diperlukan oleh gcd(89,55), dua bilangan Fibonacci yang berurutan, dan blok terakhir keluaran memperlihatkan bahwa gcd$(F(k+1), F(k))$ selalu memerlukan tepat $k-1$ pembagian. Itu bukan kebetulan. Membagi $F(k+1)$ dengan $F(k)$ memberi hasil bagi 1 dan sisa $F(k+1) - F(k) = F(k-1)$, sehingga algoritma itu menuruni barisan Fibonacci satu langkah per pembagian, penurunan paling lambat yang mungkin. Pada 1844 Gabriel Lamé menjadikannya teorema pertama yang pernah dibuktikan tentang waktu jalan sebuah algoritma [10]:

> **TEOREMA (Lamé, 1844).** Bila $a > b \geq 1$ dan algoritma Euklides melakukan k pembagian pada (a, b), maka $a \geq F(k+2)$ dan $b \geq F(k+1)$.

Bukti dengan induksi pada k. Bila $k = 1$, maka $b \geq 1 = F(2)$ dan $a > b$ memberi $a \geq 2 = F(3)$. Bila $k \geq 2$, pembagian pertama mengubah (a, b) menjadi (b, r) dengan $r = a \bmod b \geq 1$ (kalau tidak, ia berhenti setelah satu pembagian), dan pada (b, r) algoritma melakukan $k-1$ pembagian. Menurut hipotesis induksi, $b \geq F(k+1)$ dan $r \geq F(k)$. Karena $a \geq b + r$ (hasil baginya paling sedikit 1), $a \geq F(k+1) + F(k) = F(k+2)$. ▫

Gabungkan teorema itu dengan Persamaan (10.8): bila diperlukan k pembagian maka $b \geq F(k+1) \geq \varphi^{k-1}$, sehingga

$$k \;\leq\; 1 + \log_{\varphi} b \;\approx\; 1 + 4.785 \log_{10} b. \tag{10.12}$$

Menurut Persamaan (10.12), banyaknya pembagian paling banyak sekitar lima kali banyaknya digit desimal bilangan yang lebih kecil: $O(\log b)$. Untuk bilangan di bawah 100, batas itu adalah $1 + \log_{\varphi} 99 \approx 10.5$, dan kasus terburuk yang sebenarnya 9. Untuk dua bilangan 64 bit, batasnya paling banyak 93. Algoritma Euklides cukup cepat untuk dipakai di dalam aritmetika kriptografi pada bilangan ribuan digit.

Kedalaman rekursinya adalah banyaknya pembagian ditambah satu, sehingga bentuk rekursifnya memakai *stack* $O(\log b)$: tidak pernah lebih dari seratus *frame* untuk bilangan 64 bit. Inilah rekursi yang boleh Anda tulis tanpa berpikir dua kali. Knuth [9] juga menganalisis banyaknya pembagian *rata-rata*; nilainya sekitar $0.843 \ln b + 1.47$.

10.7 Pemangkatan Cepat

10.7.1 Mengurangi satu: n perkalian

Untuk menghitung m^n dengan n bilangan bulat tak negatif, definisi "kalikan m dengan dirinya sendiri n kali" menghasilkan gelung berisi n perkalian. Ada pula definisi rekursif yang sama sederhananya:

$$m^0 = 1, \qquad m^n = m \cdot m^{n-1} \quad (n \geq 1) \tag{10.13}$$

Bila ditulis sebagai kode, Persamaan (10.13) menghasilkan fungsi rekursif, `powerNaive` di **Kode 10.7**, yang bentuknya sama dengan faktorial: $n+1$ panggilan, n perkalian, waktu dan *stack* $\Theta(n)$. Untuk 2^{10} itu berarti 10 perkalian; untuk $7^{1{,}000{,}000{,}000}$, satu miliar.

10.7.2 Membagi dua: sekitar $2\lg n$ perkalian

Rekursi yang jauh lebih baik diperoleh dari pengkuadratan. Bila n genap, $m^n = (m^{n/2})^2$; bila n ganjil, $m^n = (m^{(n-1)/2})^2 \cdot m$. Dengan pembagian bilangan bulat, kedua kasus itu berbunyi

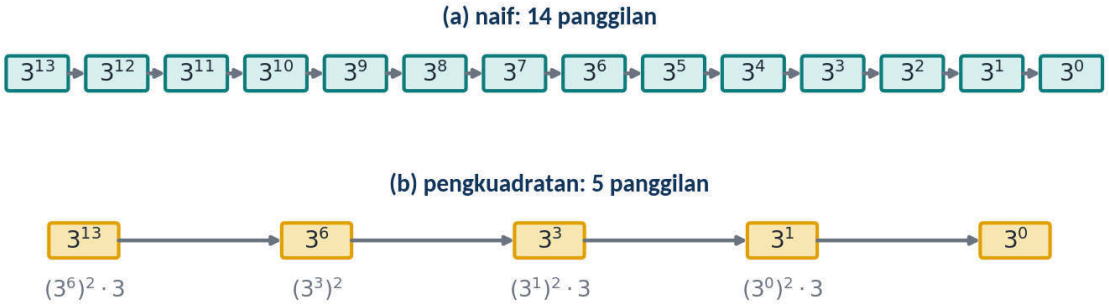
$$m^n = \begin{cases} 1 & \text{jika } n = 0, \\ (m^{\lfloor n/2 \rfloor})^2 & \text{jika } n \text{ genap}, \\ (m^{\lfloor n/2 \rfloor})^2 \cdot m & \text{jika } n \text{ ganjil}. \end{cases} \tag{10.14}$$

Panggilan rekursif dalam Persamaan (10.14) dilakukan *sekali*, pada separuh pangkat, dan hasilnya dikuadratkan; melakukannya dua kali, seperti `power(m, n/2) * power(m, n/2)`, akan mengembalikan semua kerja yang telah dihemat (Soal Latihan 10.9 menanyakan sebabnya). **Pseudokode 10.4** menuliskannya.

Pseudokode 10.4 Pemangkatan dengan pengkuadratan berulang.

```
FAST-POWER(m, n)
 1.  if n = 0
 2.      return 1
 3.  half ← FAST-POWER(m, ⌊n/2⌋)          // one call on half the exponent
 4.  if n is even
 5.      return half · half
 6.  return half · half · m
```

Telusurilah 3^{13}. Karena $13 = 2 \cdot 6 + 1$, $3^{13} = (3^6)^2 \cdot 3$; $3^6 = (3^3)^2$; $3^3 = (3^1)^2 \cdot 3$; $3^1 = (3^0)^2 \cdot 3$; dan $3^0 = 1$. Dalam perjalanan naik kembali: $3^1 = 1 \cdot 1 \cdot 3 = 3$, $3^3 = 3 \cdot 3 \cdot 3 = 27$, $3^6 = 27 \cdot 27 = 729$, $3^{13} = 729 \cdot 729 \cdot 3 = 1{,}594{,}323$. Lima panggilan alih-alih empat belas, seperti diperlihatkan **Gambar 10.7**.



Gambar 10.7 Menghitung 3^{13}. (a) Mengurangi pangkat dengan satu: 14 panggilan. (b) Membagi dua pangkatnya: 5 panggilan; di bawah setiap panggilan tertera cara hasilnya dibangun dari panggilan berikutnya.

Pangkatnya terbagi dua di setiap panggilan sampai mencapai 0. Mulai dari $n \geq 1$, diperlukan $\lfloor \lg n \rfloor + 1$ kali pembagian dua untuk mencapai 0 (banyaknya digit biner n), sehingga

$$\text{panggilan}(n) = \lfloor \lg n \rfloor + 2 \quad (n \geq 1), \qquad \text{perkalian} \leq 2(\lfloor \lg n \rfloor + 1). \tag{10.15}$$

Dilihat dalam biner, metode ini terang benderang: $13 = 1101_2$, dan setiap tingkat rekursi menangani satu digit biner, mengkuadratkan untuk setiap digit dan mengalikan dengan m untuk setiap angka 1. Knuth menelusuri gagasan ini lebih dari dua ribu tahun ke belakang, sampai kepada sarjana India Piṅgala [9].

10.7.3 Pemangkatan modular

Bilangannya sendiri tumbuh terlalu cepat untuk berguna: $7^{1{,}000{,}000{,}000}$ memiliki sekitar 845 juta digit desimal. Yang sesungguhnya diperlukan kriptografi, *hashing* (Bab 15), dan banyak soal pemrograman adalah $m^n \bmod p$ untuk suatu modulus p. Karena $(x \cdot y) \bmod p = ((x \bmod p)(y \bmod p)) \bmod p$, kita boleh mereduksi setelah setiap perkalian, dan setiap nilai antara tetap di bawah p. Dengan $p = 1{,}000{,}000{,}007$, pilihan yang lazim, setiap hasil kali dua bilangan yang sudah direduksi berada di bawah $10^{18} < 2^{63}$ dan muat dalam bilangan bulat 64 bit. RSA, sistem kunci publik yang dipakai setiap hari dalam koneksi TLS peramban web, menghitung pangkat semacam itu dengan pangkat ratusan digit; tanpa pengkuadratan berulang, hal itu mustahil [3].

Kode 10.7 menelusuri 3^{13}, membandingkan kedua rekursi pada lima pangkat eksak, lalu menghitung $7^n \bmod 1{,}000{,}000{,}007$ untuk pangkat sampai 10^{18}.

Kode 10.7 `06_power.cpp`: pemangkatan dengan n perkalian dan dengan pengkuadratan berulang, beserta banyaknya panggilan, dan pemangkatan modular.

```
// 06_power.cpp -- m^n by n multiplications, and by repeated squaring,
// with call counts and modular exponentiation.
#include <cstdio>
typedef unsigned long long ull;

long long calls = 0;

ull powerNaive(ull m, int n) {  // m^n = m * m^(n-1)
    ++calls;
    if (n == 0) return 1;
    return m * powerNaive(m, n - 1);
}

ull powerFast(ull m, int n) {  // m^n = (m^(n/2))^2, times m if n is odd
    ++calls;
    if (n == 0) return 1;
    ull half = powerFast(m, n / 2);
    if (n % 2 == 0) return half * half;
    return half * half * m;
}

ull powTraced(ull m, int n, int depth) {
    if (n == 0) {
        std::printf("%*spow(%llu, 0) = 1\n", 4 * depth, "", m);
        return 1;
    }
    std::printf("%*spow(%llu, %d) needs pow(%llu, %d)\n", 4 * depth, "", m, n, m,
                n / 2);
    ull half = powTraced(m, n / 2, depth + 1);
    ull result = n % 2 == 0 ? half * half : half * half * m;
    if (n % 2 == 0)
        std::printf("%*spow(%llu, %d) = %llu^2 = %llu\n", 4 * depth, "", m, n, half,
                    result);
    else
        std::printf("%*spow(%llu, %d) = %llu^2 * %llu = %llu\n", 4 * depth, "", m,
                    n,
                    half, m, result);
    return result;
}

const ull MOD = 1000000007ULL;  // a prime; every product stays < 2^63
long long mcalls = 0;
```

Kode 10.7 (Lanjutan) *06_power.cpp: pemangkatan dengan n perkalian dan dengan pengkuadratan berulang, beserta banyaknya panggilan, dan pemangkatan modular.*

```
ull powerNaiveMod(ull m, long long n) {
    ++mcalls;
    if (n == 0) return 1;
    return m % MOD * powerNaiveMod(m, n - 1) % MOD;
}

ull powerFastMod(ull m, long long n) {
    ++mcalls;
    if (n == 0) return 1;
    ull half = powerFastMod(m, n / 2);
    ull sq = half * half % MOD;
    return n % 2 == 0 ? sq : sq * (m % MOD) % MOD;
}

int main() {
    std::printf("Tracing pow(3, 13):\n");
    powTraced(3, 13, 0);

    std::printf("\n m   n            naive  calls            fast  calls\n");
    int cases[][2] = {{2, 10}, {3, 13}, {2, 63}, {3, 40}, {7, 22}};
    for (auto &c : cases) {
        calls = 0;
        ull a = powerNaive(c[0], c[1]);
        long long ca = calls;
        calls = 0;
        ull b = powerFast(c[0], c[1]);
        std::printf("%2d %3d %20llu %6lld %20llu %6lld\n", c[0], c[1], a, ca, b,
                    calls);
    }

    std::printf("\n7^n mod 1000000007\n");
    std::printf("                    n       naive  calls         fast  calls\n");
    long long ns[] = {10, 100, 900, 1000000000LL, 10000000000000000000LL};
    for (long long n : ns) {
        mcalls = 0;
        ull b = powerFastMod(7, n);
        long long cb = mcalls;
        if (n <= 900) {  // deeper naive recursion: see Section 10.12
            mcalls = 0;
            ull a = powerNaiveMod(7, n);
            std::printf("%20lld %11llu %6lld %12llu %6lld\n", n, a, mcalls, b, cb);
        } else {
            std::printf("%20lld %11s %6s %12llu %6lld\n", n, "-", "-", b, cb);
        }
    }
    return 0;
```

Kode 10.7 (Lanjutan) *06_power.cpp: pemangkatan dengan n perkalian dan dengan pengkuadratan berulang, beserta banyaknya panggilan, dan pemangkatan modular.*

```
}
```

```
Tracing pow(3, 13):
pow(3, 13) needs pow(3, 6)
    pow(3, 6) needs pow(3, 3)
        pow(3, 3) needs pow(3, 1)
            pow(3, 1) needs pow(3, 0)
                pow(3, 0) = 1
            pow(3, 1) = 1^2 * 3 = 3
        pow(3, 3) = 3^2 * 3 = 27
    pow(3, 6) = 27^2 = 729
pow(3, 13) = 729^2 * 3 = 1594323

 m   n            naive  calls            fast  calls
 2  10                1024     11                1024      5
 3  13             1594323     14             1594323      5
 2  63  9223372036854775808     64  9223372036854775808      7
 3  40 12157665459056928801     41 12157665459056928801      7
 7  22  3909821048582988049     23  3909821048582988049      6

7^n mod 1000000007
                   n       naive  calls        fast  calls
                  10   282475249     11   282475249      5
                 100   946501044    101   946501044      8
                 900   928292332    901   928292332     11
          1000000000           -      -   312556845     31
 1000000000000000000           -      -   259616729     61
```

Banyaknya panggilan cocok dengan Persamaan (10.15) di setiap baris: 5 untuk $n = 10$ dan $n = 13$, 7 untuk $n = 63$ (enam digit biner, $\lfloor \lg 63 \rfloor = 5$), 31 untuk $n = 10^9$, dan 61 untuk $n = 10^{18}$. Metode naif memerlukan $n + 1$ panggilan dan bahkan tidak dicoba di atas $n = 900$: dengan semiliar panggilan ia akan memakan beberapa detik di C++, sedangkan di Java dan Python rekursinya akan *crash* jauh sebelumnya, karena alasan-alasan di Subbab 10.12 (di Python, 900 sedikit di bawah batas bawaan 1.000). Baris $2^{63} = 9{,}223{,}372{,}036{,}854{,}775{,}808$ hanya muat karena program itu memakai bilangan bulat 64 bit *tak bertanda*; `long long` bertanda berhenti pada $2^{63} - 1$. Fungsi bawaan Python `pow(7, n, 1000000007)` memakai metode yang sama dan memberikan hasil yang sama.

10.8 Menara Hanoi, Diselesaikan Secara Rekursif

10.8.1 Gagasan rekursifnya

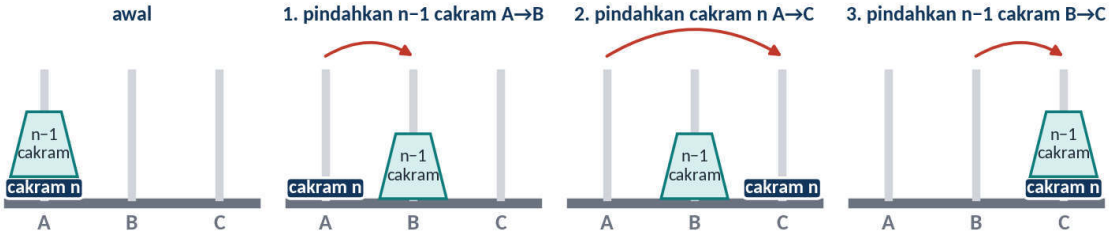
Subbab 5.2 memperkenalkan teka-teki Édouard Lucas dari tahun 1883 [11]: pindahkan menara berisi n cakram dari tiang A ke tiang C, satu cakram setiap kali, tanpa pernah meletakkan cakram di atas cakram yang lebih kecil, dengan tiang B sebagai pembantu. Ketika menelusuri penyelesaian tiga cakram di sana, kita mengamati bahwa langkah 1–3 membawa dua cakram kecil ke B, langkah 4 membawa cakram besar ke C, dan langkah 5–7 membawa kedua cakram kecil dari B ke atasnya. Menyelesaikan tiga cakram berarti menyelesaikan dua cakram dua kali, dengan satu langkah di antaranya.

Pengamatan itu *adalah* algoritma rekursifnya. Untuk memindahkan n cakram dari tiang `from` ke tiang `to`, dengan `via` sebagai pembantu:

1. pindahkan $n - 1$ cakram yang lebih kecil dari `from` ke `via`, dengan `to` sebagai pembantu;
2. pindahkan cakram terbesar, cakram n, dari `from` ke `to`;
3. pindahkan $n - 1$ cakram yang lebih kecil dari `via` ke `to`, dengan `from` sebagai pembantu.

Gambar 10.8 memperlihatkan ketiga langkah itu dengan $n - 1$ cakram yang lebih kecil digambar sebagai satu blok. Langkah yang terasa seperti curang adalah langkah 1 dan 3: “pindahkan $n - 1$

cakram" adalah masalah yang justru sedang kita coba selesaikan. Tetapi itu adalah contoh yang *lebih kecil* dari masalah tersebut, dan hanya itu yang diperlukan rekursi. Selama $n-1$ cakram itu berpindah, cakram n terletak di dasar sebuah tiang dan, karena yang terbesar, tidak pernah menghalangi langkah apa pun, sehingga masalah yang lebih kecil itu benar-benar teka-teki yang sama dengan satu cakram lebih sedikit. *Base case*-nya $n = 0$: tidak ada yang perlu dipindahkan. (**Gambar 10.8** dan Subbab 5.2 memakai $n = 1$, satu langkah, sebagai *base case*; keduanya benar, dan $n = 0$ membuat kodenya lebih pendek.)



Gambar 10.8 Penyelesaian rekursif Menara Hanoi: singkirkan $n-1$ cakram yang lebih kecil, pindahkan cakram terbesar, lalu pindahkan kembali $n-1$ cakram itu ke atasnya.

Pseudokode 10.5 Menara Hanoi secara rekursif.

```
HANOI(n, from, to, via)
 1.  if n = 0
 2.      return                          // base case: nothing to move
 3.  HANOI(n - 1, from, via, to)         // clear the n - 1 smaller disks onto via
 4.  move disk n from `from` to `to`
 5.  HANOI(n - 1, via, to, from)         // bring them back on top of disk n
```

10.8.2 Programnya

Kode 10.8 adalah **Pseudokode 10.5** dalam C++, dengan satu tambahan yang membuat program itu memeriksa dirinya sendiri: setiap tiang adalah `std::stack<int>`, *stack* dari Bab 5, dan `moveDisk` sungguh-sungguh melakukan *pop* cakram dari satu tiang dan *push* ke tiang lain, sambil mencetak `ILLEGAL MOVE` bila cakram itu tidak berada di puncak tiangnya atau akan mendarat di atas cakram yang lebih kecil. Program itu mencetak ketujuh langkah untuk tiga cakram, lalu menyelesaikan teka-teki untuk 1 sampai 20 cakram, memeriksa banyaknya langkah terhadap $2^n - 1$, mengukur rekursi terdalam, dan memastikan semua cakram berakhir di C.

Kode 10.8 `07_hanoi.cpp`: Menara Hanoi secara rekursif, dengan setiap langkah diperiksa pada tiga *stack* sungguhan.

```cpp
// 07_hanoi.cpp -- the Towers of Hanoi solved by recursion. Every move is also
// played on three real stacks, which check that no rule is ever broken.
#include <cstdio>
#include <stack>

std::stack<int> peg[3];  // peg[0] = A, peg[1] = B, peg[2] = C
long long moves = 0;
int depth = 0, maxDepth = 0;
bool printing = false;

void moveDisk(int disk, char from, char to) {
    std::stack<int> &src = peg[from - 'A'], &dst = peg[to - 'A'];
    if (src.empty() || src.top() != disk || (!dst.empty() && dst.top() < disk)) {
        std::printf("ILLEGAL MOVE of disk %d from %c to %c\n", disk, from, to);
        return;
    }
    src.pop();
    dst.push(disk);
    ++moves;
    if (printing)
        std::printf("%2lld. Move disk %d from rod %c to rod %c\n", moves, disk,
                    from, to);
}

// Move the n smallest disks from rod `from` to rod `to`, with rod `via` as helper.
void hanoi(int n, char from, char to, char via) {
    if (n == 0) return;  // base case: nothing to move
    ++depth;
    if (depth > maxDepth) maxDepth = depth;
    hanoi(n - 1, from, via, to);  // 1. clear the n-1 smaller disks onto via
    moveDisk(n, from, to);        // 2. move the largest disk
    hanoi(n - 1, via, to, from);  // 3. bring the n-1 disks back on top of it
    --depth;
}

void solve(int n) {
    for (auto &p : peg) p = std::stack<int>();
    for (int d = n; d >= 1; --d) peg[0].push(d);
    moves = 0;
    maxDepth = 0;
    hanoi(n, 'A', 'C', 'B');
}
```

Kode 10.8 (Lanjutan) *07_hanoi.cpp: Menara Hanoi secara rekursif, dengan setiap langkah diperiksa pada tiga stack sungguhan.*

```cpp
int main() {
    printing = true;
    std::printf("Three disks, from A to C:\n");
    solve(3);
    printing = false;

    std::printf("\n n      moves    2^n - 1  depth  all on C\n");
    for (int n = 1; n <= 20; ++n) {
        solve(n);
        bool done = (int)peg[2].size() == n && peg[0].empty() && peg[1].empty();
        std::printf("%2d %10lld %10lld %6d  %s\n", n, moves, (1LL << n) - 1,
                    maxDepth,
                    done ? "yes" : "NO");
    }
    unsigned long long legend = ~0ULL;  // 2^64 - 1, all 64 bits set
    std::printf("\nThe legend's 64 disks need %llu moves.\n", legend);
    return 0;
}
```

```
Three disks, from A to C:
 1. Move disk 1 from rod A to rod C
 2. Move disk 2 from rod A to rod B
 3. Move disk 1 from rod C to rod B
 4. Move disk 3 from rod A to rod C
 5. Move disk 1 from rod B to rod A
 6. Move disk 2 from rod B to rod C
 7. Move disk 1 from rod A to rod C

 n      moves    2^n - 1  depth  all on C
 1          1          1      1  yes
 2          3          3      2  yes
 3          7          7      3  yes
 4         15         15      4  yes
 5         31         31      5  yes
 6         63         63      6  yes
 7        127        127      7  yes
 8        255        255      8  yes
 9        511        511      9  yes
10       1023       1023     10  yes
11       2047       2047     11  yes
12       4095       4095     12  yes
13       8191       8191     13  yes
14      16383      16383     14  yes
15      32767      32767     15  yes
16      65535      65535     16  yes
17     131071     131071     17  yes
18     262143     262143     18  yes
19     524287     524287     19  yes
20    1048575    1048575     20  yes

The legend's 64 disks need 18446744073709551615 moves.
```

Ketujuh langkahnya persis sama dengan langkah di Subbab 5.2.1 (bandingkan Gambar 5.2), dan sama pula dengan langkah yang dibuat gelung tiga-*stack* di Subbab 5.2.3: metode rekursif dan iteratif menemukan satu-satunya penyelesaian terpendek yang sama. Tidak ada satu baris `ILLEGAL MOVE` pun dalam 2.097.130 langkah (total untuk 1 sampai 20 cakram), sebuah pemeriksaan eksperimental atas argumen di Subbab 10.8.1 bahwa cakram terbesar tidak pernah menghalangi. Kedalaman rekursinya sama dengan n: menara 20 cakram hanya memerlukan 20 *frame* sekaligus, walaupun ia membuat lebih dari sejuta langkah. Di Java (**Kode 10.9**) setiap tiang adalah `ArrayDeque<Integer>`; di Python (**Kode 10.10**) setiap tiang adalah list yang dipakai sebagai *stack*, seperti di Subbab 5.16.3.

Kode 10.9 `Hanoi.java`: Menara Hanoi secara rekursif dalam Java.

```java
// Hanoi.java -- the Towers of Hanoi solved by recursion. Every move is also
// played on three real stacks, which check that no rule is ever broken.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.List;

public class Hanoi {
    static List<ArrayDeque<Integer>> peg = new ArrayList<>();  // A, B, C
    static long moves = 0;
    static int depth = 0, maxDepth = 0;
    static boolean printing = false;

    static void moveDisk(int disk, char from, char to) {
        ArrayDeque<Integer> src = peg.get(from - 'A'), dst = peg.get(to - 'A');
        if (src.isEmpty() || src.peek() != disk
                || (!dst.isEmpty() && dst.peek() < disk)) {
            System.out.printf("ILLEGAL MOVE of disk %d from %c to %c%n", disk, from,
                              to);
            return;
        }
        src.pop();
        dst.push(disk);
        ++moves;
        if (printing)
            System.out.printf("%2d. Move disk %d from rod %c to rod %c%n", moves,
                              disk,
                              from, to);
    }

    // Move the n smallest disks from rod `from` to rod `to`, with `via` as helper.
    static void hanoi(int n, char from, char to, char via) {
        if (n == 0) return;  // base case: nothing to move
        ++depth;
        if (depth > maxDepth) maxDepth = depth;
        hanoi(n - 1, from, via, to);  // 1. clear the n-1 smaller disks onto via
        moveDisk(n, from, to);        // 2. move the largest disk
        hanoi(n - 1, via, to, from);  // 3. bring the n-1 disks back on top of it
        --depth;
    }
```

Kode 10.9 (Lanjutan) *Hanoi.java: Menara Hanoi secara rekursif dalam Java.*

```
    static void solve(int n) {
        peg.clear();
        for (int i = 0; i < 3; ++i) peg.add(new ArrayDeque<>());
        for (int d = n; d >= 1; --d) peg.get(0).push(d);
        moves = 0;
        maxDepth = 0;
        hanoi(n, 'A', 'C', 'B');
    }

    public static void main(String[] args) {
        printing = true;
        System.out.println("Three disks, from A to C:");
        solve(3);
        printing = false;

        System.out.println("\n n      moves    2^n - 1  depth  all on C");
        for (int n = 1; n <= 20; ++n) {
            solve(n);
            boolean done = peg.get(2).size() == n && peg.get(0).isEmpty()
                           && peg.get(1).isEmpty();
            System.out.printf("%2d %10d %10d %6d  %s%n", n, moves, (1L << n) - 1,
                              maxDepth,
                              done ? "yes" : "NO");
        }
        long legend = ~0L;  // all 64 bits set: 2^64 - 1 unsigned
        System.out.printf("%nThe legend's 64 disks need %s moves.%n",
                          Long.toUnsignedString(legend));
    }
}
```

Kode 10.10 `07_hanoi.py`: Menara Hanoi secara rekursif dalam Python.

```
# 07_hanoi.py -- the Towers of Hanoi solved by recursion. Every move is also
# played on three real stacks, which check that no rule is ever broken.

peg = {"A": [], "B": [], "C": []}  # each peg is a list used as a stack
moves = 0
depth = max_depth = 0
printing = False


def move_disk(disk, frm, to):
    global moves
    src, dst = peg[frm], peg[to]
    if not src or src[-1] != disk or (dst and dst[-1] < disk):
        print(f"ILLEGAL MOVE of disk {disk} from {frm} to {to}")
        return
    dst.append(src.pop())
    moves += 1
    if printing:
        print(f"{moves:2d}. Move disk {disk} from rod {frm} to rod {to}")


def hanoi(n, frm, to, via):
    """Move the n smallest disks from rod frm to rod to, with rod via as helper."""
    global depth, max_depth
    if n == 0:
        return  # base case: nothing to move
    depth += 1
    max_depth = max(max_depth, depth)
    hanoi(n - 1, frm, via, to)  # 1. clear the n-1 smaller disks onto via
    move_disk(n, frm, to)       # 2. move the largest disk
    hanoi(n - 1, via, to, frm)  # 3. bring the n-1 disks back on top of it
    depth -= 1


def solve(n):
    global moves, max_depth
    peg["A"], peg["B"], peg["C"] = list(range(n, 0, -1)), [], []
    moves = max_depth = 0
    hanoi(n, "A", "C", "B")
```

Kode 10.10 (Lanjutan) *07_hanoi.py: Menara Hanoi secara rekursif dalam Python.*

```
def main():
    global printing
    printing = True
    print("Three disks, from A to C:")
    solve(3)
    printing = False

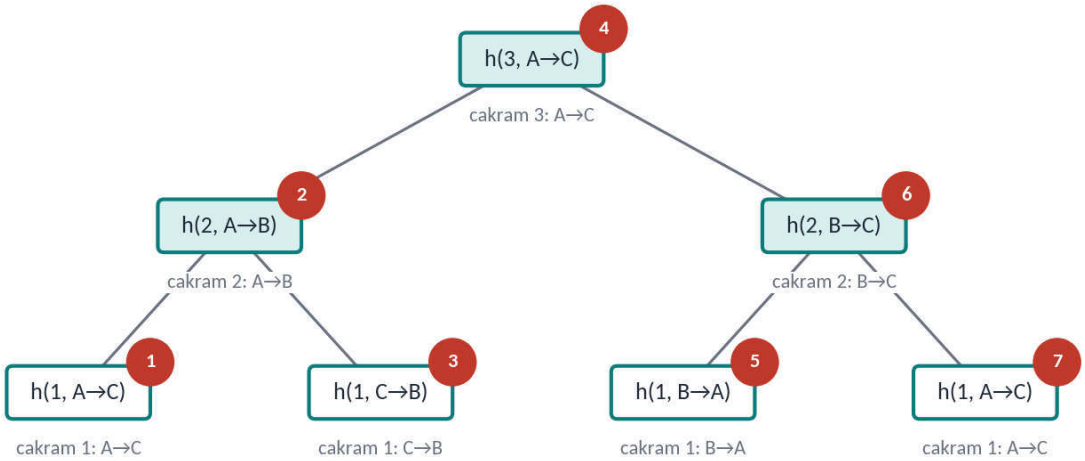
    print("\n n      moves    2^n - 1  depth  all on C")
    for n in range(1, 21):
        solve(n)
        done = len(peg["C"]) == n and not peg["A"] and not peg["B"]
        print(f"{n:2d} {moves:10d} {2**n - 1:10d} {max_depth:6d}  "
              f"{'yes' if done else 'NO'}")
    print(f"\nThe legend's 64 disks need {2**64 - 1} moves.")


if __name__ == "__main__":
    main()
```

Keduanya mencetak keluaran **Kode 10.8** baris demi baris.

10.8.3 Urutan panggilan: sebuah pohon rekursi

Gambar 10.9 menggambar setiap panggilan `hanoi(3, A, C, B)` sebagai simpul sebuah pohon; panggilan dengan $n = 0$, yang langsung kembali, tidak digambar. Setiap simpul memiliki dua panggilan rekursif dari baris 3 dan 5 sebagai anak-anaknya, dan membuat langkahnya sendiri (baris 4) *di antara* keduanya. Angka merah memberikan urutan terjadinya langkah-langkah: pertama semua yang ada di subpohon kiri, lalu langkah simpul itu sendiri, lalu semua yang ada di subpohon kanan. Membaca pohon itu dalam urutan tersebut (Bab 11 menyebutnya *penelusuran in-order*) menghasilkan daftar langkah keluaran tadi: cakram 1 A→C, cakram 2 A→B, cakram 1 C→B, cakram 3 A→C, cakram 1 B→A, cakram 2 B→C, cakram 1 A→C.



Gambar 10.9 Pohon rekursi `hanoi(3, A, C, B)`; `h(k, X→Y)` memindahkan k cakram dari X ke Y. Setiap panggilan membuat langkahnya sendiri di antara langkah-langkah kedua anaknya; angka merah memberikan urutan langkah-langkahnya.

10.8.4 Mengapa tepat $2^n - 1$ langkah, dan mengapa tidak kurang

Misalkan $M(n)$ banyaknya langkah yang dibuat `HANOI` untuk n cakram. Baris 4 membuat satu langkah dan baris 3 serta 5 masing-masing membuat $M(n-1)$:

$$M(0) = 0, \qquad M(n) = 2M(n-1) + 1 \quad (n \geq 1). \tag{10.16}$$

Ini adalah rekurens dari Subbab 5.2.2, yang kini dibaca langsung dari programnya. Penyelesaiannya $M(n) = 2^n - 1$, dan inilah bukti induksi yang ditunda oleh Subbab 5.2.2. *Base case*: $M(0) = 0 = 2^0 - 1$. Langkah induktif: bila $M(n-1) = 2^{n-1} - 1$, maka $M(n) = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1$. ▫

Bahwa algoritma rekursif itu membuat $2^n - 1$ langkah belum menunjukkan bahwa tidak ada algoritma yang lebih cerdik yang dapat membuat lebih sedikit. Berikut buktinya bahwa tidak ada. Misalkan $L(n)$ banyaknya langkah terkecil yang dengannya barisan langkah sah *apa pun* dapat memindahkan n cakram dari satu tiang ke tiang lain. Kita buktikan $L(n) \geq 2^n - 1$ dengan induksi. Untuk $n = 0$ hal itu jelas. Untuk $n \geq 1$, perhatikan sembarang penyelesaian dan langkah *pertama* cakram terbesar, cakram n. Pada saat itu cakram n adalah cakram teratas di tiangnya, sehingga setiap cakram yang lebih kecil berada di tempat lain, dan cakram n berpindah ke tiang yang tidak memuat cakram yang lebih kecil, sehingga semua $n-1$ cakram yang lebih kecil bertumpuk di tiang ketiga. Membawa mereka ke sana dari tiang awal adalah pemindahan $n-1$ cakram (cakram n terletak di bawah mereka sepanjang waktu dan tidak pernah berpengaruh), yang memerlukan paling sedikit $L(n-1)$ langkah. Sekarang perhatikan langkah *terakhir* cakram n, ke tiang tujuan. Sesudahnya, $n-1$ cakram yang lebih kecil, yang pada saat itu semuanya berada di tiang ketiga

karena alasan yang sama, harus dibawa ke atas cakram n: paling sedikit $L(n-1)$ langkah lagi. Bersama paling sedikit satu langkah cakram n, setiap penyelesaian membuat paling sedikit $2L(n-1)+1 \geq 2(2^{n-1}-1)+1 = 2^n - 1$ langkah. ▫

Jadi algoritma rekursif itu *optimal*: ongkos eksponensialnya milik masalahnya, bukan milik algoritmanya. Itulah perbedaan hakikinya dengan Fibonacci naif. Kedua rekursi melakukan panggilan sebanyak eksponensial, dan keduanya berbentuk "dua panggilan rekursif ditambah sedikit kerja". Tetapi panggilan-panggilan Fibonacci naif saling tumpang tindih (kedua submasalahnya berbagi hampir segalanya), sehingga mengingat jawaban menghapus pemborosannya. Kedua panggilan `HANOI` memindahkan *cakram yang berbeda di antara tiang yang berbeda*: tidak ada yang perlu diingat, dan setiap langkah dari $2^n - 1$ langkah itu diperlukan. Untuk 64 cakram dalam legenda, baris terakhir program memberikan banyaknya langkah, 18,446,744,073,709,551,615, sekitar 585 miliar tahun dengan satu langkah per detik (Subbab 5.2.2), dan tidak ada algoritma yang akan pernah lebih baik. Graham, Knuth, dan Patashnik [12] membuka buku mereka *Concrete Mathematics* dengan teka-teki ini dan memakainya untuk memperkenalkan rekurens secara umum.

10.9 Rekursi pada Larik dan Linked List

10.9.1 Dua cara membuat larik lebih kecil

Larik $A[0..n-1]$ tidak tampak rekursif, tetapi ia menjadi rekursif begitu kita memilih cara membuatnya lebih kecil. Ada dua pilihan baku.

- **Kupas satu elemen.** Larik berisi n elemen adalah $n-1$ elemen pertamanya diikuti elemen terakhirnya (atau elemen pertamanya diikuti sisanya). *Base case*-nya larik kosong. Cara ini menghasilkan rekursi dengan $n+1$ panggilan dan kedalaman n, seperti faktorial.
- **Belah dua.** Larik $A[lo..hi]$ adalah separuh kirinya $A[lo..mid]$ diikuti separuh kanannya $A[mid+1..hi]$, dengan $mid = lo + \lfloor (hi - lo)/2 \rfloor$. *Base case*-nya satu elemen (atau tidak ada). Cara ini menghasilkan rekursi sedalam $\lceil \lg n \rceil + 1$, seperti *binary search* (Subbab 2.5.5) dan *merge sort* (Subbab 3.8).

Bagian pertama **Kode 10.11** menerapkan keduanya pada larik kecil berisi banyaknya peminjaman harian di Pustaka Ceria. `sumArr(A, n)` mengupas elemen terakhir: jumlah $A[0..n-1]$ adalah jumlah $A[0..n-2]$ ditambah $A[n-1]$, dan jumlah larik kosong adalah 0. `maxArr(A, lo, hi)` membelah dua: maksimum keseluruhan adalah yang lebih besar di antara maksimum kedua belahannya, dan maksimum satu elemen adalah elemen itu sendiri. `searchArr` adalah *linear search* yang ditulis secara rekursif: lihat $A[i]$, dan bila bukan kuncinya, cari mulai dari $i+1$. `binSearch` adalah *binary search* rekursif dari Subbab 2.5.5. `reverseArr` menukar kedua ujung lalu membalik apa yang ada di antaranya. `isPalindrome` menanyakan apakah sebuah kata terbaca sama dari belakang: kedua huruf ujungnya harus sama, dan kata di antaranya harus palindrom; kata dengan nol atau satu huruf selalu palindrom.

10.9.2 Linked list adalah struktur rekursif

Subbab 7.6.2 memberikan definisi rekursif sebuah list: *sebuah list adalah kosong, atau sebuah simpul pertama yang diikuti sebuah list*. Setiap rekursi atas *singly linked list* mengikuti definisi itu: *base case*-nya pointer nol, dan *recursive case*-nya menangani simpul pertama lalu melakukan rekursi pada `p->next`. Bagian kedua **Kode 10.11** mengumpulkan yang paling berguna.

- `length(p)` dan `sumList(p)`: 0 untuk list kosong, selain itu 1 + length(p.next), atau p.data + sum(p.next).
- `contains(p, key)`: *false* untuk list kosong; selain itu *true* bila simpul pertama memuat kuncinya, dan bila tidak, jawaban untuk sisanya. Operator `||` berhenti pada kecocokan pertama.
- `printForward(p)` mencetak simpul pertama *lalu* melakukan rekursi; `printBackward(p)` melakukan rekursi *lebih dahulu* dan mencetak sesudahnya. Satu perubahan urutan itu mencetak list dari belakang, tanpa pointer `prev` (Bab 9) dan tanpa membalik apa pun (Subbab 7.6.6): setiap panggilan menunggu di *stack*, memegang simpulnya, sampai panggilan-panggilan untuk semua simpul sesudahnya selesai mencetak, persis seperti *frame-frame* pada **Gambar 10.2**. *Stack* itu memainkan peran tautan mundur yang tidak ada.
- `copyList(p)` membuat salinan yang mandiri: simpul baru yang memuat `p->data` dan yang `next`-nya adalah salinan sisanya.
- `removeAll(p, key)` menghapus setiap simpul yang memuat `key` dan mengembalikan simpul pertama yang baru: ia lebih dahulu membersihkan sisa list, lalu menghapus simpul pertama (mengembalikan sisa yang sudah bersih) atau menautkannya kembali ke sisa yang sudah bersih itu. Karena fungsi itu *mengembalikan head* yang baru, pemanggil menulis `head = removeAll(head, 40);`, dan kasus "simpul pertama harus dibuang" tidak memerlukan kode khusus; bandingkan dengan kasus-kasus khusus `deleteKey` di Subbab 7.6.5.
- `destroy(p)` membebaskan sisa list lebih dahulu, lalu simpul pertama, dan menghitung simpul-simpul yang dibebaskan.

Kode 10.11 `08_arrays_lists.cpp`: rekursi atas larik (mengupas satu elemen atau membelah dua) dan atas *singly linked list* (simpul pertama dan sisanya).

```cpp
// 08_arrays_lists.cpp -- recursion on arrays (a first element and the rest, or two
// halves) and on singly linked lists (a first node and the rest of the list).
#include <cstdio>
#include <cstring>

// ---------- arrays: A[lo..hi] ----------
int sumArr(const int A[], int n) {       // A[0..n-1]
    if (n == 0) return 0;                // the empty array
    return sumArr(A, n - 1) + A[n - 1];  // the first n-1, plus the last
}

int maxArr(const int A[], int lo, int hi) {  // A[lo..hi], lo <= hi
    if (lo == hi) return A[lo];
    int mid = lo + (hi - lo) / 2;
    int left = maxArr(A, lo, mid), right = maxArr(A, mid + 1, hi);
    return left > right ? left : right;
}

int searchArr(const int A[], int n, int i, int key) {  // first index >= i
    if (i == n) return -1;                             // ran off the end: absent
    if (A[i] == key) return i;
    return searchArr(A, n, i + 1, key);
}

int binSearch(const int A[], int lo, int hi, int key) {  // A sorted ascending
    if (lo > hi) return -1;
    int mid = lo + (hi - lo) / 2;
    if (A[mid] == key) return mid;
    if (key < A[mid]) return binSearch(A, lo, mid - 1, key);
    return binSearch(A, mid + 1, hi, key);
}

void reverseArr(int A[], int lo, int hi) {
    if (lo >= hi) return;  // zero or one element left
    int t = A[lo]; A[lo] = A[hi]; A[hi] = t;
    reverseArr(A, lo + 1, hi - 1);
}

bool isPalindrome(const char *s, int lo, int hi) {
    if (lo >= hi) return true;
    if (s[lo] != s[hi]) return false;
    return isPalindrome(s, lo + 1, hi - 1);
}
```

Kode 10.11 (Lanjutan) *08_arrays_lists.cpp: rekursi atas larik (mengupas satu elemen atau membelah dua) dan atas singly linked list (simpul pertama dan sisanya).*

```
void printArr(const char *label, const int A[], int n) {
    std::printf("%s", label);
    for (int i = 0; i < n; ++i) std::printf(" %d", A[i]);
    std::printf("\n");
}

// ---------- singly linked lists ----------
struct Node {
    int data;
    Node *next;
};

Node *push(Node *head, int x) { return new Node{x, head}; }

int length(const Node *p) { return p == nullptr ? 0 : 1 + length(p->next); }
int sumList(const Node *p) { return p == nullptr ? 0 : p->data + sumList(p->next); }

bool contains(const Node *p, int key) {
    if (p == nullptr) return false;
    return p->data == key || contains(p->next, key);
}

void printForward(const Node *p) {  // print, then recurse
    if (p == nullptr) return;
    std::printf(" %d", p->data);
    printForward(p->next);
}

void printBackward(const Node *p) {  // recurse, then print
    if (p == nullptr) return;
    printBackward(p->next);
    std::printf(" %d", p->data);
}

Node *copyList(const Node *p) {  // a new node for every node
    if (p == nullptr) return nullptr;
    return new Node{p->data, copyList(p->next)};
}
```

Kode 10.11 (Lanjutan) *08_arrays_lists.cpp: rekursi atas larik (mengupas satu elemen atau membelah dua) dan atas singly linked list (simpul pertama dan sisanya).*

```
Node *removeAll(Node *p, int key) {  // returns the new first node
    if (p == nullptr) return nullptr;
    Node *rest = removeAll(p->next, key);
    if (p->data == key) {
        delete p;
        return rest;
    }
    p->next = rest;
    return p;
}

int destroy(Node *p) {  // returns the number freed
    if (p == nullptr) return 0;
    int freed = destroy(p->next);  // free the rest first
    delete p;
    return freed + 1;
}

int main() {
    int loans[] = {12, 7, 31, 4, 25, 18};  // loans per day at Pustaka Ceria
    int n = 6;
    printArr("loans:", loans, n);
    std::printf("sum = %d, max = %d\n", sumArr(loans, n), maxArr(loans, 0, n - 1));
    std::printf("search 4 -> index %d, search 9 -> index %d\n",
                searchArr(loans, n, 0, 4), searchArr(loans, n, 0, 9));
    reverseArr(loans, 0, n - 1);
    printArr("reversed:", loans, n);

    int sorted[] = {4, 7, 12, 18, 25, 31};
    std::printf("binary search 25 -> index %d, 5 -> index %d\n",
                binSearch(sorted, 0, 5, 25), binSearch(sorted, 0, 5, 5));
    const char *words[] = {"racecar", "rotor", "library", "a", ""};
    for (const char *w : words)
        std::printf("\"%s\" is %sa palindrome\n", w,
                    isPalindrome(w, 0, (int)std::strlen(w) - 1) ? "" : "not ");
```

Kode 10.11 (Lanjutan) *08_arrays_lists.cpp: rekursi atas larik (mengupas satu elemen atau membelah dua) dan atas singly linked list (simpul pertama dan sisanya).*

```
    Node *head = nullptr;
    int values[] = {50, 40, 30, 40, 20};
    for (int x : values) head = push(head, x);  // pushed at the front
    std::printf("\nlist forward :");
    printForward(head);
    std::printf("\nlist backward:");
    printBackward(head);
    std::printf("\nlength = %d, sum = %d, contains 30: %s, contains 35: %s\n",
                length(head), sumList(head), contains(head, 30) ? "yes" : "no",
                contains(head, 35) ? "yes" : "no");
    Node *copy = copyList(head);
    head = removeAll(head, 40);
    std::printf("after removing 40:");
    printForward(head);
    std::printf("\nthe copy is untouched:");
    printForward(copy);
    std::printf("\nfreed %d + %d nodes\n", destroy(head), destroy(copy));
    return 0;
}
```

```
loans: 12 7 31 4 25 18
sum = 97, max = 31
search 4 -> index 3, search 9 -> index -1
reversed: 18 25 4 31 7 12
binary search 25 -> index 4, 5 -> index -1
"racecar" is a palindrome
"rotor" is a palindrome
"library" is not a palindrome
"a" is a palindrome
"" is a palindrome

list forward : 20 40 30 40 50
list backward: 50 40 30 40 20
length = 5, sum = 180, contains 30: yes, contains 35: no
after removing 40: 20 30 50
the copy is untouched: 20 40 30 40 50
freed 3 + 5 nodes
```

Uji palindrom itu juga benar untuk string kosong: dengan hi $= -1$ syarat `lo >= hi` langsung terpenuhi. Bagian *linked list* membangun 20, 40, 30, 40, 50 dengan melakukan *push* 50, 40, 30, 40, 20 di depan, mencetaknya maju dan mundur, lalu membuang kedua 40 sementara salinannya tetap menyimpannya. Karena program itu mengalokasikan dan membebaskan simpul, ia dijalankan di bawah Valgrind dan di bawah AddressSanitizer bersama UndefinedBehaviorSanitizer (keduanya diperkenalkan di Subbab 7.11). Ringkasan Valgrind:

```
==PID== HEAP SUMMARY:
==PID==     in use at exit: 0 bytes in 0 blocks
==PID==   total heap usage: 12 allocs, 12 frees, 77,984 bytes allocated
==PID==
==PID== All heap blocks were freed -- no leaks are possible
==PID==
==PID== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

(Kedua belas alokasi itu adalah 10 simpul dan dua blok yang dialokasikan pustaka *runtime* C dan C++ untuk keperluannya sendiri.) AddressSanitizer juga tidak melaporkan apa pun. Program Java `ArraysLists.java` dan program Python `08_arrays_lists.py` mencetak keluaran yang sama; di sana `destroy` hanya memutus tautan simpul-simpul dan menghitungnya, dan *garbage collector* yang membebaskannya.

10.9.3 Ongkos, dan sebuah peringatan

Setiap fungsi list di atas melakukan satu panggilan per simpul ditambah satu untuk pointer nol: waktu $\Theta(n)$ dan *stack* $\Theta(n)$. Itulah peringatan Subbab 7.6.2 sekali lagi. Rekursi atas list berisi sejuta simpul memerlukan sejuta *frame*, terlalu banyak untuk *stack* bawaan ketiga bahasa (Subbab 10.12 memberikan angka-angkanya), sedangkan versi gelung di Bab 7 tidak memerlukannya sama sekali. Rekursi atas list sangat baik untuk berpikir, untuk membuktikan, dan untuk list pendek; untuk list panjang, tulislah gelungnya, atau pakailah *stack* eksplisit seperti yang dilakukan Subbab 10.13 untuk `printBackward`. Rekursi membelah dua pada larik tidak mengalami masalah itu: kedalaman $\lceil \lg n \rceil + 1$ hanyalah 31 untuk semiliar elemen.

10.10 Pohon Rekursi

10.10.1 Menggambar panggilan-panggilannya

Kita sudah menggambar beberapa lukisan tentang panggilan yang dilakukan sebuah rekursi: satu rantai untuk faktorial (**Gambar 10.2**), pohon biner yang lebat untuk Fibonacci naif (**Gambar 10.4**), rantai pangkat yang terbagi dua (**Gambar 10.7**), dan pohon biner lengkap untuk

Hanoi (**Gambar 10.9**). Lukisan semacam itu, satu simpul per panggilan dengan panggilan-panggilan rekursifnya sebagai anak, disebut *pohon rekursi* (*recursion tree*). Bila di setiap simpul kita tuliskan kerja yang dilakukan panggilan itu *sendiri* (tanpa menghitung anak-anaknya), total waktu jalannya adalah jumlah semua bilangan di pohon itu, dan kedalaman rekursinya, yang menentukan memori *stack*, adalah tinggi pohonnya.

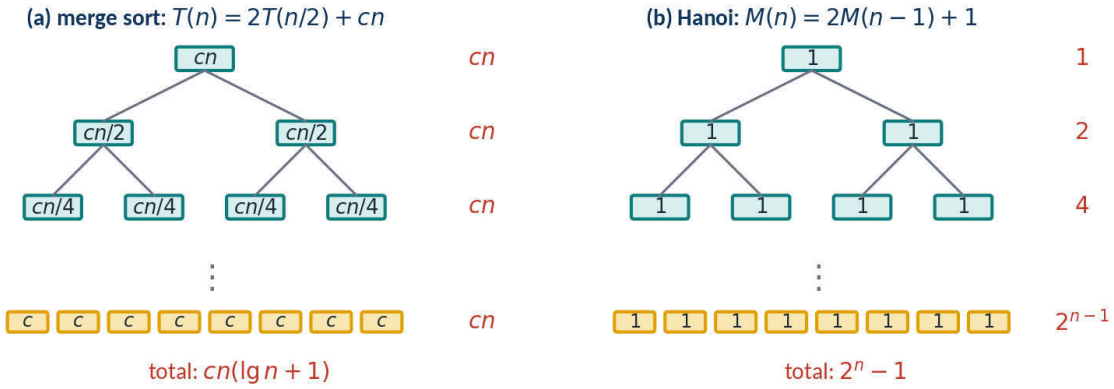
Jumlah itu paling mudah dihitung aras demi aras. Dengan cara itulah Subbab 3.8.4 menemukan ongkos *merge sort*, dan itulah metode umum yang dijanjikan di sana.

10.10.2 Merge sort dan Hanoi, aras demi aras

Gambar 10.10(a) adalah pohon rekursi *merge sort* pada n elemen, dengan panggilan pada k elemen melakukan kerja ck untuk menggabungkan (untuk suatu konstanta c) dan melakukan dua panggilan pada $k/2$ elemen; anggaplah n perpangkatan dua. Akarnya mengerjakan cn. Aras 1 memiliki dua simpul $cn/2$: totalnya lagi-lagi cn. Aras 2 memiliki empat simpul $cn/4$: cn. Secara umum aras i memiliki 2^i simpul $cn/2^i$, totalnya cn, dan daun-daunnya berada pada aras $\lg n$, tempat sublariknya tinggal satu elemen. Ada $\lg n + 1$ aras, masing-masing berongkos cn, sehingga

$$T(n) = 2T(n/2) + cn \quad \Rightarrow \quad T(n) = cn(\lg n + 1) = \Theta(n \lg n). \tag{10.17}$$

Gambar 10.10(b) adalah pohon rekursi Hanoi untuk n cakram, dengan setiap panggilan membuat satu langkah dan dua panggilan pada $n-1$ cakram. Aras i memiliki 2^i simpul yang masing-masing membuat satu langkah, dan arasnya berkisar dari 0 sampai $n-1$: totalnya $1 + 2 + 4 + \cdots + 2^{n-1} = 2^n - 1$, yaitu Persamaan (10.16) yang diselesaikan sekali lagi, kali ini tanpa menebak jawabannya.



Gambar 10.10 Pohon rekursi dengan ongkos setiap aras di sebelah kanan. (a) *Merge sort*: $\lg n + 1$ aras yang masing-masing berongkos cn. (b) Hanoi: aras i berongkos 2^i, dan jumlah semua aras $2^n - 1$.

Mengapa kedua pohon ini, yang sama-sama biner, memberikan total yang begitu berbeda? Pada *merge sort* masalahnya *terbelah dua* di setiap aras, sehingga pohonnya hanya memiliki $\lg n + 1$ aras, dan kerja per aras tetap cn. Pada Hanoi masalahnya menyusut *satu* di setiap aras, sehingga pohonnya memiliki n aras, dan banyaknya simpul berlipat dua di setiap aras. Bentuk rekurensnya menentukan segalanya: berapa banyak submasalah, seberapa lebih kecil, dan seberapa banyak kerja di luar panggilan.

10.10.3 Tabel rekurens

Tabel 10.2 mengumpulkan rekurens-rekurens bab ini beserta penyelesaiannya, yang masing-masing telah dibuktikan di atas dengan penguraian, dengan pohon rekursi, atau dengan induksi. Tabel itu mencakup sebagian besar algoritma rekursif yang dijumpai pada tahap awal belajar

algoritma. (Rekurens berbentuk umum $T(n) = aT(n/b) + f(n)$ diselesaikan dengan *master theorem* [3], yang dibahas tuntas dalam buku-buku analisis algoritma.)

Tabel 10.2 Rekurens yang lazim, penyelesaiannya, dan algoritma dalam buku ini yang menghasilkannya.

Rekurens	Penyelesaian	Tempat munculnya
$T(n) = T(n-1) + c$	$\Theta(n)$	faktorial, panjang list, *linear search*, m^n naif
$T(n) = T(n/2) + c$	$\Theta(\log n)$	*binary search*, pemangkatan cepat, *fast doubling*
$T(n) = 2T(n/2) + c$	$\Theta(n)$	maksimum dengan membelah dua (`maxArr`)
$T(n) = 2T(n/2) + cn$	$\Theta(n \log n)$	*merge sort* (Persamaan (10.17))
$T(n) = T(n-1) + cn$	$\Theta(n^2)$	*selection sort* yang ditulis rekursif, kasus terburuk *quicksort*
$T(n) = 2T(n-1) + c$	$\Theta(2^n)$	Menara Hanoi (Persamaan (10.16))
$T(n) = T(n-1) + T(n-2) + c$	$\Theta(\varphi^n)$	Fibonacci naif (Persamaan (10.6))

Baris ketiga patut diberi catatan: `maxArr` membelah dua lariknya tetapi tetap memakan waktu $\Theta(n)$, karena ia harus melihat setiap elemen; pohon rekursinya memiliki n daun dan $n-1$ simpul dalam, masing-masing dengan kerja konstan. Membelah dua hanya menghemat waktu bila salah satu belahan dapat *dibuang*, seperti pada *binary search*, atau bila kerja penggabungannya dibagi dengan cerdik, seperti pada pemangkatan cepat.

10.11 Rekursi Ekor dan Apa yang Sesungguhnya Dilakukan Compiler

10.11.1 Tail call

Sebuah panggilan adalah *tail call* bila ia merupakan tindakan terakhir fungsi pemanggilnya: pemanggil mengembalikan hasil fungsi yang dipanggil apa adanya, tanpa ada yang tersisa untuk dikerjakan sesudahnya. Pada

```
long long sumPlain(long long n) {              // 1 + 2 + ... + n
    if (n == 0) return 0;
    return n + sumPlain(n - 1);                // NOT a tail call: the + waits
}

long long sumAcc(long long n, long long acc) {  // acc + 1 + 2 + ... + n
    if (n == 0) return acc;
    return sumAcc(n - 1, acc + n);             // a tail call
}
```

`sumPlain` masih harus menambahkan `n` setelah panggilan rekursifnya kembali, sehingga *frame*-nya, yang memuat `n`, harus bertahan melewati panggilan itu. `sumAcc` telah menyelesaikan seluruh kerjanya *sebelum* panggilan: jumlah sementara turun bersama parameter `acc`, sebuah *akumulator*, dan ketika panggilan itu kembali tidak ada lagi yang perlu dilakukan selain kembali. Kebenarannya mengikuti dari sebuah invarian yang berperan sebagai hipotesis induksi: untuk setiap panggilan, `acc` $+(1+2+\cdots+n)$ sama dengan hasil yang kita inginkan; hal itu benar untuk panggilan pertama `sumAcc(n, 0)` dan dipertahankan oleh setiap panggilan karena $(\text{acc}+n)+(1+\cdots+(n-1)) = \text{acc}+(1+\cdots+n)$.

Karena *frame* `sumAcc(n, acc)` tidak pernah diperlukan lagi begitu ia memanggil `sumAcc(n - 1, acc + n)`, panggilan baru itu dapat saja *memakai ulang frame* tersebut: menimpa `n` dan `acc` dengan nilai-nilai baru dan melompat kembali ke awal fungsi. Transformasi itu disebut *tail-call*

elimination (penghapusan panggilan ekor, atau *tail-call optimization*), dan ia mengubah fungsi rekursif ekor menjadi persis gelung

```
while (n != 0) { acc = acc + n; n = n - 1; }
return acc;
```

yang memakai satu *frame* seberapa pun besarnya *n*. Guy Steele pada 1977 berargumen bahwa panggilan prosedur pada posisi ekor semestinya dikompilasi sebagai lompatan, "LAMBDA: the ultimate GOTO" [13], dan bahasa pemrograman Scheme mewajibkannya sejak itu. C, C++, Java, dan Python tidak mewajibkannya. Apa yang sesungguhnya dilakukan implementasi-implementasinya adalah persoalan fakta, dan sisa subbab ini menjawabnya dengan eksperimen.

10.11.2 Apa yang dilakukan GCC: sebuah eksperimen

Kode 10.12 memuat `sumPlain` dan `sumAcc` serta fungsi ketiga, `sumKeep`, yang ditulis sedemikian rupa sehingga tidak ada *compiler* yang dapat menghapus rekursinya: ia menyimpan `n` dalam variabel lokal `volatile` dan membacanya kembali *setelah* panggilan rekursifnya kembali, sehingga *frame*-nya memang harus bertahan melewati panggilan itu. Program itu menerima nama fungsi dan *n* dari baris perintah.

Kode 10.12 `09_tail.cpp`: rekursi bukan-ekor, rekursi ekor, dan rekursi yang *frame*-nya harus bertahan, dapat dipilih dari baris perintah.

```
// 09_tail.cpp -- three ways to add 1 + 2 + ... + n recursively. Run it as
//   ./09_tail plain|acc|keep n
// and compare builds made with -O0 and with -O2 (Section 10.11).
#include <cstdio>
#include <cstdlib>
#include <cstring>

long long sumPlain(long long n) {  // NOT a tail call: the addition
    if (n == 0) return 0;          // happens after the call returns
    return n + sumPlain(n - 1);
}

long long sumAcc(long long n, long long acc) {  // a tail call: the answer so far
    if (n == 0) return acc;                     // travels down in acc
    return sumAcc(n - 1, acc + n);
}

long long sumKeep(long long n) {  // the frame must survive the call:
    volatile long long mine = n;  // `mine` is read after it returns
    if (n == 0) return 0;
    long long rest = sumKeep(n - 1);
    return mine + rest;
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        std::printf("usage: %s plain|acc|keep n\n", argv[0]);
        return 1;
    }
    long long n = std::atoll(argv[2]);
    long long s;
    if (std::strcmp(argv[1], "plain") == 0) s = sumPlain(n);
    else if (std::strcmp(argv[1], "acc") == 0) s = sumAcc(n, 0);
    else s = sumKeep(n);
    std::printf("%s(%lld) = %lld\n", argv[1], n, s);
    return 0;
}
```

Kami membangunnya dua kali, tanpa optimisasi (`-O0`) dan dengan optimisasi yang lazim (`-O2`), dan menjalankan setiap fungsi dengan $n = 100{,}000$ dan $n = 10{,}000{,}000$ pada mesin Linux buku ini, yang *stack* bawaannya 8 MB:

```
$ g++ -std=c++17 -O0 -Wall -Wextra -o tail_O0 09_tail.cpp
$ g++ -std=c++17 -O2 -Wall -Wextra -o tail_O2 09_tail.cpp
$ ./tail_O0 plain 100000
plain(100000) = 5000050000
$ ./tail_O0 plain 10000000
Segmentation fault
$ ./tail_O0 acc 100000
acc(100000) = 5000050000
$ ./tail_O0 acc 10000000
Segmentation fault
$ ./tail_O0 keep 100000
keep(100000) = 5000050000
$ ./tail_O0 keep 10000000
Segmentation fault
$ ./tail_O2 plain 100000
plain(100000) = 5000050000
$ ./tail_O2 plain 10000000
plain(10000000) = 50000005000000
$ ./tail_O2 acc 100000
acc(100000) = 5000050000
$ ./tail_O2 acc 10000000
acc(10000000) = 50000005000000
$ ./tail_O2 keep 100000
keep(100000) = 5000050000
$ ./tail_O2 keep 10000000
Segmentation fault
```

Dengan `-O0` setiap fungsi *crash* pada sepuluh juta: *compiler* menerjemahkan setiap panggilan secara harfiah, satu *frame* per panggilan, dan sepuluh juta *frame* tidak muat dalam 8 MB. Dengan `-O2`, `sumAcc` berjalan sampai sepuluh juta tanpa masalah, sesuai ramalan *tail-call elimination*. Yang mengejutkan, `sumPlain`, yang *tidak* rekursif ekor, juga berhasil. Dan `sumKeep` tetap *crash*. Bahasa *assembly* yang dihasilkan GCC menjelaskan ketiga hasil itu. (*Assembly* adalah set instruksi prosesor itu sendiri yang ditulis dalam kata-kata; Anda tidak perlu menguasainya untuk mengikuti argumennya, cukup memperhatikan apakah kata `call` muncul.) Berikut `sumAcc` dengan `-O0`, dihasilkan dengan `g++ -S -masm=intel` dan ditampilkan tanpa direktif pembukuannya:

```
sumAcc(long long, long long):
    push rbp
    mov rbp, rsp
    sub rsp, 16
    mov QWORD PTR -8[rbp], rdi
    mov QWORD PTR -16[rbp], rsi
    cmp QWORD PTR -8[rbp], 0
    jne .L5
    mov rax, QWORD PTR -16[rbp]
    jmp .L6
.L5:
    mov rdx, QWORD PTR -16[rbp]
    mov rax, QWORD PTR -8[rbp]
    add rdx, rax
    mov rax, QWORD PTR -8[rbp]
    sub rax, 1
    mov rsi, rdx
    mov rdi, rax
    call sumAcc(long long, long long)
    nop
.L6:
    leave
    ret
```

Baris `call sumAcc(long long, long long)` adalah panggilan rekursif sungguhan: ia melakukan *push* alamat kembali, lalu fungsi yang dipanggil melakukan *push frame*-nya sendiri dengan `push rbp` dan `sub rsp, 16`. Sekarang fungsi yang sama dengan `-O2`:

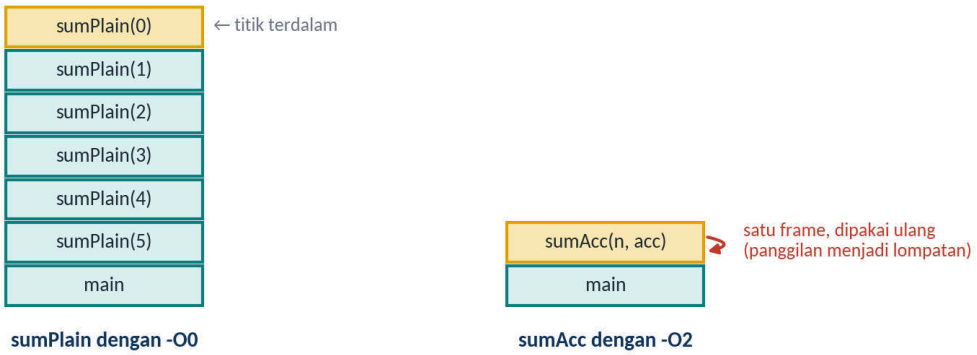
```
sumAcc(long long, long long):
    mov rax, rsi
    test rdi, rdi
    je .L14
    lea rdx, -1[rdi]
    test dil, 1
    je .L11
    add rax, rdi
    mov rdi, rdx
    test rdx, rdx
    je .L25
.L11:
    lea rax, -1[rax+rdi*2]
    sub rdi, 2
    jne .L11
.L14:
    ret
.L25:
    ret
```

Tidak ada `call` sama sekali. Rekursinya telah menjadi gelung pada label `.L11`, yang bahkan menangani dua nilai `n` per iterasi (ia menambahkan $n + (n-1) = 2n - 1$ dengan satu instruksi `lea` dan mengurangkan 2), dengan sedikit pembuka untuk n ganjil. Dan `sumPlain` dengan `-O2`:

```
sumPlain(long long):
    xor eax, eax
    test rdi, rdi
    je .L1
.L2:
    add rax, rdi
    sub rdi, 1
    jne .L2
.L1:
    ret
```

Inilah gelung `acc += n; n -= 1;` dalam tiga instruksi. GCC mengenali bahwa operasi yang tertunda pada `sumPlain` adalah penjumlahan, yang bersifat asosiatif, memperkenalkan akumulator atas prakarsanya sendiri, lalu menghapus *tail call* yang dihasilkannya. Pengoptimasi GCC melakukan hal ini untuk penjumlahan dan perkalian yang menunggu satu panggilan rekursif, dan itu pula sebabnya `factRec` dan `powerNaive` yang rekursif tidak pernah mendekati *stack overflow* dengan `-O2`, dan mengapa Fibonacci naif pada **Tabel 10.1** lebih cepat di C++ daripada yang mungkin diduga (salah satu dari kedua panggilannya menjadi gelung; yang lain tetap panggilan sungguhan). Opsi yang bertanggung jawab adalah `-foptimize-sibling-calls`, "Optimize sibling and tail recursive calls", yang diaktifkan GCC pada `-O2`, `-O3`, dan `-Os`, tetapi tidak pada `-O0` atau `-O1` [14]. `sumKeep` tidak dapat ditransformasi: *frame*-nya memuat nilai `volatile` yang harus dibaca setelah panggilan. GCC menyisipkan (*inline*) tiga tingkat rekursi ke dalam setiap panggilan untuk menghemat waktu, tetapi sebuah `call` sungguhan tetap ada, demikian pula pertumbuhan *frame*-nya.

Gambar 10.11 mempertentangkan kedua keadaan itu: dengan `-O0` *stack* memuat satu *frame* untuk setiap panggilan yang tertunda; setelah *tail-call elimination*, satu *frame* dipakai ulang untuk seluruh perhitungan.



Gambar 10.11 Kiri: tanpa optimisasi, `sumPlain(5)` memiliki enam *frame* miliknya sendiri di *stack* pada titik terdalam. Kanan: dengan `-O2`, *tail call* `sumAcc` dikompilasi sebagai lompatan, dan satu *frame* melayani seluruh rekursi (GCC melakukan hal yang sama untuk `sumPlain` setelah memperkenalkan akumulator).

10.11.3 Mengapa Anda tidak boleh bergantung padanya di C++

Eksperimen itu memperlihatkan apa yang dilakukan GCC 13.3 pada satu mesin. Itu *bukan* janji. Standar C++ tidak mengatakan apa pun tentang *tail call*, sehingga transformasi itu adalah optimisasi yang boleh atau tidak diterapkan *compiler*. Ia lenyap pada `-O0`, yang merupakan cara kebanyakan orang membangun program saat *debugging*; ia dapat lenyap bila sebuah fungsi memiliki objek lokal dengan *destructor* (*destructor* itu harus berjalan *setelah* panggilan, sehingga panggilan itu bukan lagi tindakan terakhir), bila argumen dilewatkan lewat *stack* pada konvensi pemanggilan tertentu, bila alamat sebuah variabel lokal lolos keluar, atau sekadar pada *compiler* atau versi yang berbeda. Karena itu program yang kebenarannya bergantung pada *tail-call elimination* dapat berjalan pada *build* rilis dan *crash* pada *build debug*, jenis *bug* yang paling tidak menyenangkan.

Untuk kode yang harus bergantung padanya, GCC (sejak versi 15) dan Clang sama-sama menyediakan atribut eksplisit, `[[gnu::musttail]]` atau `[[clang::musttail]]`, yang diletakkan pada pernyataan `return`: “It asserts that the call must be a tail call that does not allocate extra stack space, so it is safe to use tail recursion to implement long-running loops. If the compiler cannot generate a `musttail` tail call it reports an error” [15]. Seri GCC terkini per September 2026 adalah GCC 16, yang rilis pertamanya, 16.1, terbit pada April 2026 [16]. Atribut itu bukan bagian dari C++ standar, sehingga buku ini tidak memakainya; bila sebuah rekursi mungkin dalam, kita menulis gelungnya.

10.11.4 JVM dan CPython tidak menghapus tail call

Mesin virtual Java HotSpot tidak menghapus *tail call*, begitu pula *compiler* JIT-nya. **Kode 10.13** menjalankan `sumPlain` dan `sumAcc` yang rekursif ekor dengan n yang makin besar dan menangkap galatnya.

Kode 10.13 `TailCalls.java`: di Java, metode rekursif ekor memakai satu *frame* per panggilan, persis seperti metode lainnya.

```java
// TailCalls.java -- the JVM does not eliminate tail calls: a tail-recursive method
// still uses one stack frame per call, exactly like a non-tail-recursive one.
public class TailCalls {
    static long sumPlain(long n) {  // not a tail call
        if (n == 0) return 0;
        return n + sumPlain(n - 1);
    }

    static long sumAcc(long n, long acc) {  // a tail call
        if (n == 0) return acc;
        return sumAcc(n - 1, acc + n);
    }

    static long sumLoop(long n) {  // what a tail call "means"
        long acc = 0;
        while (n != 0) {    // sumAcc(n, acc) becomes
            acc = acc + n;  //   acc = acc + n; n = n - 1;
            n = n - 1;      // and start again from the top
        }
        return acc;
    }

    static String attempt(boolean tail, long n) {
        try {
            return String.valueOf(tail ? sumAcc(n, 0) : sumPlain(n));
        } catch (StackOverflowError e) {
            return "StackOverflowError";
        }
    }

    public static void main(String[] args) {
        System.out.printf("%10s %22s %22s %16s%n", "n", "sumPlain", "sumAcc",
                          "sumLoop");
        for (long n : new long[] {1000, 10000, 1000000, 10000000})
            System.out.printf("%10d %22s %22s %16d%n", n, attempt(false, n),
                              attempt(true, n), sumLoop(n));
    }
}
```

```
         n              sumPlain               sumAcc         sumLoop
      1000                 500500               500500          500500
     10000               50005000             50005000        50005000
   1000000     StackOverflowError     StackOverflowError     500000500000
  10000000     StackOverflowError     StackOverflowError   50000005000000
```

Kedua metode gagal pada n yang sama; menjadi rekursif ekor tidak membawa perbedaan apa pun. Metode `sumLoop` memperlihatkan "arti" sebuah *tail call*: parameter-parameternya menjadi variabel yang ditimpa, dan panggilannya menjadi lompatan ke awal gelung. CPython berperilaku sama, dengan sengaja. Guido van Rossum, pencipta Python, menjelaskan pada 2009 bahwa ia tidak akan menambahkan *tail-call elimination*, terutama karena hal itu merusak *stack trace* yang diandalkan para pemrogram Python untuk *debugging*, dan karena hal itu akan mendorong gaya pemrograman yang asing bagi bahasa tersebut [17]. **Kode 10.14** melakukan eksperimen yang sama.

Kode 10.14 `09_tail.py`: di Python, fungsi rekursif ekor menabrak batas rekursi seperti fungsi lainnya.

```python
# 09_tail.py -- CPython does not eliminate tail calls: a tail-recursive function
# still uses one frame per call and hits the recursion limit like any other.
import sys


def sum_plain(n):  # not a tail call
    if n == 0:
        return 0
    return n + sum_plain(n - 1)


def sum_acc(n, acc):  # a tail call
    if n == 0:
        return acc
    return sum_acc(n - 1, acc + n)


def sum_loop(n):  # what a tail call "means"
    acc = 0
    while n != 0:                # sum_acc(n, acc) becomes
        acc, n = acc + n, n - 1  #   acc = acc + n; n = n - 1
    return acc                   # and start again from the top


def attempt(fn, n):
    try:
        return str(fn(n))
    except RecursionError:
        return "RecursionError"


def main():
    print(f"recursion limit: {sys.getrecursionlimit()}")
    print(f"{'n':>10s} {'sum_plain':>16s} {'sum_acc':>16s} {'sum_loop':>16s}")
    for n in [900, 1000, 10000, 1000000]:
        print(f"{n:10d} {attempt(sum_plain, n):>16s} "
              f"{attempt(lambda k: sum_acc(k, 0), n):>16s} {sum_loop(n):16d}")


if __name__ == "__main__":
    main()
```

```
recursion limit: 1000
         n        sum_plain          sum_acc         sum_loop
       900           405450           405450           405450
      1000   RecursionError   RecursionError           500500
     10000   RecursionError   RecursionError         50005000
   1000000   RecursionError   RecursionError     500000500000
```

Aturannya sederhana. Di Java dan Python, fungsi rekursif ekor sama dalamnya dengan masukannya; bila Anda menginginkan gelung, tulislah gelung. Di C++ dengan optimisasi, GCC dan Clang *biasanya* mengubah rekursi ekor (dan akumulasi sederhana) menjadi gelung; nikmatilah, tetapi jangan bergantung padanya.

10.12 Seberapa Dalam Rekursi Boleh Berjalan? Batas di C++, Java, dan Python

10.12.1 C++: ukuran stack

Di C++, batas kedalaman tidak lain adalah ukuran *stack thread* dibagi ukuran sebuah *frame*. Bila *stack* berukuran S bita dan setiap panggilan menempati *frame* berukuran s bita, sebuah rekursi dapat mencapai kira-kira

$$d_{\max} \approx \frac{S}{s} \tag{10.18}$$

tingkat sebelum meluap. Di Linux, ukuran *stack thread* utama ditetapkan oleh batas *shell*, yang dicetak oleh `ulimit -s` dalam kilobita; nilai bawaannya 8.192 KB (8 MB). Di Windows ukurannya ditetapkan oleh *linker* ketika program dibangun, secara bawaan 1 MB untuk *linker* Microsoft [18] maupun MinGW. Di macOS, *stack thread* utama berukuran 8 MB. *Thread* yang dibuat sebuah program biasanya mendapat lebih sedikit, sering kali 512 KB sampai 8 MB bergantung pada sistemnya.

Kode 10.15 mengukur kedua bilangan dalam Persamaan (10.18) pada mesin sungguhan. Fungsi `probe` melakukan rekursi sedalam n tingkat dengan *frame* berukuran realistis (ia memuat larik `volatile` 64 bita dan melakukan panggilan bukan-ekor, sehingga pengoptimasi tidak dapat menghapus keduanya). Fungsi itu mencatat alamat larik lokalnya pada kedalaman 0 dan pada kedalaman 1.000; karena *stack* tumbuh ke arah alamat yang lebih rendah, selisihnya dibagi 1.000 adalah ukuran *frame* rata-rata.

Kode 10.15 `10_depth.cpp`: mengukur ukuran sebuah *stack frame* dan melakukan rekursi sampai kedalaman yang dipilih.

```
// 10_depth.cpp -- how deep can a C++ recursion go? Run it as  ./10_depth n
// It measures the size of one stack frame, then recurses n levels deep.
#include <cstdint>
#include <cstdio>
#include <cstdlib>

const int FAR = 1000;
std::uintptr_t top = 0, far = 0;  // a local's address at depth 0 and FAR

long long probe(long long n, int depth) {
    volatile char scratch[64];  // a frame of realistic size
    scratch[0] = (char)n;
    if (depth == 0) top = (std::uintptr_t)&scratch[0];
    if (depth == FAR) far = (std::uintptr_t)&scratch[0];
    if (n == 0) return 0;
    return 1 + probe(n - 1, depth + 1) + scratch[0] * 0;  // not a tail call
}

int main(int argc, char *argv[]) {
    long long n = argc > 1 ? std::atoll(argv[1]) : FAR;
    long long reached = probe(n, 0);
    // the stack grows toward lower addresses on the usual machines
    double frame = n >= FAR ? (double)(top - far) / FAR : 0.0;
    std::printf("one frame = %.0f bytes; reached depth %lld, using about %.1f MB\n",
                frame, reached, frame * reached / (1024.0 * 1024.0));
    return 0;
}
```

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o depth 10_depth.cpp
$ ulimit -s
8192
$ ./depth 1000
one frame = 72 bytes; reached depth 1000, using about 0.1 MB
$ ./depth 100000
one frame = 72 bytes; reached depth 100000, using about 6.9 MB
$ ./depth 116000
one frame = 72 bytes; reached depth 116000, using about 8.0 MB
$ ./depth 117000
Segmentation fault
$ ./depth 1000000
Segmentation fault
$ ulimit -s 262144
$ ./depth 1000000
one frame = 72 bytes; reached depth 1000000, using about 68.7 MB
$ g++ -std=c++17 -O0 -Wall -Wextra -o depth0 10_depth.cpp
$ ./depth0 100000
Segmentation fault
$ ./depth0 1000
one frame = 112 bytes; reached depth 1000, using about 0.1 MB
```

Dengan `-O2` setiap *frame* berukuran 72 bita, dan Persamaan (10.18) meramalkan 8,388,608/72 ≈ 116,508 tingkat. Hasil jalannya cocok sampai ribuan: 116.000 tingkat berhasil, 117.000 *crash*. Menaikkan batasnya dengan `ulimit -s 262144` (256 MB) meloloskan sejuta tingkat, dengan pemakaian sekitar 69 MB. Bila dibangun dengan `-O0`, fungsi yang sama memakai 112 bita per *frame*, dan 100.000 tingkat, yang baik-baik saja dengan `-O2`, sudah *crash*: batas kedalaman sebuah program C++ bergantung pada tingkat optimisasinya.

Stack juga dapat diperbesar dari dalam proses pembangunan program, pilihan yang *portable* untuk program yang Anda distribusikan:

- **Windows (MSVC):** tautkan dengan `/STACK:268435456` untuk *stack* 256 MB [18]; **Windows (MinGW g++):** `g++ ... -Wl,--stack,268435456`.
- **Linux dan macOS:** jalankan rekursi yang dalam itu di sebuah *thread* yang ukuran *stack*-nya ditetapkan secara eksplisit dengan `pthread_attr_setstacksize` (`std::thread` standar tidak menyediakan cara untuk memilihnya); atau mintalah pengguna menaikkan `ulimit -s`.

Semua ini adalah obat untuk rekursi yang *pada hakikatnya* dalam, seperti penelusuran rekursif atas struktur yang sangat dalam. Untuk rekursi yang dalam hanya karena mengupas satu elemen demi satu elemen dari sebuah list panjang, obat yang lebih baik adalah gelung.

10.12.2 Java: `-Xss`, dan thread dengan stack sendiri

Mesin virtual Java memberi setiap *thread* sebuah *stack* berukuran tetap, yang ditetapkan oleh opsi `-Xss`. Untuk JDK 25 nilai bawaannya 1.024 KB di Linux dan macOS pada x64, 2.048 KB pada AArch64 (misalnya Apple silicon), dan bergantung pada sistem di Windows [19]. *Thread* yang dibuat di dalam program dapat meminta ukuran lain: `Thread.ofPlatform().stackSize(bytes)`, yang tersedia sejak Java 21, atau konstruktor lama `Thread(group, task, name, stackSize)`. Dokumentasinya mengingatkan bahwa mesin virtual bebas memperlakukan nilai itu sebagai saran [20]. **Kode 10.16** mengukur kedalaman tempat *thread* utama meluap, tiga kali, lalu menjalankan `sumPlain(1000000)` dalam *thread* dengan *stack* 256 MB.

Kode 10.16 `Depth.java`: kedalaman rekursi yang dapat dicapai di *thread* utama Java, dan rekursi dalam di sebuah *thread* dengan *stack* yang lebih besar.

```java
// Depth.java -- how deep can a Java recursion go? First in the main thread,
// then in a thread created with a much larger stack.
public class Depth {
    static int depth;

    static void dive() {  // recurses until the stack is full
        ++depth;
        dive();
    }

    static int measure() {
        depth = 0;
        try {
            dive();
        } catch (StackOverflowError e) {
            // the stack is unwound when the error propagates; we may continue
        }
        return depth;
    }

    static long sumPlain(long n) { return n == 0 ? 0 : n + sumPlain(n - 1); }

    public static void main(String[] args) throws InterruptedException {
        for (int trial = 1; trial <= 3; ++trial)
            System.out.printf("main thread, trial %d: "
                              + "StackOverflowError at depth %d%n",
                              trial, measure());

        long[] result = new long[1];
        Runnable deep = () -> result[0] = sumPlain(1000000);
        Thread t = Thread.ofPlatform().stackSize(256L * 1024 * 1024)   // 256 MB
                         .unstarted(deep);
        t.start();
        t.join();
        System.out.printf("in a big thread, sumPlain(1000000) = %d%n", result[0]);
    }
}
```

```
main thread, trial 1: StackOverflowError at depth 10730
main thread, trial 2: StackOverflowError at depth 23608
main thread, trial 3: StackOverflowError at depth 23608
in a big thread, sumPlain(1000000) = 500000500000
```

Ketiga percobaan itu tidak sepakat, dan ketidaksepakatan itu mengandung pelajaran. Percobaan pertama berjalan di interpreter, yang *frame*-nya besar, dan mencapai 10.730 tingkat. Pada percobaan kedua, *compiler* JIT telah mengompilasi `dive`, yang *frame* hasil kompilasinya sekitar 44 bita, dan kedalamannya naik lebih dari dua kali lipat. Dalam beberapa kali menjalankan program ini kami mengamati nilai dari 10.730 sampai 59.020 untuk kode yang sama: di Java, kedalaman maksimum bukan bilangan tetap, dan program yang melakukan rekursi sedalam 20.000 tingkat bisa berjalan pada satu kesempatan dan gagal pada kesempatan berikutnya. *Thread* dengan *stack* 256 MB menghitung `sumPlain` sejuta tingkat tanpa keluhan. `StackOverflowError` dapat ditangkap, seperti yang dilakukan `measure`, karena *stack*-nya sudah diurai ketika blok `catch` berjalan; tetapi menangkapnya adalah alat diagnosis, bukan rancangan.

10.12.3 Python: batas rekursi

CPython membatasi rekursi dengan menghitung, bukan dengan mengukur memori. Fungsi `sys.getrecursionlimit()` mengembalikan batas itu, secara bawaan 1.000, dan `sys.setrecursionlimit(limit)` mengubahnya; dokumentasinya menjelaskan bahwa “this limit prevents infinite recursion from causing an overflow of the C stack and crashing Python”, dan mengingatkan bahwa “a too-high limit can lead to a crash” [21]. Perlindungannya berubah pada

versi-versi terakhir. Sejak Python 3.11, panggilan dari kode Python ke fungsi Python tidak lagi memakai *stack* C milik interpreter: “Most Python function calls now consume no C stack space … This also means recursive functions can recurse significantly deeper (if the user increases the recursion limit)” [22]. Sejak Python 3.12 batas rekursi “applies only to Python code”; fungsi bawaan dilindungi oleh mekanisme terpisah [23]. **Kode 10.17** mengukur nilai bawaan, menaikkan batasnya, dan akhirnya menjalankan rekursi sejuta tingkat di sebuah *thread* dengan *stack* yang lebih besar.

Kode 10.17 `10_depth.py`: batas rekursi Python, cara menaikkannya, dan rekursi dalam di sebuah *thread* dengan *stack* yang lebih besar.

```
# 10_depth.py -- how deep can a Python recursion go? The recursion limit, how to
# raise it, and a thread with a bigger stack for the cases that need one.
import sys
import threading

depth = 0


def dive():  # recurses until Python stops it
    global depth
    depth += 1
    dive()


def measure():
    global depth
    depth = 0
    try:
        dive()
    except RecursionError:
        pass  # the frames are unwound; we may continue
    return depth


def sum_plain(n):
    return 0 if n == 0 else n + sum_plain(n - 1)


def main():
    print(f"Python {sys.version_info.major}.{sys.version_info.minor}")
    print(f"default limit {sys.getrecursionlimit()}: "
          f"RecursionError at depth {measure()}")

    sys.setrecursionlimit(200_000)
    print(f"limit {sys.getrecursionlimit()}: RecursionError at depth {measure()}")
    print(f"sum_plain(150000) = {sum_plain(150_000)}")
```

Kode 10.17 (Lanjutan) *10_depth.py: batas rekursi Python, cara menaikkannya, dan rekursi dalam di sebuah thread dengan stack yang lebih besar.*

```
    # Deeper still: raise the limit and run the work in a thread with a large
    # stack, so that the C stack cannot run out first on any Python version.
    sys.setrecursionlimit(1_100_000)
    threading.stack_size(256 * 1024 * 1024)  # 256 MB for new threads
    result = []
    t = threading.Thread(target=lambda: result.append(sum_plain(1_000_000)))
    t.start()
    t.join()
    print(f"in a big thread, sum_plain(1000000) = {result[0]}")


if __name__ == "__main__":
    main()
```

```
Python 3.11
default limit 1000: RecursionError at depth 997
limit 200000: RecursionError at depth 199997
sum_plain(150000) = 11250075000
in a big thread, sum_plain(1000000) = 500000500000
```

Batas bawaan menghentikan rekursi pada kedalaman 997, bukan 1.000, karena tiga *frame* (modul, `main`, dan `measure`) sudah berada di *stack*. Dengan batas yang dinaikkan menjadi 200.000, rekursi sedalam 150.000 tingkat berhasil; pada Python 3.11 dan sesudahnya hal itu berjalan bahkan di *thread* utama, karena panggilan Python murni menyimpan *frame*-nya di memori *heap*. Bagian terakhir menaikkan batas di atas sejuta dan memakai `threading.stack_size` untuk memberi *thread* baru *stack* 256 MB, yang melindungi program pada versi dan platform apa pun seandainya ada kode C dalam rantai panggilan yang memakai *stack* C. Rekursi dalam di Python juga lambat: setiap panggilan memakan puluhan nanodetik, dan rekursi sejuta tingkat memakan sebagian detik yang terasa.

10.12.4 Batas-batas itu sekilas

Tabel 10.3 merangkum ketiga bahasa.

Tabel 10.3 Batas kedalaman rekursi di C++, Java, dan Python, sebagaimana diukur pada mesin buku ini pada September 2026.

	C++	Java	Python
Batas bawaan	ukuran *stack*: 8 MB pada *thread* utama Linux dan macOS, 1 MB di Windows	*stack thread*: 1.024 KB (Linux/macOS x64), 2.048 KB (AArch64)	1.000 panggilan bersarang
Terukur di sini	116.000 tingkat *frame* 72 bita (`-O2`)	10.730 sampai 59.020 tingkat, berubah-ubah mengikuti JIT	997 tingkat
Yang terjadi	*segmentation fault* (Windows: eksepsi *stack overflow*)	`StackOverflowError`	`RecursionError`
Cara menaikkan	`ulimit -s`; *linker* `/STACK` atau `-Wl,--stack`; *thread* dengan `pthread_attr_setstacksize`	`java -Xss512m`; `Thread.ofPlatform().stackSize(...)`	`sys.setrecursionlimit`; `threading.stack_size` untuk *thread* baru
Tail call dihapus ?	sering pada `-O2`, tidak pernah dijamin; `[[gnu::musttail]]`/`[[clang::musttail]]` untuk memaksanya	tidak	tidak

10.13 Dari Rekursi ke Iterasi dengan Stack Eksplisit

10.13.1 Kapan harus diubah

Sebuah rekursi sebaiknya diubah menjadi gelung bila kedalamannya dapat besar, yakni sebanding dengan ukuran masukan dan bukan dengan logaritmanya, dan masukannya mungkin besar: rekursi atas *linked list* berisi sejuta simpul, penelusuran atas struktur yang bersarang sampai kedalaman yang tidak diketahui dan yang datang dari luar program (sebuah berkas, halaman web, pengguna), atau rekursi naif yang juga akan eksponensial. Rekursi boleh tetap rekursi bila kedalamannya logaritmik (*binary search*, pemangkatan cepat, Euklides, *merge sort*, dan *quicksort* bila ia melakukan rekursi pada bagian yang lebih kecil lebih dahulu), atau dibatasi oleh sesuatu yang kecil dan diketahui (20 cakram menara Hanoi, tinggi pohon seimbang), dan bila versi rekursifnya jelas lebih mudah dibuat benar. **Tabel 10.4** memberikan keputusan untuk algoritma-algoritma bab ini.

Tabel 10.4 Rekursi atau iterasi: keputusan untuk setiap algoritma bab ini.

Algoritma	Kedalaman	Rekomendasi
faktorial, m^n naif, jumlah $1..n$	n	gelung (rekursi ekor dengan akumulator, lalu gelung)
Fibonacci	n (dan waktu eksponensial bila naif)	dua variabel, atau *fast doubling* untuk n yang sangat besar
FPB Euklides, pemangkatan cepat, *binary search*	$O(\log n)$	rekursi boleh; gelungnya sama mudahnya
merge sort, maksimum dengan membelah dua	$O(\log n)$	rekursi
Menara Hanoi	n, dengan n sangat kecil	rekursi
panjang list, pencarian, penyalinan, cetak mundur	n	gelung, atau *stack* eksplisit bila urutannya penting
menu bersarang, pohon (Bab 11), direktori	kedalaman sarangnya	rekursi bila kedalamannya pasti kecil; *stack* eksplisit bila masukannya datang dari luar

10.13.2 Metode umumnya

Setiap rekursi dapat diubah menjadi gelung dengan *stack* milik kita sendiri, karena persis itulah yang dilakukan mesin untuk menjalankannya. Resepnya mencerminkan Subbab 10.3:

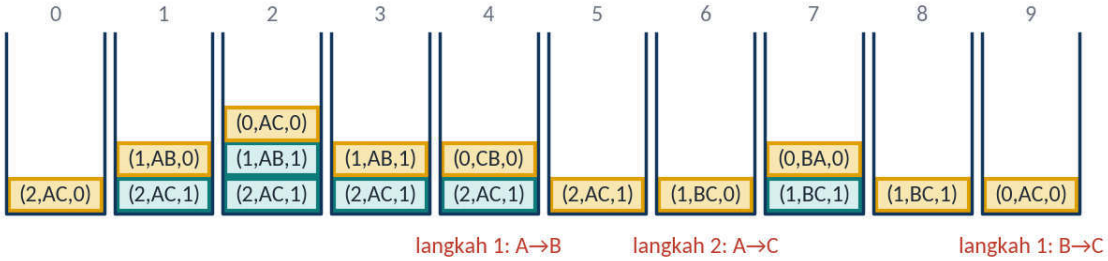
1. Definisikan rekaman `Frame` yang memuat semua yang perlu diingat oleh satu panggilan: parameternya, variabel lokal yang harus bertahan melewati panggilan rekursif, dan sebuah *tahap* (atau *program counter*) yang menyatakan di bagian mana badan fungsi panggilan itu harus dilanjutkan ketika sebuah panggilan rekursif kembali.
2. Ganti panggilan pertama dengan melakukan *push frame*-nya ke *stack* eksplisit, sebuah `std::vector`, `ArrayDeque`, atau list Python.
3. Ulangi selama *stack* tidak kosong: lihat *frame* teratas dan lanjutkan dari tahapnya. *Base case* melakukan *pop* terhadap *frame*-nya. Panggilan rekursif mencatat tahap berikutnya di *frame* saat ini lalu melakukan *push frame* baru.
4. Nilai kembalian, bila ada, diteruskan lewat sebuah variabel atau lewat medan *frame* di bawahnya.

Stack eksplisit itu tinggal di *heap*, sehingga ukurannya dibatasi oleh memori komputer, gigabita alih-alih megabita. Dan kita boleh menyimpan lebih sedikit daripada mesin: hanya medan yang

benar-benar kita perlukan, bukan alamat kembali dan register tersimpan dari sebuah *frame* sungguhan.

10.13.3 Hanoi dengan stack berisi frame

Untuk `HANOI` (**Pseudokode 10.5**) sebuah *frame* memerlukan n, ketiga nama tiang, dan sebuah tahap: tahap 0 berarti "baru dipanggil: lakukan panggilan rekursif pertama", tahap 1 berarti "panggilan rekursif pertama sudah kembali: lakukan langkahnya dan panggilan kedua". Panggilan kedua adalah *tail call*: begitu langkahnya dibuat, tidak ada lagi isi *frame* itu yang diperlukan. Maka, persis seperti yang dilakukan *tail-call elimination*, kita melakukan *pop* terhadap *frame* itu sebelum melakukan *push frame* panggilan kedua. Dengan begitu *stack* eksplisitnya tetap sependek kedalaman rekursi yang sebenarnya. Bagian pertama **Kode 10.18** mengimplementasikannya; **Gambar 10.12** memperlihatkan *stack* itu, digambar sebagai rangkap tiga (n, tiang asal dan tujuan, tahap), untuk $n = 2$.



Gambar 10.12 *Stack* eksplisit selama `hanoiIter(2)` berjalan; setiap *frame* memperlihatkan (n, tiang asal dan tujuan, tahap), dan *frame* teratas disorot. Setiap kolom adalah *stack* setelah satu putaran gelung; langkah-langkahnya tertera di bawah putaran yang membuatnya.

Ikutilah kolom-kolom **Gambar 10.12**. *Frame* (2, AC, 0) di-*push* pertama kali. Putaran 1 mengubah tahapnya menjadi 1 dan melakukan *push* panggilan rekursif pertama, (1, AB, 0), yang memindahkan satu cakram dari A ke B. Putaran 2 melakukan hal yang sama untuknya dan melakukan *push* (0, AC, 0), sebuah *base case*, yang di-*pop* oleh putaran 3. Pada putaran 4 *frame* (1, AB, 1) berada di puncak dengan tahap 1: ia di-*pop*, membuat langkahnya (cakram 1 dari A ke B), dan melakukan *push* panggilan keduanya (0, CB, 0). Demikian seterusnya: setelah putaran 9 *stack*-nya kosong dan ketiga langkah telah dibuat.

10.13.4 List panjang yang ditelusuri mundur

Bagian kedua **Kode 10.18** menelusuri list berisi 1.000.000 simpul secara mundur. `printBackward` rekursif dari Subbab 10.9 akan memerlukan sejuta *frame*, jauh lebih banyak daripada 116.000 yang muat di *stack* bawaan 8 MB. Dengan *stack* eksplisit, kita melakukan *push* pointer ke setiap simpul sambil berjalan maju, lalu melakukan *pop*, yang menghasilkan simpul-simpul dalam urutan terbalik. *Stack* itu adalah sebuah `vector` berisi sejuta pointer, 8 MB di *heap*, yang sama sekali tidak menjadi masalah.

Kode 10.18 `11_explicit_stack.cpp`: Menara Hanoi dengan *stack* eksplisit berisi *frame*, dan list berisi sejuta simpul yang ditelusuri mundur.

```cpp
// 11_explicit_stack.cpp -- recursion replaced by a stack that we manage ourselves:
// the Towers of Hanoi with a stack of frames, and a backward walk of a long list.
#include <cstdio>
#include <vector>
using namespace std;

struct Frame {  // everything one call of hanoi() needs to remember
    int n;
    char from, to, via;
    int stage;  // 0: just called, 1: first half done
};

long long hanoiIter(int n, bool print, size_t &maxStack) {
    vector<Frame> st;  // our own call stack
    long long moves = 0;
    st.push_back({n, 'A', 'C', 'B', 0});
    maxStack = 1;
    while (!st.empty()) {
        Frame &f = st.back();
        if (f.n == 0) {  // base case: return at once
            st.pop_back();
        } else if (f.stage == 0) {  // first recursive call
            f.stage = 1;
            Frame sub = {f.n - 1, f.from, f.via, f.to, 0};
            st.push_back(sub);  // f is not used after this: push_back may move it
        } else {                // the first half has returned
            Frame done = f;
            st.pop_back();  // nothing is left to do after the second call,
            ++moves;        // so this frame can go before it is made
            if (print)
                printf("%2lld. Move disk %d from rod %c to rod %c\n", moves, done.n,
                       done.from, done.to);
            st.push_back({done.n - 1, done.via, done.to, done.from, 0});
        }
        if (st.size() > maxStack) maxStack = st.size();
    }
    return moves;
}

struct Node {
    int data;
    Node *next;
};
```

Kode 10.18 (Lanjutan) *11_explicit_stack.cpp: Menara Hanoi dengan stack eksplisit berisi frame, dan list berisi sejuta simpul yang ditelusuri mundur.*

```
int main() {
    size_t maxStack;
    printf("Three disks, with an explicit stack:\n");
    hanoiIter(3, true, maxStack);
    for (int n : {10, 16, 20}) {
        long long m = hanoiIter(n, false, maxStack);
        printf("n = %2d: %7lld moves, at most %zu frames on our stack\n", n, m,
               maxStack);
    }

    const int N = 1000000;  // far too deep for a recursive walk (Section 10.12)
    Node *head = nullptr;
    for (int i = N; i >= 1; --i) head = new Node{i, head};  // 1, 2, ..., N
    vector<const Node *> st;
    for (const Node *p = head; p != nullptr; p = p->next) st.push_back(p);
    printf("\nThe list 1..%d visited backward starts with:", N);
    for (int k = 0; k < 5; ++k) {
        printf(" %d", st.back()->data);
        st.pop_back();
    }
    printf("\n");
    while (head != nullptr) {  // release the list, iteratively as well
        Node *next = head->next;
        delete head;
        head = next;
    }
    return 0;
}
```

```
Three disks, with an explicit stack:
 1. Move disk 1 from rod A to rod C
 2. Move disk 2 from rod A to rod B
 3. Move disk 1 from rod C to rod B
 4. Move disk 3 from rod A to rod C
 5. Move disk 1 from rod B to rod A
 6. Move disk 2 from rod B to rod C
 7. Move disk 1 from rod A to rod C
n = 10:    1023 moves, at most 11 frames on our stack
n = 16:   65535 moves, at most 17 frames on our stack
n = 20: 1048575 moves, at most 21 frames on our stack

The list 1..1000000 visited backward starts with: 1000000 999999 999998 999997 999996
```

Langkah-langkah untuk tiga cakram identik dengan langkah-langkah pada **Kode 10.8**, dan untuk n cakram *stack* eksplisitnya tidak pernah memuat lebih dari $n + 1$ *frame*, sama dengan kedalaman rekursinya ditambah *frame base case*-nya. Program itu juga diperiksa dengan AddressSanitizer, yang tidak menemukan galat. Versi Java dan Python, `ExplicitStack.java` dan `11_explicit_stack.py`, mencetak baris-baris yang sama; di Python *stack* eksplisitnya adalah list biasa, dan penelusuran sejuta simpul yang sebagai rekursi akan memerlukan `sys.setrecursionlimit` berjalan sebagai gelung di dalam batas bawaan.

> **WAWASAN KUNCI:** Rekursi tidak menambahkan kemampuan yang tidak dimiliki gelung; rekursi apa pun dapat diganti dengan gelung dan sebuah *stack*, dan mesin itu sendiri melakukan persis hal tersebut. Yang ditambahkan rekursi adalah *kejernihan*: mesin memelihara *stack* itu untuk Anda, dengan benar, setiap kali. Ubahlah menjadi *stack* eksplisit bila kedalamannya mungkin besar, dan pertahankan rekursinya bila tidak.

10.14 Profitina dalam Praktik

Profitina adalah platform visibilitas AI (GEO): ia mengukur seberapa siap situs web sebuah merek untuk dikutip oleh mesin penjawab AI, berdasarkan sinyal-sinyal seperti konten yang dapat diekstraksi mesin, data terstruktur, dan kredibilitas teknis, lalu membantu memperbaiki apa yang ditemukannya [24]. Servernya adalah aplikasi Python yang dibangun dengan FastAPI. Beberapa tugasnya memiliki bentuk yang persis sama dengan yang dipelajari dalam bab ini; kami menguraikan bentuk masalahnya dan kelas penyelesaiannya, bukan implementasi produk itu sendiri.

- **Menjelajahi sebuah situs.** Sebuah situs web adalah himpunan halaman yang dihubungkan oleh tautan, dan "kunjungi sebuah halaman, lalu kunjungi setiap halaman yang ditautkannya" adalah deskripsi rekursif sebuah penjelajahan (*crawl*). Bila ditulis sebagai fungsi rekursif, deskripsi itu juga berbahaya: struktur tautan sebuah situs dapat sedalam ribuan halaman (rantai paginasi "berikutnya, berikutnya, berikutnya…", kalender yang menautkan setiap hari berikutnya), dan setiap tingkat akan menjadi satu *frame* Python, sehingga batas bawaan 1.000 cepat tercapai. Karena itu sebuah *crawler* memelihara *stack* atau antreannya sendiri berisi halaman yang masih harus dikunjungi, bersama himpunan halaman yang sudah dilihat dan batas kedalaman: pola *stack* eksplisit dari Subbab 10.13, dengan urutan kunjungan sebagai pilihan rancangan (Bab 13 mempelajari persis pilihan ini sebagai *depth-first search* dan *breadth-first search*).
- **Data terstruktur yang bersarang.** Data terstruktur yang dibaca mesin penjawab, deskripsi schema.org yang disematkan pada halaman sebagai JSON-LD, bersifat bersarang: sebuah `Organization` memuat `PostalAddress`, sebuah `Article` memiliki `publisher` berupa `Organization` dengan sebuah `logo`, dan seterusnya sampai kedalaman berapa pun. Memeriksanya adalah penelusuran rekursif, satu fungsi yang menangani sebuah objek dengan menangani setiap medannya, persis seperti `countServices` di bawah ini. Karena masukannya datang dari luar, penelusuran itu memerlukan penjaga kedalaman: halaman yang jahat atau rusak dapat menyarangkan larik sampai seratus ribu tingkat, dan pengurai atau pemeriksa rekursif kemudian gagal dengan `RecursionError` (modul `json` milik Python sendiri berbuat demikian pada masukan seperti itu).
- **Mengingat hasil.** Sebagian pekerjaan penilaian mahal dan diminta berulang kali untuk masukan yang sama. Mengingat hasil berdasarkan masukannya, yaitu memoisasi dari Subbab 10.5.3, dengan tembolok *least-recently-used* yang berbatas dari Subbab 9.9, menghindarkan penghitungan ulang; aturannya sama dengan Fibonacci: memoisasi hanya menguntungkan bila submasalah yang sama benar-benar berulang.

Versi yang dapat dijalankan dari gagasan-gagasan ini memakai studi kasus buku ini, Profitina Laundry. Menu layanannya bersarang: kategori memuat layanan dan kategori yang lebih kecil, seperti diperlihatkan **Gambar 10.13**. Menu itu adalah sebuah *pohon*, pokok bahasan Bab 11, dan merupakan masukan yang wajar bagi program rekursif.



Gambar 10.13 Menu layanan Profitina Laundry: kategori (hijau kebiruan) memuat layanan (kuning) dan kategori yang lebih kecil, sampai kedalaman berapa pun.

Kode 10.19 menyimpan setiap entri menu sebagai `Item` dengan nama, harga (0 untuk kategori), dan daftar anak. Empat fungsi rekursif pendek menjawab empat pertanyaan: `printMenu` mencetak menu sebagai kerangka menjorok (entrinya dahulu, lalu anak-anaknya, satu tingkat lebih dalam), `countServices` menghitung layanan (sebuah layanan bernilai 1; sebuah kategori bernilai jumlah atas anak-anaknya), `height` mengukur seberapa dalam sarangnya, dan `cheapest` menemukan layanan termurah. Akhirnya `countServicesIter` menghitung layanan sekali lagi dengan *stack* eksplisit, seperti yang akan dilakukan sebuah *crawler* atau pemeriksa JSON bila masukannya tidak tepercaya.

Kode 10.19 `12_laundry_menu.cpp`: fungsi-fungsi rekursif atas menu layanan Profitina Laundry yang bersarang, dan penghitungan yang sama dengan *stack* eksplisit.

```cpp
// 12_laundry_menu.cpp -- Profitina Laundry's service menu is a nested structure:
// a category holds services and smaller categories. Recursion walks it naturally;
// an explicit stack walks it without using the call stack.
#include <algorithm>
#include <cstdio>
#include <string>
#include <utility>
#include <vector>
using namespace std;

struct Item {
    string name;
    long price;             // rupiah; 0 for a category
    vector<Item> children;  // empty for a service
};

Item service(string name, long price) { return Item{move(name), price, {}}; }
Item category(string name,
              vector<Item> kids) { return Item{move(name), 0, move(kids)}; }

void printMenu(const Item &it, int depth) {  // pre-order: item, then children
    if (it.children.empty())
        printf("%*s%-*s Rp%6ld\n", 2 * depth, "", 30 - 2 * depth, it.name.c_str(),
               it.price);
    else
        printf("%*s%s\n", 2 * depth, "", it.name.c_str());
    for (const Item &c : it.children) printMenu(c, depth + 1);
}

int countServices(const Item &it) {
    if (it.children.empty()) return 1;  // base case: a service
    int total = 0;
    for (const Item &c : it.children) total += countServices(c);
    return total;
}

int height(const Item &it) {  // levels below the top
    int h = 0;
    for (const Item &c : it.children) h = max(h, 1 + height(c));
    return h;
}
```

Kode 10.19 (Lanjutan) *12_laundry_menu.cpp: fungsi-fungsi rekursif atas menu layanan Profitina Laundry yang bersarang, dan penghitungan yang sama dengan stack eksplisit.*

```
const Item *cheapest(const Item &it) {  // the cheapest service
    if (it.children.empty()) return &it;
    const Item *best = nullptr;
    for (const Item &c : it.children) {
        const Item *b = cheapest(c);
        if (best == nullptr || b->price < best->price) best = b;
    }
    return best;
}

int countServicesIter(const Item &root) {  // the same count, no recursion
    vector<const Item *> st = {&root};
    int total = 0;
    while (!st.empty()) {
        const Item *it = st.back();
        st.pop_back();
        if (it->children.empty()) ++total;
        for (const Item &c : it->children) st.push_back(&c);
    }
    return total;
}

int main() {
    Item menu = category("Profitina Laundry", {
        category("Wash by weight (per kg)", {
            service("Regular, 3 days", 7000),
            service("Express, 1 day", 12000),
            service("Lightning, 6 hours", 18000)}),
        category("Per item", {
            category("Clothing", {
                service("Shirt", 8000),
                service("Suit", 35000),
                service("Evening dress", 45000)}),
            category("Household", {
                service("Bed cover", 30000),
                service("Blanket", 25000),
                service("Carpet (per m2)", 20000)})}),
        service("Ironing only (per kg)", 5000)});
```

Kode 10.19 (Lanjutan) *12_laundry_menu.cpp: fungsi-fungsi rekursif atas menu layanan Profitina Laundry yang bersarang, dan penghitungan yang sama dengan stack eksplisit.*

```
    printMenu(menu, 0);
    const Item *c = cheapest(menu);
    printf("\nservices: %d recursively, %d with an explicit stack\n",
           countServices(menu), countServicesIter(menu));
    printf("levels below the top: %d\n", height(menu));
    printf("cheapest service: %s, Rp%ld\n", c->name.c_str(), c->price);
    return 0;
}
```

```
Profitina Laundry
  Wash by weight (per kg)
    Regular, 3 days            Rp  7000
    Express, 1 day             Rp 12000
    Lightning, 6 hours         Rp 18000
  Per item
    Clothing
      Shirt                    Rp  8000
      Suit                     Rp 35000
      Evening dress            Rp 45000
    Household
      Bed cover                Rp 30000
      Blanket                  Rp 25000
      Carpet (per m2)          Rp 20000
  Ironing only (per kg)        Rp  5000

services: 10 recursively, 10 with an explicit stack
levels below the top: 3
cheapest service: Ironing only (per kg), Rp5000
```

Setiap fungsi rekursif memiliki *base case* "sebuah layanan, tanpa anak" dan *recursive case* "sebuah kategori: gabungkan jawaban anak-anaknya". Tidak satu pun perlu tahu seberapa dalam menunya, dan itulah kekuatan rekursi pada data bersarang; versi *stack* eksplisit memberikan hitungan yang sama tanpa memakai *call stack* sama sekali, dan itulah kekuatannya bila kedalaman tidak berada dalam kendali kita. Versi Java menyimpan entri dalam sebuah `record`, dan versi Python dalam sebuah `dataclass`; keduanya mencetak kerangka yang sama.

10.15 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Tidak ada *base case*, atau *base case* yang tidak pernah tercapai. `fact(n)` dengan uji `n == 0` berputar selamanya untuk argumen negatif, dan fungsi yang mengurangkan 2 dari argumen ganjil melompati 0. Ujilah masukan terkecil (0, 1, 2, dan nilai negatif apa pun yang mungkin diterima fungsi itu) sebelum yang lain, dan tulislah uji *base case* yang mencakup segala sesuatu di bawah masukan sah terkecil (`n <= 1`, bukan `n == 1`).

Panggilan rekursif yang tidak mengecilkan masalah. Melewatkan `n` alih-alih `n - 1`, atau `lo, hi` alih-alih `lo, mid - 1`, melakukan rekursi pada masalah yang sama selamanya. Untuk setiap panggilan rekursif, sebutkan besaran yang berkurang (Subbab 10.2.5).

Mengharapkan *stack overflow* bersikap sopan. Di C++ ia mematikan program tanpa pesan (Subbab 10.3.4). Ketahuilah kedalaman rekursi Anda: n untuk mengupas satu elemen, $\lg n$ untuk membelah dua.

Rekursi atas list panjang. `length`, `printBackward`, dan kawan-kawannya memerlukan satu *frame* per simpul; sejuta simpul meluapkan setiap *stack* bawaan (Subbab 10.9.3 dan 10.12). Pakailah gelung atau *stack* eksplisit.

Rekursi naif dengan submasalah yang tumpang tindih. `fibNaive(40)` melakukan 331 juta panggilan (Persamaan (10.7)). Bila argumen yang sama berulang, pakailah memoisasi, atau hitunglah secara *bottom-up* (Subbab 10.5).

Memanggil dua kali apa yang semestinya dipanggil sekali. Menulis `power(m, n/2) * power(m, n/2)` melakukan dua panggilan identik dan menghilangkan semua manfaat pembelahan dua; simpan hasilnya dalam sebuah variabel (Subbab 10.7.2).

Memercayai bilangan titik mengambang untuk jawaban eksak. Rumus Binet dalam `double` salah mulai $F(71)$ (Subbab 10.5.7).

Bergantung pada *tail-call elimination*. Hal itu terjadi pada `-O2` di GCC dan Clang, tidak pada `-O0`, tidak pernah di Java atau CPython, dan tidak pernah dijamin oleh standar C++ (Subbab 10.11).

Melupakan luapan. 21! dan $F(93)$ tidak muat dalam 64 bit, dan C++ tidak memperingatkan (Subbab 10.3.3 dan 10.5.8). Pakailah bilangan bulat Python, `BigInteger`, atau sebuah modulus.

Menaikkan batas rekursi Python tanpa berpikir. `sys.setrecursionlimit(10**9)` menghapus perlindungannya, bukan *stack*-nya. Naikkan batas itu hanya sejauh yang benar-benar diperlukan rekursinya, dan untuk rekursi yang sangat dalam jalankan di *thread* dengan *stack* yang lebih besar, atau ubahlah menjadi gelung (Subbab 10.12.3 dan 10.13).

Kehilangan nilai kembalian. Fungsi rekursif yang mengembalikan *head* list yang baru, seperti `removeAll`, harus dipanggil sebagai `head = removeAll(head, key)`; memanggilnya sebagai `removeAll(head, key)` membiarkan `head` menunjuk ke simpul yang mungkin sudah dihapus.

10.16 Ringkasan Bab

- **Definisi rekursif** atau **fungsi rekursif** mendefinisikan sebuah jawaban berdasarkan jawaban untuk contoh yang benar-benar lebih kecil. Ia memerlukan ***base case*** yang menjawab secara langsung dan ***recursive case*** yang maju menuju *base case* itu.
- **Induksi** membuktikan kebenaran fungsi rekursif: periksa *base case*-nya, anggaplah panggilan yang lebih kecil benar, dan periksa penggabungannya. Bentuk tertutup seperti $f(n) = 6 \cdot 2^n - 3$ (Persamaan (10.2)) dibuktikan dengan cara yang sama.
- Setiap panggilan yang tertunda adalah sebuah ***stack frame***. *Stack* menggulung sampai *base case* dan mengurai ketika panggilan-panggilan kembali (**Gambar 10.2**); waktu sebuah rekursi adalah kerja semua panggilannya, dan memori tambahannya adalah banyaknya *frame* terbesar yang ada pada saat yang sama.
- **Faktorial** dengan gelung maupun dengan rekursi sama-sama memakan waktu $\Theta(n)$ (Persamaan (10.4)); gelung memakai memori $O(1)$, rekursi $\Theta(n)$. 20! adalah faktorial terbesar yang muat dalam 64 bit.
- Rekursi naif **Fibonacci** melakukan $2F(n+1) - 1$ panggilan, $\Theta(\varphi^n)$ (Persamaan (10.7), Persamaan (10.8)): 331.160.281 panggilan untuk $F(40)$. Memoisasi melakukan $2n - 1$ panggilan, gelung *bottom-up* n penjumlahan dengan memori $O(1)$, *fast doubling* $\lfloor \lg n \rfloor + 2$ panggilan (Persamaan (10.9)), dan rumus Binet (Persamaan (10.10)) dalam `double` hanya eksak sampai $F(70)$ (**Tabel 10.1**).
- **Algoritma Euklides** $\gcd(a, b) = \gcd(b, a \bmod b)$ (Persamaan (10.11)) memerlukan paling banyak sekitar $1 + \log_{\varphi} b$ pembagian (Lamé, Persamaan (10.12)); bilangan Fibonacci yang berurutan adalah kasus terburuknya.
- **Pemangkatan cepat** membelah dua pangkatnya: $\lfloor \lg n \rfloor + 2$ panggilan (Persamaan (10.15)) alih-alih $n + 1$, dan dengan modulus ia menghitung $7^{10^{18}} \bmod (10^9 + 7)$ dalam 61 panggilan.
- **Menara Hanoi** rekursif membuat $M(n) = 2^n - 1$ langkah (Persamaan (10.16)), dan tidak ada algoritma yang dapat membuat lebih sedikit: ongkos eksponensialnya milik masalahnya.
- **Larik** direkursikan dengan mengupas satu elemen (kedalaman n) atau membelah dua (kedalaman $\lg n$); **linked list** direkursikan pada simpul pertama dan sisanya, sehingga mencetak mundur hanya memerlukan perubahan satu baris dan memakan satu *frame* per simpul.
- **Pohon rekursi** menjumlahkan kerja aras demi aras: $T(n) = 2T(n/2) + cn$ menghasilkan $\Theta(n \lg n)$ (Persamaan (10.17)), dan rekurens lazim lainnya ada di **Tabel 10.2**.

- ***Tail call*** adalah panggilan yang hasilnya dikembalikan tanpa diubah. GCC dengan `-O2` mengompilasi `sumAcc` yang rekursif ekor, dan bahkan `sumPlain` yang berakumulasi, menjadi gelung, tetapi tidak dengan `-O0`; JVM dan CPython tidak pernah melakukannya. Jangan bergantung padanya; `[[gnu::musttail]]` dan `[[clang::musttail]]` tersedia untuk kode yang harus bergantung padanya.
- **Batas kedalaman** (**Tabel 10.3**, Persamaan (10.18)): sekitar 116.000 *frame* 72 bita dalam *stack* C++ 8 MB; 10.000 sampai 60.000 dalam *stack thread* bawaan Java 1 MB; 1.000 di Python secara bawaan. Setiap batas dapat dinaikkan, dan setiap bahasa menyediakan *thread* dengan *stack* yang lebih besar.
- **Setiap rekursi dapat menjadi gelung dengan *stack* eksplisit** berisi *frame* yang memuat parameter, variabel lokal yang bertahan, dan sebuah tahap (**Gambar 10.12**); lakukan ini bila kedalamannya mungkin besar (**Tabel 10.4**).

10.17 Soal Latihan

10.1 Fungsi g didefinisikan oleh $g(0) = 0$ dan $g(n) = g(n-1) + 2n - 1$ untuk $n \geq 1$. Hitunglah $g(1)$ sampai $g(5)$ dengan menguraikan definisinya, tebaklah sebuah bentuk tertutup, dan buktikan dengan induksi.

10.2 Gambarlah *call stack*, seperti pada **Gambar 10.2**, pada saat `f(0)` akan kembali selama perhitungan `f(3)` di **Kode 10.1**. Berapa banyak *frame* `f` yang ada di *stack*, dan berapa nilai `n` pada masing-masing?

10.3 Tulislah fungsi rekursif `digitSum(n)` yang mengembalikan jumlah digit desimal sebuah bilangan bulat tak negatif, misalnya digitSum(2026) $= 2 + 0 + 2 + 6 = 10$. Sebutkan *base case*-nya, dan berikan waktu jalannya dalam banyaknya digit.

10.4 Tulislah fungsi rekursif `printBinary(n)` yang mencetak representasi biner sebuah bilangan bulat tak negatif, misalnya `1101` untuk 13. Jelaskan mengapa panggilan rekursifnya harus diletakkan *sebelum* pernyataan `print`.

10.5 Tulislah fungsi rekursif `isSorted(A, n)` yang memutuskan apakah $A[0..n-1]$ terurut naik. Apa *base case*-nya, waktu jalannya, dan kedalaman rekursinya?

10.6 Buktikan dengan induksi bahwa rekursi naif menghitung $F(k)$ tepat $F(n-k+1)$ kali ketika menghitung $F(n)$, untuk $1 \leq k \leq n$. Periksalah klaim itu terhadap keluaran **Kode 10.5** untuk $n = 6$.

10.7 *Fungsi 91* John McCarthy didefinisikan untuk bilangan bulat oleh $M(n) = n - 10$ bila $n > 100$, dan $M(n) = M(M(n+11))$ bila tidak [25]. Programkan fungsi itu, hitung panggilannya untuk beberapa argumen, dan tunjukkan bahwa $M(n) = 91$ untuk setiap $n \leq 100$ yang Anda uji. Mengapa bukti bahwa fungsi itu selalu berhenti lebih sulit daripada untuk fungsi lain mana pun dalam bab ini?

10.8 Algoritma Euklides *yang diperluas* mengembalikan, selain $g = \gcd(a, b)$, bilangan bulat x dan y dengan $ax + by = g$. Turunkan *recursive case*-nya dari Persamaan (10.11): bila $bx_1 + (a \bmod b)y_1 = g$, nyatakan x dan y melalui x_1 dan y_1. Programkan dan temukan x, y untuk (228,90) dan (1071,462).

10.9 (a) Hitunglah $3^{13} \bmod 7$ dengan tangan memakai pengkuadratan berulang, sambil mereduksi modulo 7 setelah setiap perkalian. (b) Programkan pemangkatan cepat modular dan hitunglah $7^{222} \bmod 1000$. (c) Seorang mahasiswa menulis *recursive case* **Pseudokode 10.4**

sebagai `return power(m, n/2) * power(m, n/2) * (n % 2 == 1 ? m : 1);`. Tuliskan rekurens banyaknya panggilan yang dilakukan versi ini bila n perpangkatan dua, lalu selesaikan.

10.10 Dalam penyelesaian Menara Hanoi untuk n cakram, berapa kali cakram k berpindah ($1 \le k \le n$)? Buktikan jawaban Anda dengan induksi pada n, dan periksa bahwa totalnya atas semua cakram adalah $2^n - 1$.

10.11 Tulislah fungsi-fungsi rekursif yang mencetak (a) semua himpunan bagian dari {1,2,3} dan (b) semua permutasi huruf-huruf `ABC`. Berapa baris yang dicetak masing-masing untuk himpunan berisi n elemen, dan seberapa dalam masing-masing rekursinya?

10.12 Untuk setiap fungsi berikut, katakan apakah panggilan rekursifnya *tail call*, dan bila bukan, tulis ulang dengan akumulator agar menjadi *tail call*: (a) `factRec` dari **Kode 10.2**; (b) `gcdRec` dari **Kode 10.6**; (c) `length` dari **Kode 10.11**; (d) `printBackward` dari **Kode 10.11**. Untuk fungsi mana hal itu mustahil tanpa struktur data tambahan, dan mengapa?

10.13 Sebuah fungsi rekursif C++ memiliki *frame* berukuran 48 bita. (a) Dengan Persamaan (10.18), taksirlah rekursi terdalam yang mungkin pada *stack* Linux 8 MB dan pada *stack* Windows 1 MB. (b) Sebuah program Python memanggil fungsi rekursif pada list berisi 5.000 elemen dan gagal dengan `RecursionError`. Berikan dua cara memperbaikinya, dan katakan mana yang Anda pilih.

10.14 Sebuah robot pada kisi $r \times c$ petak hanya bergerak ke kanan atau ke bawah, dari pojok kiri atas ke pojok kanan bawah. Banyaknya lintasan memenuhi $P(r,c) = P(r-1,c) + P(r,c-1)$ dengan $P(0,c) = P(r,0) = 1$ (menghitung gerakan yang masih harus dibuat). Programkan rekursi naif dan rekursi bermemo, lalu bandingkan banyaknya panggilan untuk $r = c = 2,8,12$. Bentuk tertutup apa yang disarankan oleh banyaknya lintasan itu?

10.15 Tulis ulang `printBackward` dari **Kode 10.11** tanpa rekursi, dengan memakai *stack* eksplisit. Berapa banyak memori tambahan yang dipakainya untuk list berisi n simpul, dibandingkan dengan versi rekursifnya?

10.18 Penyelesaian Soal Latihan

10.1 Penguraian: $g(1) = 0 + 1 = 1$, $g(2) = 1 + 3 = 4$, $g(3) = 4 + 5 = 9$, $g(4) = 9 + 7 = 16$, $g(5) = 16 + 9 = 25$. Nilai-nilainya kuadrat, jadi tebaklah $g(n) = n^2$. *Base case*: $g(0) = 0 = 0^2$. Langkah induktif: bila $g(n-1) = (n-1)^2$, maka $g(n) = (n-1)^2 + 2n - 1 = n^2 - 2n + 1 + 2n - 1 = n^2$. ▫ (Definisi itu menyatakan bahwa jumlah n bilangan ganjil pertama adalah n^2.)

10.2 Pada saat itu *stack* memuat, dari bawah ke atas, *frame* `main`, `fTraced(3, 0)`, `fTraced(2, 1)`, `fTraced(1, 2)`, dan `fTraced(0, 3)`: empat *frame* fungsi itu, dengan $n = 3,2,1,0$. *Frame* dengan $n = 3,2,1$ masing-masing sedang menunggu di dalam pernyataan `long long smaller = fTraced(n - 1, depth + 1);` agar panggilannya kembali; `fTraced(0, 3)` telah mengisi `result = 3` dan sedang mencetak baris kembalinya. Jejak **Kode 10.1** memperlihatkan hal yang sama: empat baris `call` telah muncul dan baru satu baris `returns`.

Program untuk 10.3, 10.4, 10.5, 10.7, 10.8, 10.9, 10.11, 10.14, dan 10.15 dikumpulkan dalam **Kode 10.20**; keluarannya menyusul setelah listing itu.

Kode 10.20 `13_exercises.cpp`: penyelesaian soal-soal latihan pemrograman Bab 10.

```cpp
// 13_exercises.cpp -- solutions to the programming exercises of Chapter 10.
#include <cstdio>
#include <string>
#include <vector>
using namespace std;
typedef long long ll;

int digitSum(ll n) {  // Exercise 10.3
    if (n < 10) return (int)n;
    return (int)(n % 10) + digitSum(n / 10);
}

void printBinary(ll n) {             // Exercise 10.4
    if (n >= 2) printBinary(n / 2);  // the higher bits first
    printf("%lld", n % 2);
}

bool isSorted(const int A[], int n) {  // Exercise 10.5
    if (n <= 1) return true;
    return A[n - 2] <= A[n - 1] && isSorted(A, n - 1);
}

ll m91calls = 0;
int mc91(int n) {  // Exercise 10.7
    ++m91calls;
    if (n > 100) return n - 10;
    return mc91(mc91(n + 11));
}

ll extGcd(ll a, ll b, ll &x, ll &y) {  // Exercise 10.8: a x + b y = gcd
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    ll x1, y1;
    ll g = extGcd(b, a % b, x1, y1);  // b x1 + (a mod b) y1 = g
    x = y1;
    y = x1 - (a / b) * y1;
    return g;
}
```

Kode 10.20 (Lanjutan) *13_exercises.cpp: penyelesaian soal-soal latihan pemrograman Bab 10.*

```
ll powMod(ll m, ll n, ll mod) {  // Exercise 10.9
    if (n == 0) return 1 % mod;
    ll half = powMod(m, n / 2, mod);
    ll sq = half * half % mod;
    return n % 2 == 0 ? sq : sq * (m % mod) % mod;
}

// Exercise 10.11
void subsets(const vector<int> &s, size_t i, vector<int> &chosen) {
    if (i == s.size()) {
        printf("{");
        for (size_t k = 0; k < chosen.size(); ++k) printf(k ? ", %d" : "%d",
             chosen[k]);
        printf("} ");
        return;
    }
    subsets(s, i + 1, chosen);  // without s[i]
    chosen.push_back(s[i]);
    subsets(s, i + 1, chosen);  // with s[i]
    chosen.pop_back();
}

void permutations(string &s, size_t i) {  // Exercise 10.11
    if (i == s.size()) {
        printf("%s ", s.c_str());
        return;
    }
    for (size_t j = i; j < s.size(); ++j) {
        swap(s[i], s[j]);  // choose s[j] for position i
        permutations(s, i + 1);
        swap(s[i], s[j]);  // undo the choice
    }
}

ll gridCalls = 0;
ll paths(int r, int c) {  // Exercise 10.14, naive
    ++gridCalls;
    if (r == 0 || c == 0) return 1;
    return paths(r - 1, c) + paths(r, c - 1);
}
ll pathsMemo(int r, int c, vector<vector<ll>> &memo) {
    ++gridCalls;
    if (r == 0 || c == 0) return 1;
    if (memo[r][c] != 0) return memo[r][c];
    return memo[r][c] = pathsMemo(r - 1, c, memo) + pathsMemo(r, c - 1, memo);
}
```

Kode 10.20 (Lanjutan) *13_exercises.cpp: penyelesaian soal-soal latihan pemrograman Bab 10.*

```
struct Node {
    int data;
    Node *next;
};

void printBackwardIter(const Node *head) {  // Exercise 10.15
    vector<const Node *> st;
    for (const Node *p = head; p != nullptr; p = p->next) st.push_back(p);
    while (!st.empty()) {
        printf(" %d", st.back()->data);
        st.pop_back();
    }
    printf("\n");
}

int main() {
    printf("10.3  digitSum(2026) = %d, digitSum(9876543210) = %d\n",
           digitSum(2026), digitSum(9876543210LL));
    printf("10.4  13 = ");
    printBinary(13);
    printf(", 2026 = ");
    printBinary(2026);
    printf("\n");
    int a[] = {4, 7, 12, 18, 25, 31}, b[] = {4, 7, 18, 12, 25, 31};
    printf("10.5  isSorted(a) = %s, isSorted(b) = %s\n",
           isSorted(a, 6) ? "true" : "false",
           isSorted(b, 6) ? "true" : "false");

    bool all91 = true;
    for (int n = -50; n <= 100; ++n) all91 = all91 && mc91(n) == 91;
    printf("10.7  M(n) = 91 for every n from -50 to 100: %s\n",
           all91 ? "yes" : "no");
    for (int n : {87, 99, 100, 101, 150}) {
        m91calls = 0;
        int v = mc91(n);
        printf("      M(%d) = %d, calls made: %lld\n", n, v, m91calls);
    }

    ll x, y, g = extGcd(228, 90, x, y);
    printf("10.8  gcd(228, 90) = %lld = 228 * (%lld) + 90 * (%lld)\n", g, x, y);
    g = extGcd(1071, 462, x, y);
    printf("      gcd(1071, 462) = %lld = 1071 * (%lld) + 462 * (%lld)\n", g, x, y);
```

Kode 10.20 (Lanjutan) *13_exercises.cpp: penyelesaian soal-soal latihan pemrograman Bab 10.*

```
    printf("10.9  3^13 mod 7 = %lld, 7^222 mod 1000 = %lld\n", powMod(3, 13, 7),
           powMod(7, 222, 1000));

    vector<int> s = {1, 2, 3}, chosen;
    printf("10.11 subsets: ");
    subsets(s, 0, chosen);
    string abc = "ABC";
    printf("\n      permutations: ");
    permutations(abc, 0);
    printf("\n");

    for (int n : {2, 8, 12}) {
        gridCalls = 0;
        ll naive = paths(n, n);
        ll naiveCalls = gridCalls;
        vector<vector<ll>> memo(n + 1, vector<ll>(n + 1, 0));
        gridCalls = 0;
        ll fast = pathsMemo(n, n, memo);
        printf("10.14 %2d x %-2d grid: %7lld paths; naive %7lld calls, "
               "memo %3lld calls%s\n", n, n, naive, naiveCalls, gridCalls,
               fast == naive ? "" : " MISMATCH");
    }

    Node *head = nullptr;
    int values[] = {50, 40, 30, 40, 20};
    for (int v : values) head = new Node{v, head};
    printf("10.15 backward:");
    printBackwardIter(head);
    while (head != nullptr) {
        Node *next = head->next;
        delete head;
        head = next;
    }
    return 0;
}
```

```
10.3  digitSum(2026) = 10, digitSum(9876543210) = 45
10.4  13 = 1101, 2026 = 11111101010
10.5  isSorted(a) = true, isSorted(b) = false
10.7  M(n) = 91 for every n from -50 to 100: yes
      M(87) = 91, calls made: 29
      M(99) = 91, calls made: 5
      M(100) = 91, calls made: 3
      M(101) = 91, calls made: 1
      M(150) = 140, calls made: 1
10.8  gcd(228, 90) = 6 = 228 * (2) + 90 * (-5)
      gcd(1071, 462) = 21 = 1071 * (-3) + 462 * (7)
10.9  3^13 mod 7 = 3, 7^222 mod 1000 = 49
10.11 subsets: {} {3} {2} {2, 3} {1} {1, 3} {1, 2} {1, 2, 3}
      permutations: ABC ACB BAC BCA CBA CAB
10.14  2 x 2  grid:       6 paths; naive      11 calls, memo   9 calls
10.14  8 x 8  grid:   12870 paths; naive   25739 calls, memo 129 calls
10.14 12 x 12 grid: 2704156 paths; naive 5408311 calls, memo 289 calls
10.15 backward: 50 40 30 40 20
```

10.3 `digitSum` di **Kode 10.20**: *base case* $n < 10$ (satu digit adalah jumlahnya sendiri); *recursive case*: digit terakhir, n mod 10, ditambah jumlah digit $\lfloor n/10 \rfloor$, yang digitnya kurang satu. Untuk bilangan dengan d digit, fungsi itu melakukan d panggilan: waktu dan *stack* $\Theta(d) = \Theta(\log n)$. Keluarannya memberi 10 untuk 2026 dan 45 untuk 9.876.543.210.

10.4 `printBinary(n)` lebih dahulu mencetak representasi biner $\lfloor n/2 \rfloor$ (semua digit kecuali yang terakhir) lalu digit terakhir, n mod 2. Digit biner terakhir adalah yang dapat kita hitung pertama kali, tetapi harus dicetak terakhir; meletakkan panggilan rekursif lebih dahulu menunda `print`

sampai semua digit yang lebih tinggi sudah dicetak, persis seperti `printBackward` menunda setiap simpul. Keluarannya memperlihatkan $13 = 1101_2$ dan $2026 = 11111101010_2$.

10.5 `isSorted(A, n)`: larik berisi 0 atau 1 elemen terurut (*base case*); larik yang lebih panjang terurut bila dua elemen terakhirnya berurutan dan $n-1$ elemen pertamanya terurut. Fungsi itu melakukan paling banyak n panggilan, berhenti lebih awal pada inversi pertama karena `&&` tidak mengevaluasi sisi kanannya bila sisi kirinya salah: waktu $O(n)$ dan kedalaman rekursi sampai $n-1$. Keluarannya memperlihatkan `true` untuk 4, 7, 12, 18, 25, 31 dan `false` untuk 4, 7, 18, 12, 25, 31.

10.6 Misalkan $N_n(k)$ banyaknya $F(k)$ dihitung di dalam `fibNaive(n)`, untuk $1 \le k \le n$. Kita buktikan $N_n(k) = F(n-k+1)$ dengan induksi pada n. Untuk $n = 1$: $N_1(1) = 1 = F(1)$. Untuk $n = 2$: $F(2)$ dihitung sekali dan $F(1)$ sekali, dan $F(1) = F(2) = 1$. Untuk $n \ge 3$: panggilan pada n menghitung $F(n)$ sekali, yaitu $F(1)$, dan selain itu setiap penghitungan terjadi di dalam `fibNaive(n-1)` atau `fibNaive(n-2)`, sehingga untuk $k \le n-1$, $N_n(k) = N_{n-1}(k) + N_{n-2}(k) = F(n-k) + F(n-k-1) = F(n-k+1)$, dengan $N_{n-2}(n-1) = 0 = F(0)$ yang konsisten dengan rumusnya. Untuk $n = 6$: $F(6)$ sekali ($F(1)$), $F(5)$ sekali ($F(2)$), $F(4)$ dua kali ($F(3)$), $F(3)$ tiga kali ($F(4)$), $F(2)$ lima kali ($F(5)$), $F(1)$ delapan kali ($F(6)$), persis seperti yang dicetak **Kode 10.5**. ($F(0)$, yang tidak tercakup oleh rumus itu, dihitung $F(n-1) = 5$ kali.)

10.7 Keluaran **Kode 10.20** menegaskan $M(n) = 91$ untuk setiap n dari -50 sampai 100, dengan 29 panggilan untuk $M(87)$, 5 untuk $M(99)$, 3 untuk $M(100)$, dan satu untuk argumen apa pun di atas 100. Kesulitannya adalah aturan 2 dari Subbab 10.2.5: argumen panggilan dalam, $n+11$, *lebih besar*, bukan lebih kecil, dan argumen panggilan luar adalah apa pun yang dikembalikan panggilan dalam. Tidak ada besaran sederhana yang berkurang pada setiap panggilan; bukti bahwa rekursinya berhenti (dengan induksi pada $101-n$, dengan langkah 11) adalah salah satu peragaan awal verifikasi program secara formal [25].

10.8 Tulis $a = qb + r$ dengan $q = \lfloor a/b \rfloor$ dan $r = a \bmod b$. Bila $bx_1 + ry_1 = g$, substitusikan $r = a - qb$: $bx_1 + (a-qb)y_1 = ay_1 + b(x_1 - qy_1) = g$. Jadi $x = y_1$ dan $y = x_1 - qy_1$, dengan *base case* $\gcd(a,0) = a = a \cdot 1 + 0 \cdot 0$, yaitu $x = 1$, $y = 0$. Keluarannya memberi $6 = 228 \cdot 2 + 90 \cdot (-5)$ dan $21 = 1071 \cdot (-3) + 462 \cdot 7$; keduanya terbukti: $456 - 450 = 6$ dan $-3213 + 3234 = 21$.

10.9 (a) $13 = 1101_2$. $3^1 \equiv 3$; $3^3 = (3^1)^2 \cdot 3 = 27 \equiv 6$; $3^6 = 6^2 = 36 \equiv 1$; $3^{13} = (3^6)^2 \cdot 3 \equiv 1 \cdot 3 = 3 \pmod 7$. (b) Program mencetak $7^{222} \bmod 1000 = 49$. (c) Dua panggilan identik itu memberi $C(n) = 2C(n/2) + 1$ dengan $C(1) = 1 + 2C(0) = 3$ dan $C(0) = 1$; pohon rekursinya memiliki 2^i simpul pada aras i untuk $\lg n + 2$ aras, sehingga $C(n) = 2^{\lfloor \lg n + 2 \rfloor} - 1 = 4n - 1$: linear, lebih buruk daripada $n+1$ metode naif. Pembelahan dua hanya membantu bila separuhnya dihitung sekali.

10.10 Cakram k berpindah 2^{n-k} kali. Induksi pada n: untuk $n = 1$, cakram 1 berpindah sekali, yaitu 2^0 kali. Untuk $n \ge 2$, cakram n berpindah sekali, yaitu 2^0 kali (baris 4 **Pseudokode 10.5**), dan setiap cakram $k < n$ hanya berpindah di dalam kedua panggilan rekursif pada $n-1$ cakram, $2^{(n-1)-k}$ kali di masing-masing menurut hipotesis, jadi 2^{n-k} kali seluruhnya. Totalnya $\sum_{k=1}^{n} 2^{n-k} = 2^{n-1} + \cdots + 2 + 1 = 2^n - 1$, yaitu Persamaan (10.16) sekali lagi. (Subbab 5.22, Penyelesaian 5.2(c), menemukan hitungan yang sama dari nomor-nomor langkahnya.)

10.11 `subsets` memutuskan untuk setiap elemen secara bergiliran apakah elemen itu ditinggalkan atau diambil, lalu melakukan rekursi pada elemen-elemen yang tersisa; di akhir setiap rantai n keputusan ia mencetak himpunan yang terpilih. Fungsi itu mencetak 2^n himpunan (8 untuk $n = 3$) dan melakukan rekursi sedalam n tingkat. `permutations` memilih setiap huruf

yang tersisa untuk posisi i dengan sebuah pertukaran, melakukan rekursi pada posisi $i + 1$, lalu menukar kembali ("membatalkan pilihan"), pola yang disebut *backtracking* (runut balik). Fungsi itu mencetak $n!$ susunan (6 untuk $n = 3$), juga pada kedalaman n. Keluarannya mendaftar `{}` `{3}` `{2}` `{2, 3}` `{1}` `{1, 3}` `{1, 2}` `{1, 2, 3}` dan ABC ACB BAC BCA CBA CAB.

10.12 (a) `factRec` tidak rekursif ekor (perkaliannya menunggu); dengan akumulator: `factAcc(n, acc)` mengembalikan `acc` bila $n = 0$ dan selain itu `factAcc(n - 1, acc * n)`, dipanggil sebagai `factAcc(n, 1)`. (b) `gcdRec` sudah rekursif ekor: hasilnya dikembalikan tanpa diubah. (c) `length` tidak (`1 +`-nya menunggu); `lengthAcc(p, acc)` mengembalikan `acc` untuk pointer nol dan selain itu `lengthAcc(p->next, acc + 1)`. (d) `printBackward` tidak, dan tidak dapat dijadikan rekursif ekor dengan bilangan sebagai akumulator: simpul-simpulnya harus dicetak dalam urutan kebalikan dari urutan dicapainya oleh rekursi, sehingga sesuatu harus mengingat semua simpul yang belum dicetak. Sesuatu itu adalah *call stack*, atau *stack* eksplisit (Soal Latihan 10.15), atau salinan list yang dibalik. Akumulator menghapus *operasi* yang tertunda; ia tidak dapat menghapus *data* yang tertunda.

10.13 (a) 8,388,608/48 ≈ 174,763 tingkat pada *stack* 8 MB dan 1,048,576/48 ≈ 21,845 pada *stack* 1 MB, dalam praktik sedikit kurang, karena `main` dan *runtime* memakai sebagian *stack*. (b) Naikkan batasnya, misalnya dengan `sys.setrecursionlimit(10_000)`, yang aman pada Python 3.11 dan sesudahnya untuk kedalaman 5.000; atau tulis ulang fungsinya sebagai gelung, yang sama sekali tidak memerlukan batas. Gelung adalah pilihan yang lebih baik: list itu bisa bertambah panjang, dan fungsi yang gagal pada list berisi 5.000 elemen akan gagal lagi pada list berisi 50.000.

10.14 Program mencetak 6 lintasan untuk kisi 2×2, 12.870 untuk 8×8, dan 2.704.156 untuk 12×12. Rekursi naif melakukan $2P(n, n) - 1$ panggilan (11; 25.739; 5.408.311), struktur yang sama dengan Persamaan (10.7), karena daun-daunnya tepat panggilan-panggilan yang mengembalikan 1. Versi bermemo melakukan 9, 129, dan 289 panggilan: setiap pasangan dalam (r, c) yang banyaknya n^2 dihitung sekali, dengan ongkos dua panggilan, ditambah panggilan pertama, jadi $2n^2 + 1$. Banyaknya lintasan adalah koefisien binomial: $P(n, n) = \binom{2n}{n}$, karena sebuah lintasan adalah pilihan n dari $2n$ gerakannya yang mengarah ke bawah: $\binom{4}{2} = 6$, $\binom{16}{8} = 12{,}870$, $\binom{24}{12} = 2{,}704{,}156$.

10.15 `printBackwardIter` di **Kode 10.20** melakukan *push* pointer ke setiap simpul sambil berjalan maju, lalu melakukan *pop* dan mencetaknya: 50 40 30 40 20 untuk list 20, 40, 30, 40, 50. Fungsi itu memakai n pointer, masing-masing 8 bita pada mesin 64 bit, di *heap*. Versi rekursifnya memakai $n + 1$ *frame* di *call stack*, masing-masing jauh lebih besar daripada sebuah pointer (paling sedikit alamat kembali, *frame pointer* tersimpan, dan parameternya, biasanya 32 sampai 64 bita). Karena itu *stack* eksplisitnya beberapa kali lebih kecil, dan yang terpenting, tidak dibatasi oleh ukuran *call stack*.

Daftar Pustaka

[1] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, Apr. 1960.

[2] E. W. Dijkstra, "Recursive programming," *Numerische Mathematik*, vol. 2, no. 1, pp. 312–318, 1960.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 2, 4, and 31.

[4] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, sec. 1.2.1 and 1.2.8.

[5] L. E. Sigler, *Fibonacci's Liber Abaci: A Translation into Modern English of Leonardo Pisano's Book of Calculation*. New York, NY, USA: Springer, 2002, pp. 404–405.

[6] D. Takahashi, "A fast algorithm for computing large Fibonacci numbers," *Information Processing Letters*, vol. 75, no. 6, pp. 243–246, Nov. 2000.

[7] J. P. M. Binet, "Mémoire sur l'intégration des équations linéaires aux différences finies, d'un ordre quelconque, à coefficients variables," *Comptes Rendus de l'Académie des Sciences*, vol. 17, pp. 559–567, 1843.

[8] T. L. Heath, *The Thirteen Books of Euclid's Elements*, 2nd ed., vol. 2. Cambridge, U.K.: Cambridge University Press, 1926, Book VII, Propositions 1–2.

[9] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, sec. 4.5.2, 4.5.3, and 4.6.3.

[10] G. Lamé, "Note sur la limite du nombre des divisions dans la recherche du plus grand commun diviseur entre deux nombres entiers," *Comptes Rendus de l'Académie des Sciences*, vol. 19, pp. 867–870, 1844.

[11] É. Lucas, *Récréations mathématiques*, vol. 3. Paris, France: Gauthier-Villars, 1893, pp. 55–59 ("La Tour d'Hanoï").

[12] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1994, ch. 1 and sec. 6.6.

[13] G. L. Steele Jr., "Debunking the 'expensive procedure call' myth, or, procedure call implementations considered harmful, or, LAMBDA: The Ultimate GOTO," in *Proc. 1977 ACM Annual Conference*, Seattle, WA, USA, Oct. 1977, pp. 153–162.

[14] GNU Project, "Options that control optimization," *Using the GNU Compiler Collection (GCC)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. [Accessed: 24-Sep-2026].

[15] GNU Project, "Statement attributes: musttail," *Using the GNU Compiler Collection (GCC 15.1)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-15.1.0/gcc/Statement-Attributes.html. [Accessed: 24-Sep-2026].

[16] GNU Project, "GCC 16 release series." [Online]. Available: https://gcc.gnu.org/gcc-16/. [Accessed: 24-Sep-2026].

[17] G. van Rossum, "Tail recursion elimination," *Neopythonic* (blog), Apr. 22, 2009. [Online]. Available: http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html

[18] Microsoft, "/STACK (Stack allocations)," *Microsoft Learn — C/C++ Build Reference*. [Online]. Available: https://learn.microsoft.com/en-us/cpp/build/reference/stack-stack-allocations. [Accessed: 24-Sep-2026].

[19] Oracle, "The java command: -Xss," *Java SE 25 Tools Reference*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/specs/man/java.html. [Accessed: 24-Sep-2026].

[20] Oracle, "Interface Thread.Builder.OfPlatform: stackSize," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/Thread.Builder.OfPlatform.html. [Accessed: 24-Sep-2026].

[21] Python Software Foundation, "sys — System-specific parameters and functions: setrecursionlimit," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/sys.html. [Accessed: 24-Sep-2026].

[22] Python Software Foundation, "What's new in Python 3.11: inlined Python function calls." [Online]. Available: https://docs.python.org/3/whatsnew/3.11.html. [Accessed: 24-Sep-2026].

[23] Python Software Foundation, "What's new in Python 3.12: sys." [Online]. Available: https://docs.python.org/3/whatsnew/3.12.html. [Accessed: 24-Sep-2026].

[24] Profitina, “Profitina — AI Visibility (GEO) platform.” [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

[25] Z. Manna and J. McCarthy, “Properties of programs and partial function logic,” in *Machine Intelligence 5*, B. Meltzer and D. Michie, Eds. Edinburgh, U.K.: Edinburgh University Press, 1970, pp. 27–37.

Bab 11
Tree dan Binary Search Tree

Di akhir bab ini Anda akan sanggup menjelaskan apa itu *tree* (pohon) dan memakai kosakatanya dengan tepat (akar, induk, anak, saudara, daun, simpul internal, leluhur, keturunan, lintasan, tingkat, kedalaman, tinggi, derajat, subpohon); menggambar dan menuliskan sebuah pohon dalam empat notasi; menyimpan pohon dalam larik dan dalam simpul-simpul bertaut serta menyebutkan kapan masing-masing cocok; menggolongkan pohon biner sebagai *full*, *perfect*, *complete*, atau condong, serta membatasi banyak simpul dan tingginya; menelusuri pohon biner secara *preorder*, *inorder*, *postorder*, dan per tingkat (*level order*), secara rekursif maupun dengan *stack* atau *queue* eksplisit, dan memakai tiga urutan telusur-dalam untuk mengubah pohon ekspresi menjadi notasi prefiks, infiks, dan postfiks; menyatakan sifat *binary search tree* (pohon pencarian biner) serta mengimplementasikan, menelusuri dengan tangan, dan menganalisis penyisipan (pada contoh berjalan `insert(65, 5, 70, 3, 10)`), pencarian, minimum, maksimum, *successor*, *predecessor*, penghapusan dalam ketiga kasusnya dengan varian *successor* maupun varian *predecessor*, penghitungan simpul dan daun, tinggi, serta pencetakan pohon secara menyamping; mengukur bagaimana urutan penyisipan menentukan apakah sebuah pohon seimbang atau merosot; mengenali rotasi yang dipakai pohon penyeimbang-diri (AVL dan *red-black*) untuk menjaga setiap operasi tetap logaritmik; mengubah pohon umum menjadi pohon biner dan membangun kembali pohon dari daftar *preorder* beserta derajatnya; serta memilih di antara `std::map`, `TreeMap` di Java, dan alternatifnya di Python. Setiap operasi digambar keadaan demi keadaan, setiap ongkos diturunkan, dan setiap program lengkap, dikompilasi atau dijalankan dalam C++, Java, dan Python, serta diperiksa memorinya.

11.1 Dari Mana Kita Melanjutkan

Setiap struktur dalam buku ini sejauh ini berbentuk *garis*. Larik pada Bab 1 menyimpan elemennya dalam satu baris; *stack* dan *queue* pada Bab 5 menambah dan membuang di ujung-ujung sebuah baris; *linked list* pada Bab 7 dan Bab 9 adalah baris yang disambung dengan pointer. Dalam sebuah garis, setiap elemen memiliki paling banyak satu elemen sesudahnya, dan seluruh struktur dapat dijalani dari ujung ke ujung dalam satu lintasan.

Garis memaksa sebuah pilihan sulit, yang telah diperlihatkan Bab 2 dan Bab 7. *Larik terurut* dapat dicari dengan *binary search* dalam $O(\lg n)$ langkah (Subbab 2.5), tetapi menyisipkan elemen baru pada tempatnya menggeser, rata-rata, separuh larik. *Linked list* menyisipkan dalam $O(1)$ begitu ia memegang posisi yang tepat (Subbab 7.6.3), tetapi ia hanya dapat menemukan posisi itu dengan berjalan, satu simpul demi satu simpul, dalam $\Theta(n)$ langkah. Kita menginginkan keduanya sekaligus: struktur yang semudah larik terurut untuk dicari dan semudah *linked list* untuk diubah.

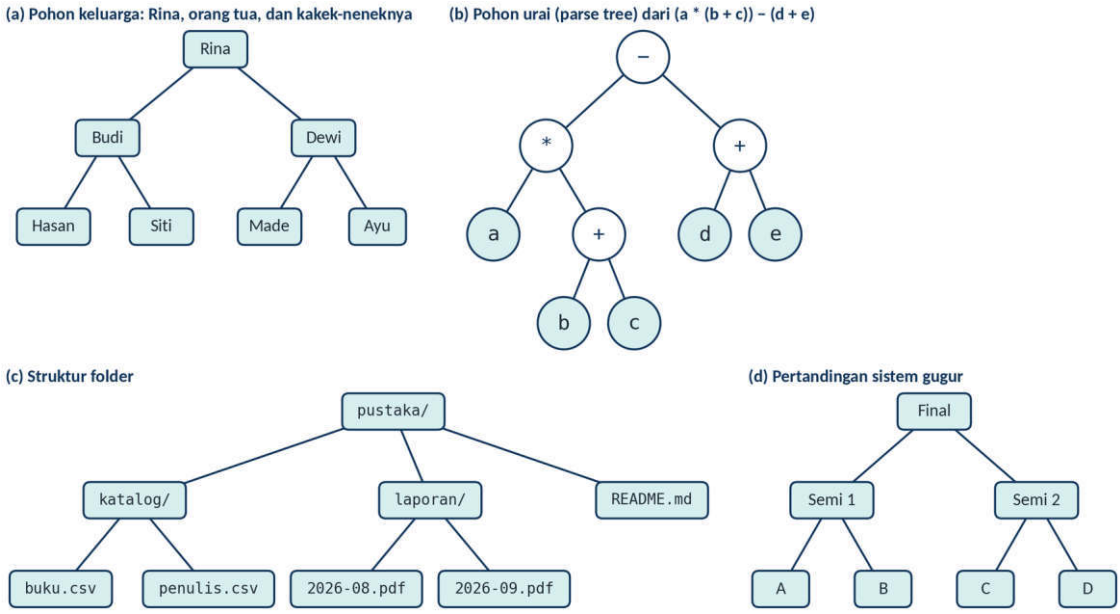
Bab ini sampai ke sana dengan meninggalkan garis. Sebuah *tree* (pohon) membolehkan setiap elemen menunjuk *beberapa* elemen di bawahnya. Pointer dari Bab 6 dan simpul dari Bab 7 tetap menjadi bahan bangunannya; yang berubah adalah bentuknya. Bentuk itu secara alami rekursif, dan karena itulah Bab 10 datang lebih dahulu: sebuah pohon adalah akar dengan pohon-pohon yang lebih kecil tergantung di bawahnya, dan hampir setiap algoritme dalam bab ini hanyalah tiga atau empat baris rekursi yang mengikuti definisi itu.

Paruh pertama bab ini membahas pohon secara umum dan pohon biner secara khusus: kosakatanya, gambarnya, representasinya, dan penelusurannya. Paruh kedua membahas *binary search tree* (BST), yang menyimpan kuncinya dalam urutan sedemikian rupa sehingga setiap pencarian dapat membuang satu subpohon utuh pada setiap langkah. Kita membangunnya dengan contoh berjalan `insert(65, 5, 70, 3, 10)`, mencari di dalamnya, menghitungnya, mencetaknya, menghapus darinya dalam ketiga kasus, lalu mengukur apa yang terjadi bila kunci yang sama datang dalam urutan yang buruk. Bab 12 mengerjakan soal-soal persis dari jenis ini, pada pohon yang dibangun dengan cara yang sama.

11.2 Apa Itu Pohon

11.2.1 Empat pohon dari kehidupan sehari-hari

Sebelum definisi apa pun, perhatikan keempat gambar pada **Gambar 11.1**. Asalnya sangat berbeda, tetapi bentuknya sama.



Gambar 11.1 Empat pohon: (a) pohon keluarga yang memuat para leluhur, (b) pohon urai yang dibangun kompilator untuk sebuah ekspresi aritmetika, (c) folder dan berkas pada sebuah diska, (d) bagan pertandingan sistem gugur.

- **Pohon keluarga (a).** Rina digambar di puncak. Di bawahnya ada kedua orang tuanya, dan di bawah setiap orang tua ada kedua orang tua dari orang tua itu. Setiap orang dalam gambar kecuali Rina dicapai dari Rina melalui tepat satu rantai garis.
- **Pohon urai (b).** Kompilator yang membaca `(a * (b + c)) - (d + e)` tidak menyimpannya sebagai deretan karakter. Ia membangun pohon ini: operator yang diterapkan *terakhir*, yaitu pengurangan, berada di puncak; kedua operannya adalah dua pohon yang lebih kecil di bawahnya; dan huruf-huruf, yang bukan operator, berada di dasar. Mengevaluasi ekspresi itu berarti mengevaluasi kedua pohon bawah, lalu menerapkan operator di puncak. Subbab 11.6.6 membangun pohon seperti ini dan membacanya kembali dalam tiga notasi.
- **Struktur folder (c).** Folder `pustaka/` memuat dua folder dan sebuah berkas; setiap folder memuat berkasnya sendiri. Sistem operasi menemukan `buku.csv` dengan berjalan dari folder teratas turun melalui lintasan `pustaka/katalog/buku.csv`. Subbab 10.14 sudah menjumpai bentuk yang sama pada menu layanan Profitina Laundry.

- **Pertandingan sistem gugur (d).** Empat tim bertanding dalam dua semifinal; kedua pemenangnya bertemu dalam laga penentuan. Dibaca dari bawah ke atas, setiap pertemuan dua garis pada sebuah kotak adalah satu pertandingan.

Setiap gambar memiliki satu elemen di puncak, setiap elemen lain tergantung di bawah tepat satu elemen, dan tidak ada yang tergantung melingkar. Hanya itulah sebuah pohon.

11.2.2 Definisinya

Pohon adalah struktur data *nonlinear*: ia melukiskan hubungan *hierarkis*, satu-ke-banyak, di antara elemen-elemennya. Setiap elemen boleh memiliki banyak elemen tepat di bawahnya, tetapi hanya satu tepat di atasnya. Gambar-gambarnya selalu meletakkan elemen istimewa yang tidak memiliki apa pun di atasnya di bagian paling atas, sehingga ilmuwan komputer menggambar pohon terbalik, dengan akar di langit dan daun di tanah.

> **DEFINISI.** Sebuah *pohon* adalah himpunan berhingga T yang terdiri atas satu *simpul* atau lebih sedemikian rupa sehingga (1) ada satu simpul yang ditetapkan secara khusus, yang disebut *akar* (*root*) dari T, dan (2) simpul-simpul selebihnya terbagi ke dalam $m \geq 0$ himpunan saling lepas $T_1, T_2, \ldots, T_m$, yang masing-masing juga sebuah pohon. $T_1, \ldots, T_m$ adalah *subpohon* dari akar. Garis yang ditarik dari akar ke akar setiap subpohon disebut *sisi* (*edge*).

Definisi ini, yang mengikuti Knuth [1], bersifat *rekursif*: ia mendefinisikan pohon dengan memakai pohon yang lebih kecil. Definisi ini tidak berputar-putar, dengan alasan yang sama seperti definisi rekursif pada Subbab 10.2.2: setiap subpohon memiliki lebih sedikit simpul daripada pohon yang memuatnya, sehingga penguraiannya berhenti pada pohon bersimpul satu, yang memiliki $m = 0$ subpohon. Rekursi dalam definisi inilah yang membuat rekursi menjadi cara alami untuk memprogram pohon.

Dua fakta segera menyusul. Pertama, dari akar ke setiap simpul ada tepat satu lintasan, karena setiap simpul termasuk tepat satu subpohon dari akar, dan di dalam subpohon itu hal yang sama berlaku lagi. Kedua, pohon dengan n simpul memiliki tepat

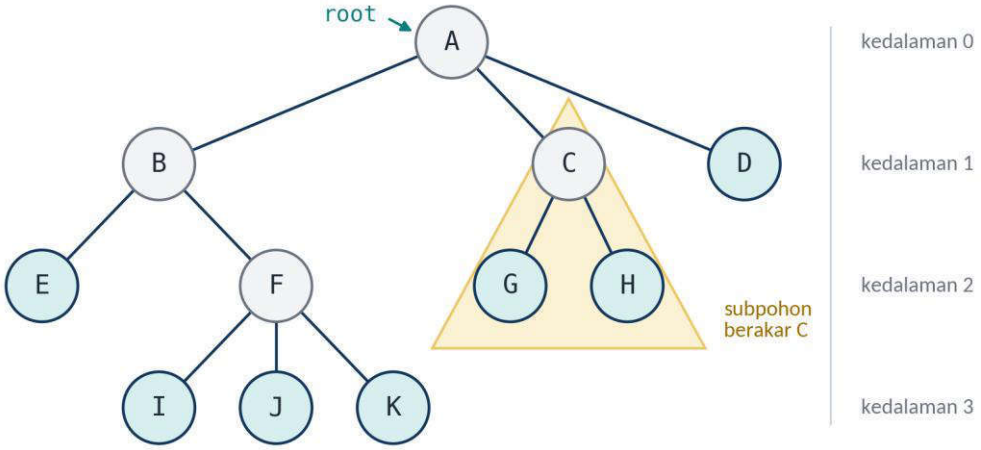
$$e = n - 1 \tag{11.1}$$

sisi. Buktinya dengan induksi pada n, metode Subbab 10.2.3. Pohon bersimpul satu tidak memiliki sisi, dan $1 - 1 = 0$. Untuk pohon yang lebih besar, andaikan klaim itu berlaku bagi semua pohon yang lebih kecil; subpohon-subpohon akar memiliki $n_1, \ldots, n_m$ simpul dengan $n_1 + \cdots + n_m = n - 1$, sehingga menurut hipotesis induksi mereka memiliki $(n_1 - 1) + \cdots + (n_m - 1) = n - 1 - m$ sisi, dan akar menambahkan satu sisi ke setiap subpohonnya yang berjumlah m, sehingga totalnya $n - 1 - m + m = n - 1$. Persamaan (11.1) patut diingat, karena ia memberi pemeriksaan cepat untuk setiap pohon yang Anda gambar: hitung simpulnya, hitung garisnya, dan banyak garis harus satu lebih sedikit. Cara lain untuk melihatnya: setiap simpul kecuali akar memiliki tepat satu sisi yang naik ke induknya, sehingga banyak sisi sama dengan banyak simpul bukan-akar.

Kumpulan nol pohon atau lebih yang saling lepas disebut *hutan* (*forest*). Membuang akar sebuah pohon meninggalkan sebuah hutan: subpohon-subpohonnya.

11.2.3 Kosakatanya

Pohon datang dengan kosakata yang luas, sebagian besar dipinjam dari pohon keluarga. **Gambar 11.2** memperlihatkan pohon tempat kita mendefinisikannya, dan **Tabel 11.1** mendaftar setiap istilah beserta nilainya pada pohon itu.



Gambar 11.2 Sebuah pohon dengan sebelas simpul. Simpul internal berwarna abu-abu, daun berwarna hijau-biru; segitiga kuning menandai subpohon berakar C, dan kedalaman setiap tingkat tertera di kanan.

- **Akar** (*root*) adalah satu-satunya simpul yang tidak memiliki simpul di atasnya: A.
- Bila sebuah sisi menghubungkan simpul u ke simpul v tepat di bawahnya, u adalah **induk** (*parent*) dari v dan v adalah **anak** (*child*) dari u. Setiap simpul kecuali akar memiliki tepat satu induk. Anak-anak dari induk yang sama adalah **saudara** (*siblings*).
- Simpul tanpa anak disebut **daun** (*leaf*) atau *simpul eksternal*. Simpul yang memiliki paling sedikit satu anak disebut **simpul internal**.
- Sebuah **lintasan** (*path*) adalah barisan simpul $v_0, v_1, \ldots, v_k$ yang setiap v_{i+1}-nya adalah anak dari v_i; **panjangnya** adalah k, banyak sisi di sepanjang lintasan itu.
- **Leluhur** (*ancestors*) sebuah simpul adalah simpul-simpul pada lintasan dari akar ke simpul itu, selain simpul itu sendiri (induknya, induk dari induknya, dan seterusnya sampai akar). **Keturunan** (*descendants*) sebuah simpul adalah simpul-simpul yang menjadikannya leluhur (anak-anaknya, cucu-cucunya, dan seterusnya).
- **Subpohon berakar** v adalah v bersama semua keturunannya. Setiap simpul dapat dipandang sebagai akar subpohon di bawahnya; pada **Gambar 11.2** subpohon berakar C memuat C, G, dan H.
- **Kedalaman** (*depth*) sebuah simpul adalah panjang lintasan dari akar ke simpul itu, yang juga sama dengan banyak leluhurnya. Akar berkedalaman 0. Himpunan simpul dengan kedalaman sama membentuk sebuah **tingkat** (*level*); dalam buku ini akar berada di tingkat 0, sehingga tingkat sebuah simpul sama dengan kedalamannya.
- **Tinggi** (*height*) sebuah simpul adalah panjang lintasan *terpanjang* dari simpul itu turun ke sebuah daun. Daun bertinggi 0. **Tinggi pohon** adalah tinggi akarnya, yang sama dengan kedalaman terbesar di antara simpul-simpulnya. Menurut kesepakatan, pohon kosong bertinggi -1, yang membuat rumus rekursif pada Subbab 11.7.5 berlaku tanpa kasus khusus.
- **Derajat** (*degree*) sebuah simpul adalah banyak anaknya. **Derajat pohon** adalah derajat terbesar di antara simpul-simpulnya.
- **Ukuran** (*size*) pohon adalah banyak simpulnya.

Tabel 11.1 Kosakata pohon, dievaluasi pada pohon **Gambar 11.2**.

Istilah	Nilai pada
Akar	A
Simpul internal	A, B, C, F
Daun	E, I, J, K, G, H, D
Anak-anak B	E, F
Saudara G	H
Leluhur J	F, B, A
Keturunan C	G, H
Kedalaman F; tinggi F	2; 1
Tinggi B; tinggi pohon	2; 3
Derajat A; derajat F; derajat pohon	3; 3; 3
Ukuran; banyak sisi	11; 10, sesuai tuntutan Persamaan (11.1)

KEDALAMAN BUKAN TINGGI. Kedalaman diukur *turun dari akar* ke sebuah simpul: "seberapa jauh saya di bawah puncak?" Tinggi diukur *naik dari daun terendah* di bawah sebuah simpul: "setinggi apa pohon yang tergantung dari saya?" Pada **Gambar 11.2**, F berkedalaman 2 dan bertinggi 1, sedangkan B berkedalaman 1 dan bertinggi 2. Hanya untuk pohon secara keseluruhan keduanya bertemu: tinggi pohon adalah kedalaman terbesar di dalamnya. Sebagian buku mulai menghitung tingkat dari 1, bukan 0; ketika membaca teks lain, periksalah kesepakatannya sebelum memakai rumus-rumusnya.

11.2.4 Contoh kerja pada pohon kecil

Agar kosakata ini menjadi otomatis, kerjakan pohon bersimpul tujuh yang akarnya A memiliki anak B dan C, dengan B beranak D dan E, serta C beranak F dan G (**Gambar 11.7** (b) menggambarnya).

- Leluhur F adalah induknya, C, dan induk C, yaitu A, sehingga leluhur(F) $=\{C, A\}$ dan F berkedalaman 2.
- Keturunan C adalah anak-anaknya, F dan G, yang tidak memiliki anak: keturunan(C) $=\{F, G\}$.
- Induk D adalah B. Anak-anak A adalah B dan C. Satu-satunya saudara F adalah G.
- Daunnya adalah D, E, F, dan G; simpul internalnya A, B, dan C.
- Setiap simpul internal berderajat 2, sehingga pohon itu berderajat 2; tingginya 2, dan ukurannya 7, dengan $7 - 1 = 6$ sisi.

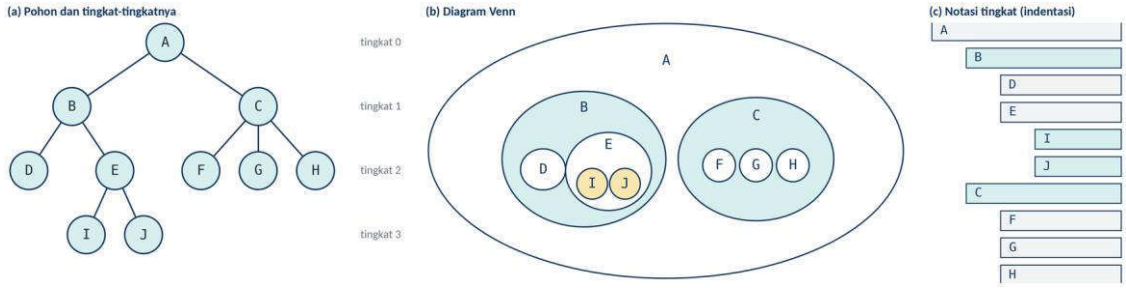
11.2.5 Pohon statis dan pohon dinamis

Dua jenis pemakaian muncul berulang kali. Pada **pohon statis** bentuknya ditetapkan sekali dan isi simpul-simpulnya tidak berubah sesudahnya: pohon urai sebuah ekspresi, bagan sebuah turnamen, daftar isi sebuah buku cetak. Pada **pohon dinamis** simpul ditambahkan (*disisipkan*) dan dibuang (*dihapus*) selama program berjalan, dan bentuknya ikut berubah: katalog perpustakaan yang bertambah dan berkurang bukunya, indeks yang tumbuh bersama setiap rekaman baru. Pohon statis dapat disimpan secara ringkas, sering kali dalam larik (Subbab 11.4.1). Pohon dinamis hampir selalu dibangun dari simpul bertaut (Subbab 11.4.2), dengan alasan yang sama seperti Bab 7 membangun list dinamis dari simpul: tidak ada yang perlu dipesan di muka, dan sebuah perubahan hanya menyentuh beberapa pointer.

Akar istimewa pada kedua jenis itu. Ia adalah simpul yang darinya setiap simpul lain dapat dilacak, dan pada pohon dinamis ia adalah simpul yang dibuat pertama kali: penyisipan pertama ke pohon kosong menciptakan akar, dan setiap simpul sesudahnya dipasang di suatu tempat di bawahnya.

11.3 Menggambar dan Menuliskan Pohon

Gambar dari atas ke bawah bukan satu-satunya cara memperlihatkan pohon. **Gambar 11.3** memperlihatkan pohon bersimpul sepuluh yang sama dalam tiga gambar, dan notasi keempat menuliskannya dalam satu baris.



Gambar 11.3 Satu pohon dalam tiga notasi: (a) gambar biasa beserta tingkat-tingkatnya, (b) diagram Venn, yang menggambar setiap subpohon sebagai himpunan di dalam himpunan induknya, (c) notasi tingkat, yang menggambar setiap simpul sebagai batang yang menjorok sesuai kedalamannya.

- **Gambar dengan tingkat (a)** adalah gambar yang dipakai di sepanjang buku ini. Tinggi pohon ini 3, dan setiap tingkat diberi label di sebelah kanan.
- **Diagram Venn (b)** menggambar setiap subpohon sebagai daerah tertutup, bersarang di dalam daerah induknya. Daerah A memuat segalanya; di dalamnya, B dan C adalah daerah yang saling lepas, karena subpohon keduanya tidak berbagi simpul; di dalam B ada D dan daerah E, yang memuat I dan J. Diagram Venn memperlihatkan sekilas bahwa subpohon-subpohon sebuah simpul tidak pernah tumpang-tindih, yang merupakan bagian (2) dari definisi.
- **Notasi tingkat (c)** mendaftar simpul satu per baris dalam urutan *preorder* (urutan Subbab 11.6.1) dan menjorokkan setiap baris sesuai kedalamannya, sehingga anak-anak sebuah simpul adalah baris-baris tepat di bawahnya yang menjorok satu langkah lebih jauh. Inilah notasi penjelajah berkas, tampilan kerangka pada pengolah kata, dan Subbab 11.7.6, yang mencetak pohon biner secara menyamping dengan semangat yang sama.
- **Notasi kurung** menuliskan sebuah simpul, lalu, di dalam kurung dan dipisahkan koma, notasi setiap subpohonnya. Untuk pohon pada **Gambar 11.3**:

```
(A(B(D,E(I,J)),C(F,G,H)))
```

Pasangan kurung terluar melingkupi seluruh pohon; setiap daun cukup ditulis namanya. Membaca dari kiri ke kanan mengunjungi simpul-simpul dalam urutan *preorder*. Notasi kurung adalah cara program LISP, dan dokumen JSON di web, menuliskan pohon sebagai teks: setiap pasangan kurung bersarang adalah sebuah subpohon. Soal 11.2 meminta Anda menyusun keempat notasi untuk pohon lain, dan program pada Subbab 11.10.3 mencetak pohon dalam notasi ini.

11.4 Merepresentasikan Pohon dalam Memori

Sebuah pohon di dalam memori memerlukan dua hal untuk setiap simpul: datanya, dan cara mencapai anak-anaknya. Ada dua jawaban klasik, yang mencerminkan dua representasi list: representasi *statis* dalam larik dan representasi *dinamis* dengan pointer.

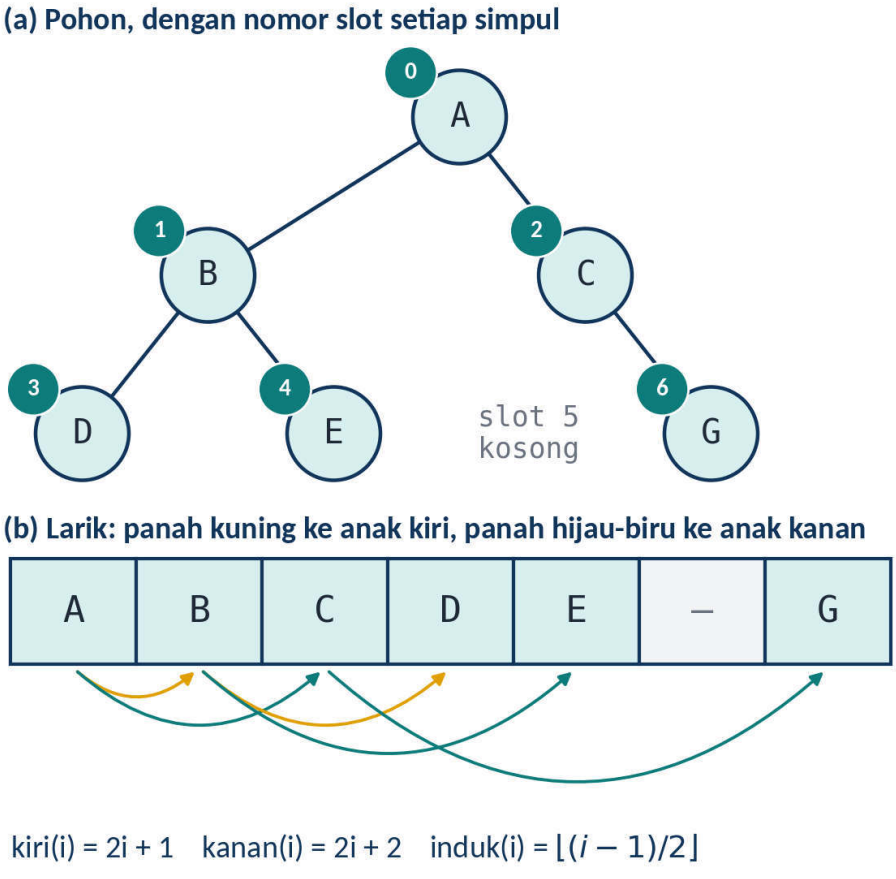
11.4.1 Representasi statis: larik

Untuk pohon biner (pohon yang simpulnya memiliki paling banyak dua anak, Subbab 11.5) ada cara rapi yang sama sekali tidak memakai pointer. Nomori posisi-posisi pohon tingkat demi tingkat, dari kiri ke kanan, mulai dari 0 untuk akar, lalu simpan simpul pada posisi i di slot i sebuah larik. Anak-anak dan induk dari simpul di slot i kemudian ditemukan dengan aritmetika:

$$left(i) = 2i + 1, \qquad right(i) = 2i + 2, \qquad parent(i) = \left\lfloor \frac{i-1}{2} \right\rfloor \tag{11.2}$$

Mengapa ini berhasil? Tingkat ℓ memuat paling banyak 2^ℓ posisi (Subbab 11.5.3), sehingga tingkat $0,1,\ldots,\ell-1$ menempati slot 0 sampai $2^\ell - 2$ dan tingkat ℓ mulai di slot $2^\ell - 1$. Simpul di slot i pada tingkat ℓ adalah simpul ke-$(i - 2^\ell + 1)$ pada tingkatnya, dihitung dari 0; anak-anaknya adalah simpul ke-$2(i - 2^\ell + 1)$ dan simpul sesudahnya pada tingkat $\ell + 1$, yang dimulai di slot $2^{\ell+1} - 1$. Dijumlahkan, anak kiri berada di slot $2^{\ell+1} - 1 + 2(i - 2^\ell + 1) = 2i + 1$, dan anak kanan satu slot sesudahnya. Rumus induk hanyalah kebalikan keduanya.

Gambar 11.4 memperlihatkan pohon bersimpul enam dalam larik bertujuh slot. Simpul C memiliki anak kanan G tetapi tidak memiliki anak kiri, sehingga slot $2 \cdot 2 + 1 = 5$ tetap kosong, ditandai `-`.



Gambar 11.4 Representasi statis: (a) setiap simpul dinomori tingkat demi tingkat, (b) simpul i disimpan di slot i, dan anak-anaknya ditemukan di slot $2i + 1$ dan $2i + 2$ menurut Persamaan (11.2).

Kode 11.1 menyimpan pohon ini dalam sebuah string, satu karakter per slot, mencetak setiap simpul beserta induk dan anak-anaknya yang dihitung dengan Persamaan (11.2), dan menelusuri pohon secara *preorder* dan *inorder* dengan mengikuti aritmetika indeks, bukan pointer.

Kode 11.1 `array_tree.cpp`: pohon biner yang disimpan dalam larik; anak dan induk ditemukan dengan aritmetika indeks.

```cpp
// array_tree.cpp -- a binary tree stored in an array (the static representation).
#include <cstdio>
#include <string>

// Slot i holds a node; its children live in slots 2i+1 and 2i+2, its parent in
// slot (i-1)/2. A '-' marks an empty slot (no node there).
const std::string TREE = "ABCDE-G";

bool exists(const std::string &t, int i) {
    return i >= 0 && i < (int) t.size() && t[i] != '-';
}

void preOrder(const std::string &t, int i) {       // root, left, right
    if (!exists(t, i)) return;
    std::printf("%c ", t[i]);
    preOrder(t, 2 * i + 1);
    preOrder(t, 2 * i + 2);
}

void inOrder(const std::string &t, int i) {        // left, root, right
    if (!exists(t, i)) return;
    inOrder(t, 2 * i + 1);
    std::printf("%c ", t[i]);
    inOrder(t, 2 * i + 2);
}

int main() {
    std::printf("slot  node  parent  left  right\n");
    for (int i = 0; i < (int) TREE.size(); ++i) {
        if (!exists(TREE, i)) {
            std::printf("%4d     -   (empty slot)\n", i);
            continue;
        }
        int p = (i - 1) / 2, l = 2 * i + 1, r = 2 * i + 2;
        std::printf("%4d  %4c  %6c  %4c  %5c\n", i, TREE[i],
                    i == 0 ? '-' : TREE[p],
                    exists(TREE, l) ? TREE[l] : '-',
                    exists(TREE, r) ? TREE[r] : '-');
    }
    std::printf("preorder: ");
    preOrder(TREE, 0);
    std::printf("\ninorder:  ");
    inOrder(TREE, 0);
    std::printf("\n");
```

Kode 11.1 (Lanjutan) *array_tree.cpp: pohon biner yang disimpan dalam larik; anak dan induk ditemukan dengan aritmetika indeks.*

```cpp
    // A right-skewed chain of h+1 nodes needs 2^(h+1)-1 slots.
    for (int h = 1; h <= 4; ++h) {
        int slots = (1 << (h + 1)) - 1;
        std::printf("right chain of %d nodes needs %2d slots (%d wasted)\n",
                    h + 1, slots, slots - (h + 1));
    }
    return 0;
}
```

```
slot  node  parent  left  right
   0     A       -     B      C
   1     B       A     D      E
   2     C       A     -      G
   3     D       B     -      -
   4     E       B     -      -
   5     -   (empty slot)
   6     G       C     -      -
preorder: A B D E C G
inorder:  D B E A C G
right chain of 2 nodes needs  3 slots (1 wasted)
right chain of 3 nodes needs  7 slots (4 wasted)
right chain of 4 nodes needs 15 slots (11 wasted)
right chain of 5 nodes needs 31 slots (26 wasted)
```

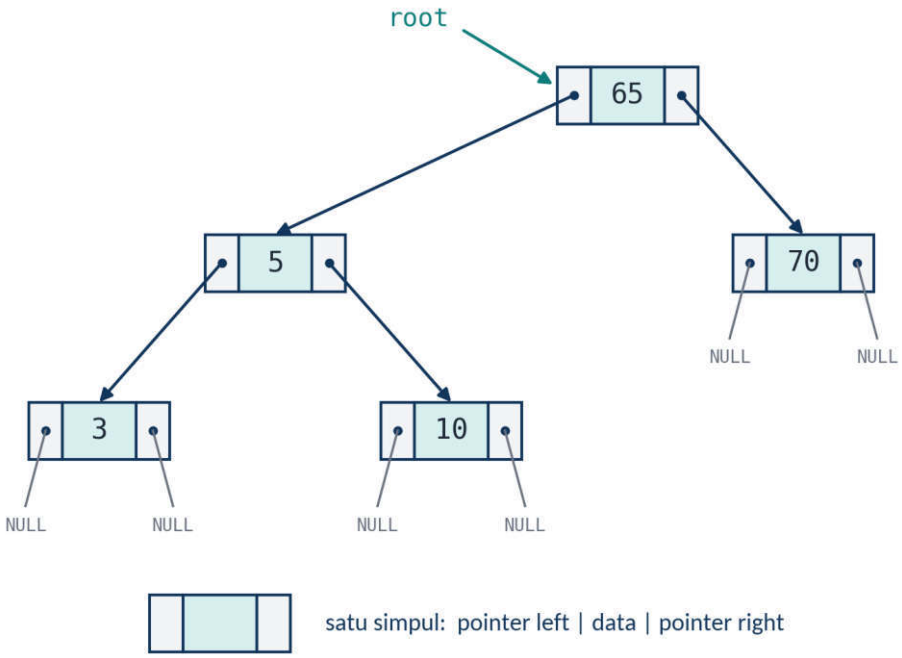
Representasi larik tidak menghabiskan memori untuk pointer dan menemukan induk dalam $O(1)$, sesuatu yang tidak sanggup dilakukan representasi bertaut pada subbab berikutnya. Kelemahannya adalah slot kosong. Pohon yang setiap tingkatnya penuh kecuali mungkin tingkat terakhir, yang diisi dari kiri, dan yang disebut *complete* oleh Subbab 11.5.2, sama sekali tidak memiliki slot kosong, dan itulah sebabnya Bab 14 menyimpan *heap*-nya dengan cara ini. Pohon yang berlubang membuang satu slot untuk setiap posisi yang hilang, dan baris-baris terakhir keluaran memperlihatkan betapa parahnya: rantai $h + 1$ simpul yang selalu belok ke kanan memerlukan $2^{h+1} - 1$ slot, sehingga rantai 21 simpul saja akan memerlukan lebih dari dua juta slot. Larik cocok untuk pohon statis dengan bentuk penuh yang sudah diketahui; pohon yang bentuknya berubah pada setiap penyisipan memerlukan pointer.

11.4.2 Representasi dinamis: simpul bertaut

Representasi dinamis memberi setiap simpul dua pointer, satu ke anak kiri dan satu ke anak kanan, dengan pointer nol di tempat anak yang tidak ada. Inilah simpul bertaut ganda dari Bab 9 dengan dua pointer yang diberi makna baru: alih-alih `prev` dan `next`, yang menunjuk sepanjang sebuah garis, kita memiliki `left` dan `right`, yang menunjuk *ke bawah* ke dua subpohon yang berbeda. Deklarasi simpulnya, dalam gaya C klasik dari Subbab 7.4.1, adalah:

```
typedef struct MyNode {
    int data;
    struct MyNode *left;
    struct MyNode *right;
} Node;
```

dan seluruh pohon dicapai melalui satu pointer ke akar, `Node *root`, yang bernilai `nullptr` bila pohon kosong. **Gambar 11.5** menggambar pohon bersimpul lima yang akan dibangun Subbab 11.7.2.



Gambar 11.5 Representasi dinamis pohon biner: setiap simpul memiliki pointer kiri, medan data, dan pointer kanan; `root` menunjuk simpul teratas, dan setiap anak yang tidak ada adalah pointer NULL.

Hitunglah pointer NULL pada gambar itu: ada enam, untuk lima simpul. Itu bukan kebetulan. Setiap simpul dari n simpul memiliki dua medan pointer, $2n$ seluruhnya, dan menurut Persamaan (11.1) tepat $n-1$ di antaranya memuat sebuah sisi, sehingga banyak pointer nol dalam pohon biner dengan n simpul adalah

$$2n - (n-1) = n + 1 \tag{11.3}$$

Lebih dari separuh medan pointer setiap pohon biner bernilai nol. Itulah harga bentuk yang luwes, dan biasanya harga itu pantas dibayar. (Sebagian implementasi memakai ulang medan-medan nol itu sebagai "benang" yang menunjuk *successor* dalam urutan *inorder*, sehingga penelusuran dapat dilakukan tanpa *stack*; Knuth [1] menguraikan *threaded tree* ini.)

11.4.3 Pohon dengan banyak anak sembarang

Bila sebuah simpul boleh memiliki anak berapa pun banyaknya, seperti pohon keluarga, folder, dan pertandingan pada **Gambar 11.1**, tidak satu pun representasi di atas cocok secara langsung. Tiga jawaban yang lazim adalah:

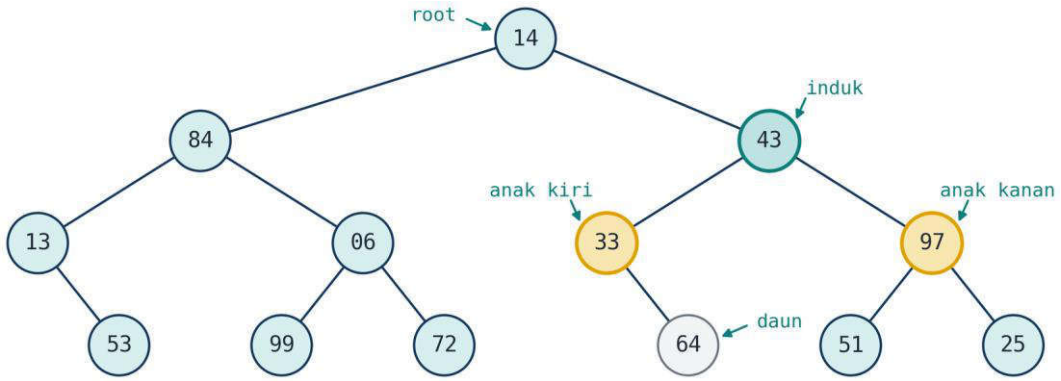
- **Larik induk.** Simpan, untuk setiap simpul, indeks induknya: `parent[i]`. Cara ini ringkas dan mudah menemukan lintasan ke akar, persis yang dilakukan sistem berkas untuk menyusun nama lintasan lengkap, tetapi menemukan anak-anak sebuah simpul memerlukan pemindaian seluruh larik.
- **Daftar anak.** Beri setiap simpul sebuah larik dinamis berisi pointer ke anak-anaknya (`std::vector<Node*>` di C++, `List` di Java, `list` di Python). Beginilah pohon DOM sebuah halaman web dan pohon menu pada Subbab 10.14 biasanya disimpan.
- **Anak pertama, saudara berikutnya.** Beri setiap simpul tepat dua pointer: satu ke *anak pertamanya* dan satu ke *saudara berikutnya*. Setiap simpul dari pohon apa pun lalu memiliki dua pointer, persis seperti simpul pohon biner, dan Subbab 11.10 memperlihatkan bahwa ini sesungguhnya cara mengubah pohon apa pun menjadi pohon biner.

11.5 Pohon Biner

11.5.1 Definisi

> **DEFINISI.** Sebuah *pohon biner* (*binary tree*) adalah pohon yang *kosong* (tidak memiliki simpul), atau terdiri atas sebuah simpul akar bersama dua pohon biner, yang disebut *subpohon kiri* dan *subpohon kanan*, yang tidak berbagi simpul.

Dua perbedaan dari pohon umum pada Subbab 11.2.2 penting. Pohon biner boleh kosong. Dan kedua subpohonnya *dibedakan*: simpul yang hanya memiliki anak kiri adalah pohon biner yang berbeda dari simpul yang hanya memiliki anak kanan, walaupun sebagai pohon umum keduanya sama. Jadi setiap simpul pohon biner memiliki paling banyak dua anak, masing-masing berlabel kiri atau kanan. **Gambar 11.6** memberi label pada bagian-bagiannya.

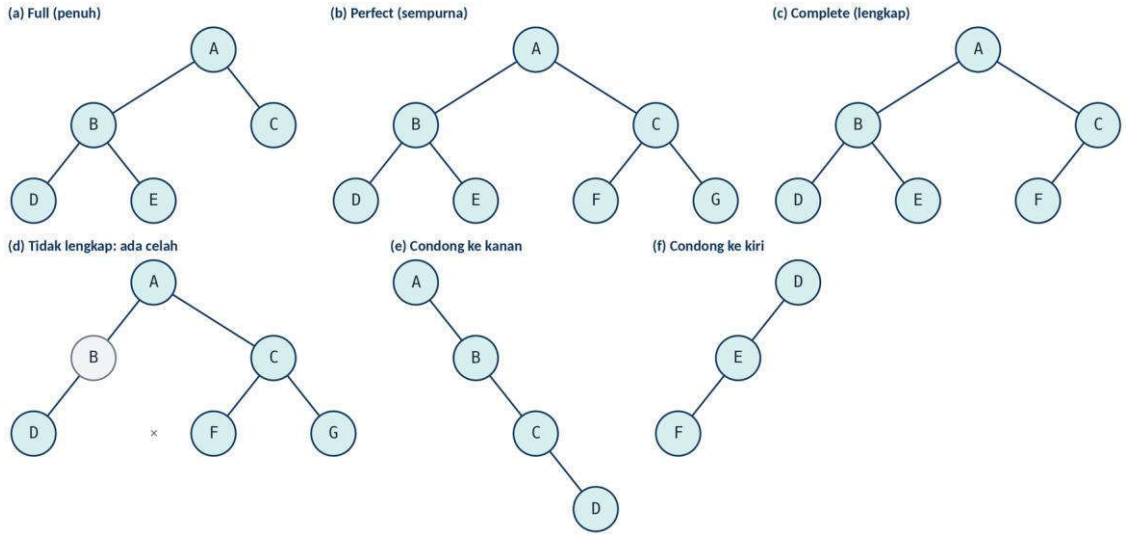


Gambar 11.6 Pohon biner dengan empat belas simpul: simpul 43 adalah induk dari anak kirinya 33 dan anak kanannya 97, dan 64, yang tidak memiliki anak, adalah daun.

Pohon biner jauh lebih penting daripada yang tersirat dari batasannya. Dua anak per simpul adalah persis yang diperlukan sebuah keputusan ya/tidak, sehingga setiap pencarian berbasis perbandingan dan setiap ekspresi dengan operator biner adalah pohon biner. Dan Subbab 11.10 memperlihatkan bahwa pohon apa pun dapat disimpan sebagai pohon biner, sehingga tidak ada yang hilang dengan mempelajari pohon biner lebih dahulu.

11.5.2 Full, perfect, complete, dan condong

Pohon biner digolongkan menurut bentuknya, dan bentuk itulah yang menentukan secepat apa operasi pada Subbab 11.7 berjalan. **Gambar 11.7** memperlihatkan keempat golongan dan satu contoh penyangkal.



Gambar 11.7 Bentuk-bentuk pohon biner: (a) *full*, (b) *perfect*, (c) *complete*, (d) tidak *complete*, karena posisi 4 pada tingkat terakhir kosong sementara posisi 5 terisi, (e) condong ke kanan, (f) condong ke kiri.

DEFINISI. Sebuah pohon biner disebut **full** (penuh; juga disebut *proper* atau *strictly binary*) bila setiap simpulnya memiliki 0 atau 2 anak. Ia disebut **perfect** (sempurna) bila ia *full* dan semua daunnya berkedalaman sama. Ia disebut **complete** (lengkap) bila setiap tingkat kecuali mungkin tingkat terakhir terisi penuh dan simpul-simpul tingkat terakhir berada sejauh mungkin di kiri. Ia disebut **condong** (*skewed* atau *degenerate*) bila setiap simpul internalnya memiliki tepat satu anak; pohon condong yang semua anaknya anak kanan disebut *condong ke kanan*, dan yang semua anaknya anak kiri disebut *condong ke kiri*.

Golongan-golongan ini beririsan dengan cara yang pasti: setiap pohon *perfect* adalah *full* sekaligus *complete*; pohon *complete* tidak harus *full* (**Gambar 11.7** (c): C hanya memiliki satu anak); pohon *full* tidak harus *complete* (**Gambar 11.7** (a) adalah *full* dan kebetulan juga *complete*; pindahkan D dan E ke bawah C maka ia tetap *full* tetapi tidak lagi *complete*). Pohon condong dengan n simpul adalah sebuah rantai: tingginya $n - 1$, yang terbesar yang mungkin, dan sesungguhnya ia sebuah *linked list* yang digambar miring.

NAMA-NAMANYA TIDAK SERAGAM. Buku teks yang lebih lama, dan sebagian catatan perkuliahan di Indonesia, memakai *full binary tree* untuk yang dalam buku ini disebut *perfect* ("semua simpul kecuali daun memiliki dua anak dan setiap lintasan sama panjang") dan *complete binary tree* untuk yang dalam buku ini disebut *full* ("setiap simpul internal memiliki dua anak, panjang lintasan boleh berbeda"). Buku ini mengikuti Cormen dkk. [2] dan pemakaian baku masa kini, yang diringkas pada **Tabel 11.2**. Ketika membaca definisi di tempat lain, periksalah mana yang dimaksud.

Tabel 11.2 Empat bentuk pohon biner, beserta nama-nama lain yang dipakai untuknya.

Buku ini	Syarat	Nama lain
Full	setiap simpul memiliki 0 atau 2 anak	*proper*, *strictly binary*; "*complete*" di sebagian teks lama
Perfect	*full*, dan semua daun berkedalaman sama	"*full*" di sebagian teks lama
Complete	semua tingkat penuh kecuali yang terakhir, yang diisi dari kiri	*nearly complete*, berbentuk *heap*
Condong	setiap simpul internal memiliki tepat satu anak	*degenerate*, rantai

11.5.3 Berapa banyak simpul muat, dan seberapa rendah pohon dapat dibuat

Karena setiap simpul memiliki paling banyak dua anak, tingkat 0 memuat paling banyak 1 simpul, tingkat 1 paling banyak 2, tingkat 2 paling banyak 4, dan setiap tingkat paling banyak dua kali tingkat di atasnya. Dengan induksi pada ℓ:

$$\text{simpul pada tingkat } \ell \;\le\; 2^{\ell} \tag{11.4}$$

Pohon biner bertinggi h memiliki tingkat 0 sampai h, sehingga menurut Persamaan (11.4) dan jumlah deret geometri ia memiliki paling banyak

$$n \;\le\; 1 + 2 + 4 + \cdots + 2^{h} = \sum_{\ell=0}^{h} 2^{\ell} = 2^{h+1} - 1 \tag{11.5}$$

simpul, dengan kesamaan tepat untuk pohon *perfect*. (Untuk memeriksa jumlah itu: $2 \cdot (1 + 2 + \cdots + 2^{h}) - (1 + 2 + \cdots + 2^{h}) = 2^{h+1} - 1$, karena semua suku tengah saling menghapus.) Dalam kesepakatan yang menomori tingkat akar dengan 1, fakta yang sama berbunyi "tingkat k memuat paling banyak 2^{k-1} simpul, dan pohon dengan k tingkat memuat paling banyak $2^{k} - 1$ simpul"; isinya sama persis.

Balikkan Persamaan (11.5) dan ia menjadi ketaksamaan terpenting dalam bab ini. Bila pohon bertinggi h memuat n simpul, maka $n \le 2^{h+1} - 1$, sehingga $2^{h+1} \ge n + 1$, $h + 1 \ge \lg(n + 1)$ dan, karena h bilangan bulat,

$$h \;\ge\; \lceil \lg(n + 1) \rceil - 1 \;=\; \lfloor \lg n \rfloor \tag{11.6}$$

Tidak ada pohon biner dengan n simpul yang lebih rendah daripada $\lfloor \lg n \rfloor$; pohon *complete* mencapai batas ini, dan pohon condong berada di ujung sebaliknya, $n - 1$. Untuk $n = 1{,}000{,}000$ itulah perbedaan antara tinggi 19 dan tinggi 999.999. Seluruh paruh kedua bab ini membahas ke ujung mana sebuah *binary search tree* akhirnya condong, karena setiap operasi padanya memakan waktu sebanding dengan tingginya.

Satu fakta hitungan lagi berguna. Pada pohon biner *full*, banyak daun L dan banyak simpul internal I memenuhi

$$L = I + 1 \tag{11.7}$$

Bukti: setiap simpul internal memiliki tepat dua anak, sehingga ada $2I$ sisi; menurut Persamaan (11.1) ada juga $n - 1 = L + I - 1$ sisi; menyamakan $2I = L + I - 1$ menghasilkan $L = I + 1$. Pertandingan sistem gugur pada **Gambar 11.1** (d) adalah pohon biner *full*: empat tim (daun) memerlukan tiga pertandingan (simpul internal). Turnamen 64 tim memerlukan 63 pertandingan, bagaimanapun undiannya.

11.6 Menelusuri Pohon Biner

Menelusuri (*traverse*) sebuah pohon berarti mengunjungi setiap simpulnya tepat sekali, sambil melakukan sesuatu pada setiap kunjungan: mencetaknya, menghitungnya, menjumlahkannya. Sebuah list memiliki satu urutan alami; pohon memiliki beberapa, dan masing-masing berguna.

11.6.1 Tiga urutan telusur-dalam

Definisi rekursif pohon biner menyarankan penelusuran rekursif: kunjungi akar, telusuri subpohon kiri, telusuri subpohon kanan. Satu-satunya pilihan adalah *kapan* mengunjungi akar relatif terhadap kedua subpohonnya, dan ketiga pilihan itu menghasilkan tiga urutan klasik:

- ***Preorder*:** kunjungi akar, lalu telusuri subpohon kiri, lalu subpohon kanan.
- ***Inorder*:** telusuri subpohon kiri, lalu kunjungi akar, lalu telusuri subpohon kanan.
- ***Postorder*:** telusuri subpohon kiri, lalu subpohon kanan, lalu kunjungi akar.

Nama-namanya menyebutkan letak akar: *pre*, sebelum subpohonnya; *in*, di antara keduanya; *post*, sesudah keduanya. Pada ketiganya, subpohon kiri ditelusuri sebelum subpohon kanan. **Pseudokode 11.1** menyatakannya.

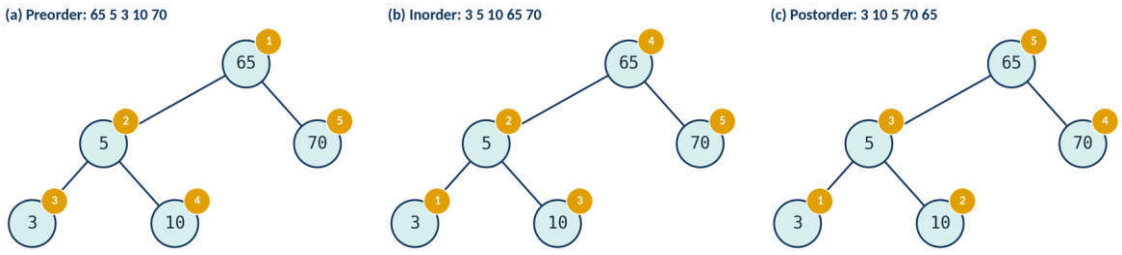
Pseudokode 11.1 Tiga penelusuran telusur-dalam; ketiganya hanya berbeda pada baris tempat simpul itu sendiri dikunjungi.

```
PREORDER(x):
 1.  if x ≠ NIL
 2.      visit x
 3.      PREORDER(x.left)
 4.      PREORDER(x.right)

INORDER(x):
 1.  if x ≠ NIL
 2.      INORDER(x.left)
 3.      visit x
 4.      INORDER(x.right)

POSTORDER(x):
 1.  if x ≠ NIL
 2.      POSTORDER(x.left)
 3.      POSTORDER(x.right)
 4.      visit x
```

Setiap prosedur memiliki bentuk yang diminta Subbab 10.2.5 dari setiap rekursi: kasus dasarnya adalah pohon kosong (`x = NIL`), yang tidak memerlukan kerja, dan setiap panggilan rekursif menerima pohon yang benar-benar lebih kecil. **Gambar 11.8** menomori kunjungan-kunjungan pada pohon yang dibangun oleh `insert(65, 5, 70, 3, 10)`.



Gambar 11.8 Urutan setiap penelusuran mengunjungi simpul-simpul pohon yang sama; lencana kuning pada sebuah simpul adalah posisinya dalam barisan kunjungan.

Telusuri *preorder* dengan tangan, satu panggilan per baris, untuk melihat rekursi bekerja:

1. PREORDER(65): kunjungi 65, lalu panggil PREORDER(5).
2. PREORDER(5): kunjungi 5, lalu panggil PREORDER(3).
3. PREORDER(3): kunjungi 3; kedua anaknya NIL, sehingga kedua panggilan segera kembali; kembali ke 5.

4. Kembali di PREORDER(5): panggil PREORDER(10): kunjungi 10; kembali; PREORDER(5) selesai; kembali ke 65.
5. Kembali di PREORDER(65): panggil PREORDER(70): kunjungi 70; kembali. Selesai.

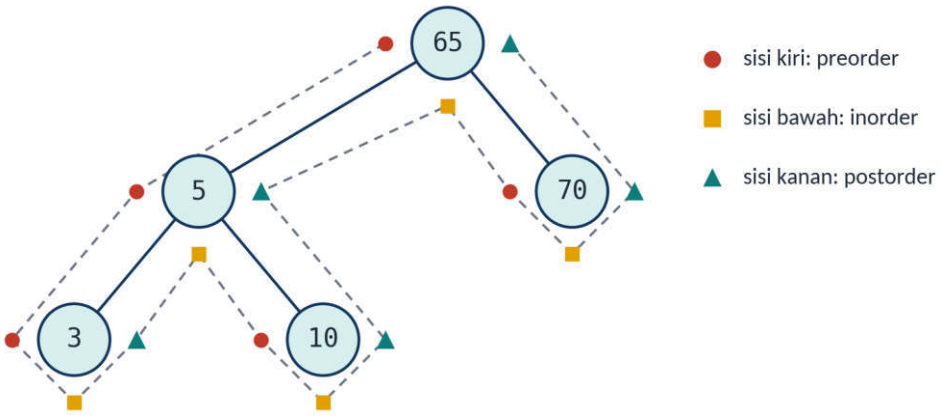
Kunjungannya adalah 65, 5, 3, 10, 70. Penelusuran dengan cara yang sama menghasilkan *inorder* 3, 5, 10, 65, 70 dan *postorder* 3, 10, 5, 70, 65. Perhatikan barisan *inorder*: ia *terurut*. Itu bukan keberuntungan. Subbab 11.7.1 membuktikan bahwa *inorder* mengunjungi kunci-kunci setiap *binary search tree* dalam urutan naik, dan sebagian besar kegunaan pohon pencarian berasal dari sifat itu.

Setiap urutan memiliki pekerjaan alaminya.

- ***Preorder*** mengunjungi sebuah simpul sebelum apa pun di bawahnya, sehingga ia adalah urutan untuk *menyalin* pohon (buat simpulnya, lalu salin subpohon-subpohonnya), untuk *mencetak*nya sebagai kerangka (Subbab 11.3), dan untuk *menyimpan*nya agar dapat dibangun kembali (Subbab 11.10.2).
- ***Inorder*** mengunjungi *binary search tree* dalam urutan terurut, sehingga ia adalah urutan untuk mendaftar, untuk kueri rentang, dan untuk memeriksa bahwa sebuah pohon sungguh pohon pencarian.
- ***Postorder*** mengunjungi sebuah simpul sesudah segala sesuatu di bawahnya, sehingga ia adalah urutan untuk *menghapus* pohon (lepaskan anak-anak sebelum induknya, atau pointer induk ke mereka hilang) dan untuk *menghitung* apa pun yang bergantung pada subpohon: ukuran, tinggi, dan nilai sebuah ekspresi (Subbab 11.6.6).

11.6.2 Berjalan mengelilingi pohon

Ada satu gambar yang memuat ketiga urutan. Bayangkan berjalan mengelilingi bagian luar pohon, mulai dari sebelah kiri akar, dengan pohon selalu di sisi tangan kanan dan masuk ke setiap celah di antara cabang, seperti diperlihatkan **Gambar 11.9**. Perjalanan ini, yang disebut *Euler tour*, melewati setiap simpul tiga kali: pertama di sisi kirinya, lalu di bawahnya (di antara kedua subpohonnya), dan terakhir di sisi kanannya.



Gambar 11.9 Perjalanan mengelilingi pohon menyentuh setiap simpul tiga kali: mendaftar simpul ketika dilewati di sisi kiri menghasilkan *preorder*, di sisi bawah menghasilkan *inorder*, dan di sisi kanan menghasilkan *postorder*.

Mendaftar setiap simpul ketika perjalanan pertama kali melewatinya, di sebelah kirinya, menghasilkan *preorder*; mendaftarnya ketika perjalanan lewat di bawahnya menghasilkan *inorder*; mendaftarnya di sebelah kanannya, yang terakhir kali, menghasilkan *postorder*. Mengikuti garis putus-putus pada **Gambar 11.9** dan hanya membaca titik merah menghasilkan 65, 5, 3, 10, 70; persegi kuning menghasilkan 3, 5, 10, 65, 70; segitiga hijau-biru menghasilkan 3,

10, 5, 70, 65. Rekursi pada **Pseudokode 11.1** *adalah* perjalanan ini: setiap panggilan tiba di simpulnya dari kiri (baris 1), turun ke subpohon kiri lalu kembali lewat bawah, turun ke subpohon kanan lalu kembali di sisi kanan, dan kembali.

11.6.3 Urutan tingkat dengan queue

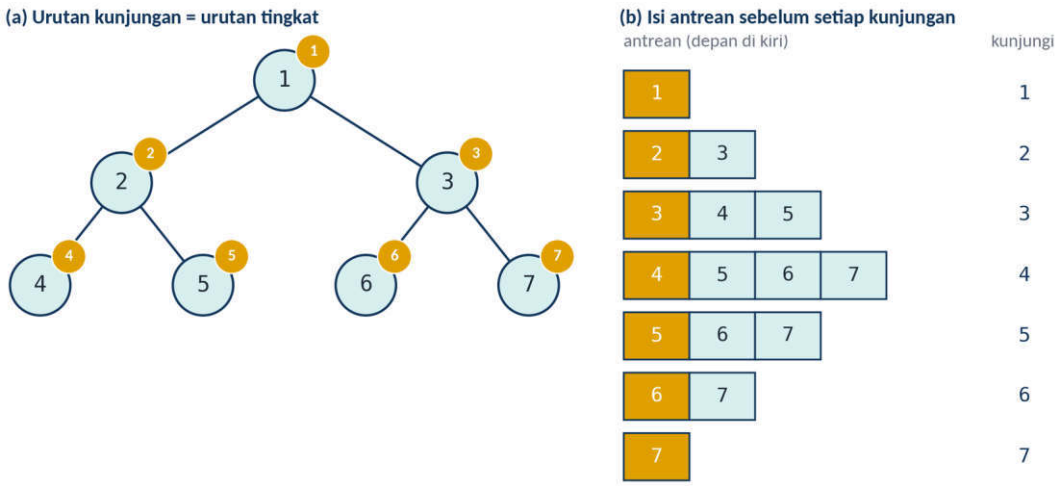
Tiga urutan di atas bersifat *telusur-dalam* (*depth-first*): ia turun sedalam mungkin pada satu cabang sebelum kembali. **Urutan tingkat** (*level order*) bersifat *telusur-lebar* (*breadth-first*): ia mengunjungi akar, lalu setiap simpul pada tingkat 1 dari kiri ke kanan, lalu setiap simpul pada tingkat 2, dan seterusnya. Pada pohon di **Gambar 11.8** urutannya 65, 5, 70, 3, 10.

Urutan tingkat tidak dapat mengikuti rekursi, karena kunjungan yang berurutan (70, lalu 3) berada di subpohon yang berbeda. Ia memerlukan *queue*, struktur FIFO dari Subbab 5.11: simpul yang menunggu dikunjungi masuk di belakang, dan simpul berikutnya yang dikunjungi keluar dari depan. **Pseudokode 11.2** menyatakan algoritmenya.

Pseudokode 11.2 `LEVEL-ORDER`: penelusuran telusur-lebar; anak-anak masuk di belakang *queue*, sehingga satu tingkat utuh dikunjungi sebelum simpul mana pun dari tingkat berikutnya.

```
LEVEL-ORDER(T):
 1.  if T.root = NIL
 2.      return
 3.  Q ← an empty queue
 4.  ENQUEUE(Q, T.root)
 5.  while Q is not empty
 6.      x ← DEQUEUE(Q)            // simpul yang paling lama menunggu
 7.      visit x
 8.      if x.left ≠ NIL
 9.          ENQUEUE(Q, x.left)
10.      if x.right ≠ NIL
11.          ENQUEUE(Q, x.right)
```

Gambar 11.10 mengikuti *queue* itu pada pohon *perfect* yang simpulnya dinomori tingkat demi tingkat, sehingga urutan kunjungannya harus keluar sebagai 1, 2, …, 7.



Gambar 11.10 Urutan tingkat pada pohon *perfect* bersimpul tujuh: (a) urutan kunjungan, (b) isi *queue* tepat sebelum setiap kunjungan, dengan simpul yang akan keluar berwarna kuning.

1. *Queue* dimulai sebagai [1]. Keluarkan 1, kunjungi, masukkan anak-anaknya 2 dan 3: [2, 3].
2. Keluarkan 2, kunjungi, masukkan 4 dan 5: [3, 4, 5].
3. Keluarkan 3, kunjungi, masukkan 6 dan 7: [4, 5, 6, 7].

4. Keluarkan 4, 5, 6, dan 7 bergiliran; semuanya daun, sehingga tidak ada lagi yang dimasukkan, dan *queue* menjadi kosong.

Mengapa urutannya benar? *Queue* selalu memuat simpul-simpul yang belum dikunjungi dari paling banyak dua tingkat berurutan, simpul tingkat saat ini di depan simpul tingkat berikutnya, karena setiap simpul pada tingkat $\ell + 1$ dimasukkan oleh induknya pada tingkat ℓ, dan induk-induk itu dikeluarkan dalam urutan kiri ke kanan. Jadi simpul-simpul setiap tingkat dikunjungi dari kiri ke kanan sebelum simpul mana pun dari tingkat berikutnya. Bab 13 memperumum algoritme ini ke graf sebagai *breadth-first search*.

11.6.4 Penelusuran tanpa rekursi

Penelusuran rekursif bergantung pada *call stack* dari Subbab 10.3 untuk mengingat ke mana harus kembali. *Stack* itu dapat diganti dengan *stack* eksplisit, teknik Subbab 10.13, dan ada dua alasan untuk melakukannya. Untuk pohon yang sangat dalam (pohon merosot dengan sejuta simpul adalah rantai sedalam sejuta panggilan) *stack* eksplisit tinggal di *heap* dan tidak dapat meluapkan *call stack* (Subbab 10.12). Selain itu, penelusuran iteratif dapat dihentikan sejenak lalu dilanjutkan, dan beginilah iterator `std::map` dan `TreeMap` (Subbab 11.11) menyusuri pohon selangkah demi selangkah.

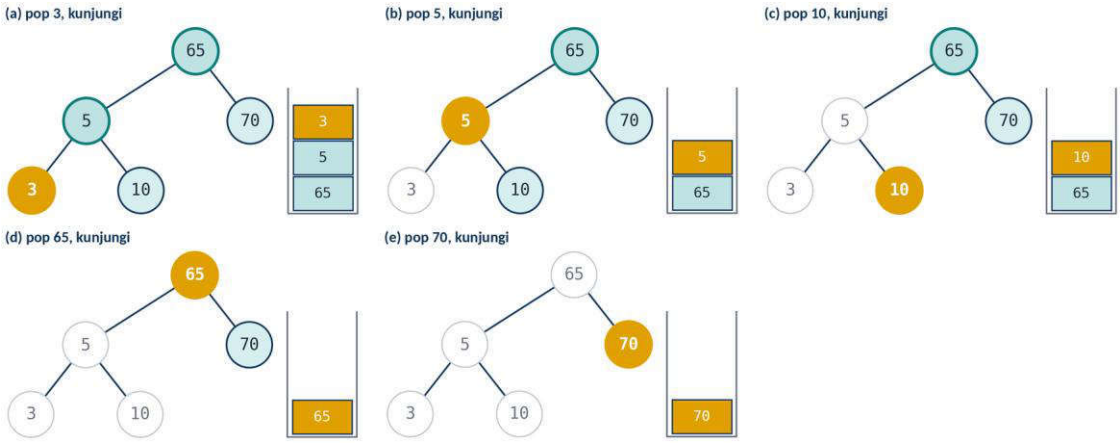
Preorder adalah yang termudah. Dorong akar; lalu berulang kali ambil sebuah simpul dari *stack*, kunjungi, dan dorong anak kanannya lalu anak kirinya. Anak kiri didorong terakhir, sehingga ia diambil pertama, dan subpohon kiri selesai sebelum subpohon kanan dimulai.

Inorder memerlukan kecermatan lebih, karena sebuah simpul harus menunggu sampai seluruh subpohon kirinya dikunjungi. **Pseudokode 11.3** memegang pointer `cur` dan sebuah *stack* berisi simpul-simpul yang subpohon kirinya sedang ditelusuri.

Pseudokode 11.3 Penelusuran *inorder* dengan *stack* eksplisit: turun ke kiri sambil mendorong setiap simpul, ambil dan kunjungi, lalu lanjutkan dengan subpohon kanan.

```
INORDER-ITERATIVE(T):
 1.  S ← an empty stack
 2.  cur ← T.root
 3.  while cur ≠ NIL or S is not empty
 4.      while cur ≠ NIL             // turun ke simpul paling kiri
 5.          PUSH(S, cur)
 6.          cur ← cur.left
 7.      cur ← POP(S)                // subpohon kirinya sudah selesai
 8.      visit cur
 9.      cur ← cur.right             // sekarang subpohon kanannya
```

Gambar 11.11 memperlihatkan isi *stack* pada setiap kunjungan di pohon **Gambar 11.8**. Gelung dalam yang pertama mendorong 65, 5, dan 3 lalu berhenti pada subpohon kiri 3 yang kosong; 3 diambil dan dikunjungi (a). Subpohon kanannya kosong, sehingga gelung luar mengambil 5 dan mengunjunginya (b), lalu berpindah ke anak kanan 5, yaitu 10, mendorongnya, lalu mengambil dan mengunjunginya (c). Sekarang *stack* hanya memuat 65, yang diambil dan dikunjungi (d), dan akhirnya 70 didorong dan dikunjungi (e).



Gambar 11.11 *Stack* eksplisit dari **Pseudokode 11.3** pada setiap kunjungan dari lima kunjungan: simpul hijau-biru berada di *stack*, simpul kuning sedang dikunjungi, dan simpul abu-abu sudah dikunjungi.

Postorder paling sulit dikerjakan dengan satu *stack*, karena sebuah simpul harus dikunjungi sesudah *kedua* subpohonnya. Sebuah trik sederhana memakai dua *stack*: *preorder* yang diubah menjadi akar, kanan, kiri mendorong setiap simpul ke *stack* kedua, dan mengosongkan *stack* kedua lalu menghasilkan kiri, kanan, akar, yaitu *postorder*. (Versi satu *stack* juga mungkin: ia mengingat simpul terakhir yang dikunjungi, untuk mengetahui apakah ia sedang naik kembali dari subpohon kiri atau dari subpohon kanan.) **Kode 11.2** mengimplementasikan keempat penelusuran tanpa rekursi; *queue*-nya dibangun dari simpul bertaut, persis *queue* bertaut dari Subbab 7.10, yang memuat pointer ke simpul pohon.

Kode 11.2 `traversals_iter.cpp`: *preorder*, *inorder*, dan *postorder* dengan *stack* eksplisit, serta urutan tingkat dengan *queue* dari simpul bertaut, disertai jejak isi *stack* dan *queue*.

```cpp
// traversals_iter.cpp -- the four traversals without recursion: preorder,
// inorder and postorder with an explicit stack, level order with a linked queue.
#include <cstdio>
#include <stack>
#include <vector>

struct Node {
    int data;
    Node *left, *right;
};

Node *node(int d, Node *l = nullptr, Node *r = nullptr) {
    return new Node{d, l, r};
}

void destroy(Node *t) {
    if (t == nullptr) return;
    destroy(t->left);
    destroy(t->right);
    delete t;
}

// ---- a queue of Node* built from linked nodes (Chapter 7) ----
struct QNode {
    Node *node;
    QNode *next;
};
struct Queue {
    QNode *front = nullptr, *rear = nullptr;
};
bool isEmptyQ(const Queue &q) { return q.front == nullptr; }
void enQueue(Queue &q, Node *n) {
    QNode *qn = new QNode{n, nullptr};
    if (isEmptyQ(q)) q.front = qn;
    else q.rear->next = qn;
    q.rear = qn;
}
Node *deQueue(Queue &q) {
    QNode *del = q.front;
    Node *n = del->node;
    q.front = del->next;
    if (q.front == nullptr) q.rear = nullptr;   // the queue became empty
    delete del;
    return n;
}
void showQ(const Queue &q) {
```

Kode 11.2 (Lanjutan) *traversals_iter.cpp: preorder, inorder, dan postorder dengan stack eksplisit, serta urutan tingkat dengan queue dari simpul bertaut, disertai jejak isi stack dan queue.*

```
    std::printf("[");
    for (QNode *p = q.front; p != nullptr; p = p->next)
        std::printf(" %d", p->node->data);
    std::printf(" ]");
}

void levelOrder(Node *root, bool trace) {
    if (root == nullptr) return;
    Queue q;
    enQueue(q, root);
    std::printf(trace ? "\n" : "");
    while (!isEmptyQ(q)) {
        if (trace) {
            std::printf("  queue ");
            showQ(q);
        }
        Node *n = deQueue(q);
        std::printf(trace ? "  visit %d\n" : "%d ", n->data);
        if (n->left != nullptr) enQueue(q, n->left);
        if (n->right != nullptr) enQueue(q, n->right);
    }
}

void preOrderIter(Node *root) {
    std::stack<Node *> s;
    if (root != nullptr) s.push(root);
    while (!s.empty()) {
        Node *n = s.top();
        s.pop();
        std::printf("%d ", n->data);
        if (n->right != nullptr) s.push(n->right);  // pushed first, popped later
        if (n->left != nullptr) s.push(n->left);
    }
}

void inOrderIter(Node *root, bool trace) {
    std::vector<Node *> s;              // a vector as a stack, so it can be printed
    Node *cur = root;
    std::printf(trace ? "\n" : "");
    while (cur != nullptr || !s.empty()) {
        while (cur != nullptr) {        // go as far left as possible
            s.push_back(cur);
            cur = cur->left;
        }
        cur = s.back();                 // the leftmost node not yet visited
        s.pop_back();
```

Kode 11.2 (Lanjutan) *traversals_iter.cpp: preorder, inorder, dan postorder dengan stack eksplisit, serta urutan tingkat dengan queue dari simpul bertaut, disertai jejak isi stack dan queue.*

```
        if (trace) {
            std::printf("  visit %-3d stack [", cur->data);
            for (Node *x : s) std::printf(" %d", x->data);
            std::printf(" ]\n");
        } else {
            std::printf("%d ", cur->data);
        }
        cur = cur->right;               // then its right subtree
    }
}

void postOrderIter(Node *root) {        // two stacks
    std::stack<Node *> s1, s2;
    if (root != nullptr) s1.push(root);
    while (!s1.empty()) {
        Node *n = s1.top();
        s1.pop();
        s2.push(n);                     // s2 collects root, right, left
        if (n->left != nullptr) s1.push(n->left);
        if (n->right != nullptr) s1.push(n->right);
    }
    while (!s2.empty()) {               // popping reverses it: left, right, root
        std::printf("%d ", s2.top()->data);
        s2.pop();
    }
}

int main() {
    // The tree built by insert(65, 5, 70, 3, 10).
    Node *t = node(65, node(5, node(3), node(10)), node(70));
    std::printf("PreOrder (stack):   ");
    preOrderIter(t);
    std::printf("\nInOrder (stack):    ");
    inOrderIter(t, false);
    std::printf("\nPostOrder (stacks): ");
    postOrderIter(t);
    std::printf("\nLevelOrder (queue): ");
    levelOrder(t, false);
    std::printf("\nInOrder, traced:");
    inOrderIter(t, true);
    destroy(t);
```

Kode 11.2 (Lanjutan) *traversals_iter.cpp: preorder, inorder, dan postorder dengan stack eksplisit, serta urutan tingkat dengan queue dari simpul bertaut, disertai jejak isi stack dan queue.*

```
    // A perfect tree numbered level by level, 1 to 7.
    Node *p = node(1, node(2, node(4), node(5)), node(3, node(6), node(7)));
    std::printf("LevelOrder of the tree 1..7, traced:");
    levelOrder(p, true);
    destroy(p);
    return 0;
}
```

```
PreOrder (stack):   65 5 3 10 70
InOrder (stack):    3 5 10 65 70
PostOrder (stacks): 3 10 5 70 65
LevelOrder (queue): 65 5 70 3 10
InOrder, traced:
  visit 3   stack [ 65 5 ]
  visit 5   stack [ 65 ]
  visit 10  stack [ 65 ]
  visit 65  stack [ ]
  visit 70  stack [ ]
LevelOrder of the tree 1..7, traced:
  queue [ 1 ]  visit 1
  queue [ 2 3 ]  visit 2
  queue [ 3 4 5 ]  visit 3
  queue [ 4 5 6 7 ]  visit 4
  queue [ 5 6 7 ]  visit 5
  queue [ 6 7 ]  visit 6
  queue [ 7 ]  visit 7
```

Empat baris pertama sesuai dengan urutan rekursif pada **Gambar 11.8**. Jalannya *inorder* yang dilacak memperlihatkan *stack* pada **Gambar 11.11**, dan urutan tingkat yang dilacak mencetak *queue* pada **Gambar 11.10** baris demi baris. Versi Java dan Python (`TraversalsIter.java`, `traversals_iter.py`) mencetak keluaran yang sama.

11.6.5 Ongkos penelusuran

Setiap penelusuran mengunjungi setiap simpul sekali dan melakukan kerja konstan per kunjungan, ditambah kerja konstan untuk setiap subpohon kosong yang diperiksanya. Kita dapat membuktikannya untuk versi rekursif dengan sebuah rekurens. Misalkan $T(n)$ adalah waktu PREORDER pada pohon dengan n simpul yang subpohon kirinya memiliki k simpul (sehingga subpohon kanannya $n-k-1$). Maka, untuk konstanta c dan d,

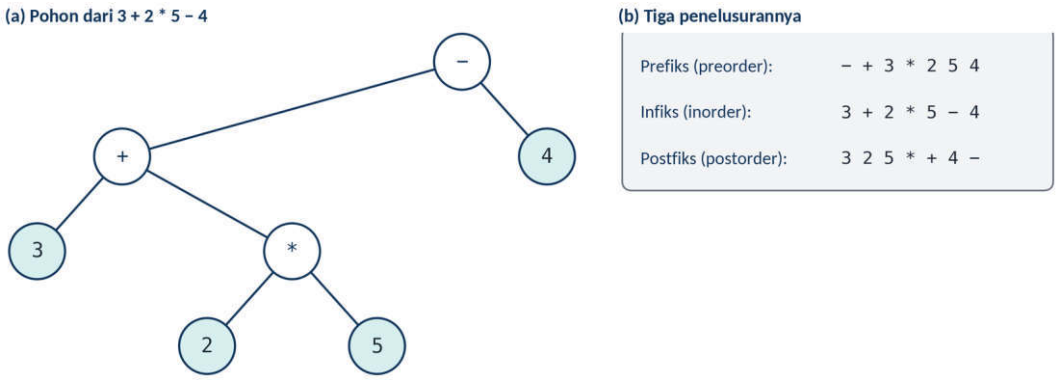
$$T(0) = c, \qquad T(n) = T(k) + T(n-k-1) + d \quad (n \geq 1) \tag{11.8}$$

Klaim: $T(n) = (c+d)\,n + c$, apa pun bentuknya. Untuk $n = 0$ ini menghasilkan c, yang benar. Untuk $n \geq 1$, andaikan klaim berlaku bagi pohon yang lebih kecil (lagi-lagi induksi): $T(n) = (c+d)k + c + (c+d)(n-k-1) + c + d = (c+d)(n-1) + 2c + d = (c+d)n + c$. Jadi setiap penelusuran adalah $\Theta(n)$, dan itu optimal, karena setiap simpul dari n simpul harus dikunjungi. Hal yang sama berlaku untuk *inorder*, *postorder*, dan urutan tingkat.

Ruang memori adalah soal lain, dan ia bergantung pada bentuk. Penelusuran rekursif dan *stack* eksplisit menyimpan paling banyak satu entri per tingkat pada lintasan yang sedang dijelajahi, sehingga memerlukan ruang tambahan $O(h)$, dengan h tinggi pohon: $O(\lg n)$ untuk pohon seimbang tetapi $O(n)$ untuk pohon condong. Urutan tingkat menyimpan sampai satu tingkat penuh dalam *queue*-nya, yaitu $O(w)$ untuk lebar w tingkat terlebar; untuk pohon *perfect* itu sekitar $n/2$ simpul, untuk rantai hanya satu.

11.6.6 Pohon ekspresi

Pohon urai pada **Gambar 11.1** (b) adalah contoh *pohon ekspresi*: pohon biner yang daunnya operan dan simpul internalnya operator biner, yang masing-masing diterapkan pada nilai kedua subpohonnya. **Gambar 11.12** memperlihatkan pohon dari `3 + 2 * 5 - 4`. Perkalian mengikat lebih kuat daripada penjumlahan, dan penjumlahan serta pengurangan dikerjakan dari kiri ke kanan (Subbab 5.8.2), sehingga ekspresi itu berarti $(3 + (2 \cdot 5)) - 4$, dan operator yang diterapkan terakhir, pengurangan, berada di akar.



Gambar 11.12 Pohon ekspresi dari `3 + 2 * 5 - 4` (a) dan tiga penelusuran telusur-dalamnya (b): *preorder* menghasilkan notasi prefiks, *inorder* menghasilkan notasi infiks, dan *postorder* menghasilkan notasi postfiks.

Ketiga penelusuran telusur-dalam sebuah pohon ekspresi adalah persis ketiga notasi pada Subbab 5.8.1:

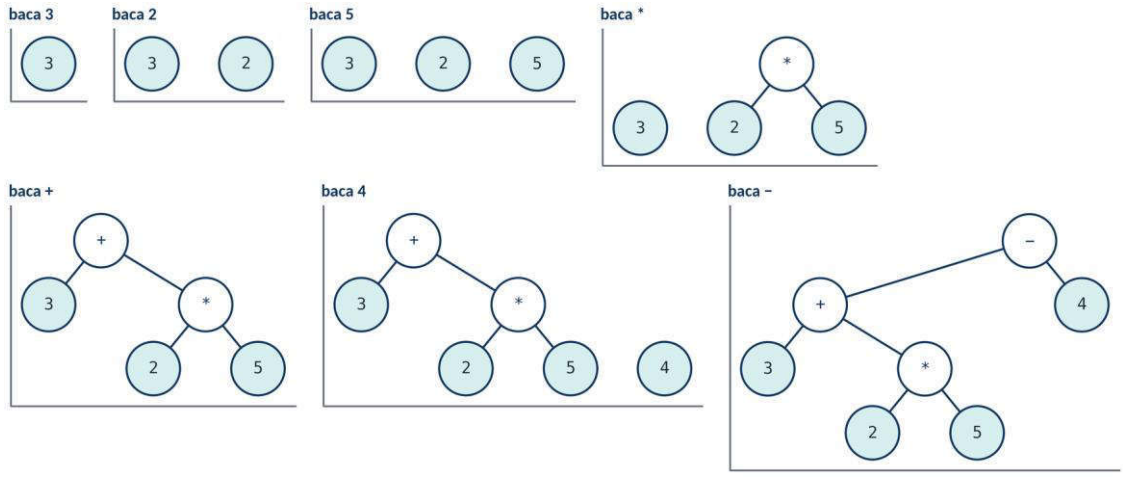
- ***Preorder*** menuliskan setiap operator sebelum operannya: `- + 3 * 2 5 4`, notasi *prefiks* (notasi Polandia).
- ***Inorder*** menuliskan setiap operator di antara operannya: `3 + 2 * 5 - 4`, notasi *infiks* yang biasa. Karena infiks memerlukan aturan prioritas atau tanda kurung agar tidak ambigu, program yang mencetak infiks dari sebuah pohon membubuhkan kurung di sekeliling subpohon setiap operator: `( ( 3 + ( 2 * 5 ) ) - 4 )`.
- ***Postorder*** menuliskan setiap operator sesudah operannya: `3 2 5 * + 4 -`, notasi *postfiks* (notasi Polandia terbalik) yang dievaluasi kalkulator Subbab 5.9 dengan sebuah *stack*.

Mengevaluasi pohon adalah penelusuran *postorder*: untuk mengetahui nilai sebuah simpul, hitung dahulu nilai kedua subpohonnya, lalu terapkan operator simpul itu. Pada daun, nilainya adalah operan itu sendiri.

Membangun pohon dari postfiks adalah algoritme *stack* dari Subbab 5.9 dengan pohon di dalam *stack*, bukan bilangan. Baca ekspresi postfiks dari kiri ke kanan. Sebuah operan menjadi pohon bersimpul satu dan didorong. Sebuah operator mengambil dua pohon, operan kanan lebih dahulu (ia didorong terakhir) lalu operan kiri, menjadikan keduanya anak dari simpul baru yang memuat operator itu, dan mendorong pohon tersebut. Pada akhirnya tersisa tepat satu pohon, dan itulah pohon ekspresinya. **Pseudokode 11.4** menyatakannya, dan **Gambar 11.13** mengikuti *stack*-nya untuk `3 2 5 * + 4 -`.

Pseudokode 11.4 Membangun pohon ekspresi dari notasi postfiks dengan *stack* berisi pohon.

```
BUILD-EXPRESSION-TREE(postfix):
 1.  S ← an empty stack of trees
 2.  for each symbol c of postfix, from left to right
 3.      z ← a new node holding c
 4.      if c is an operator
 5.          z.right ← POP(S)        // operan kanan didorong terakhir
 6.          z.left ← POP(S)
 7.      PUSH(S, z)
 8.  return POP(S)                   // satu-satunya pohon yang tersisa
```



Gambar 11.13 *Stack* berisi pohon selama `3 2 5 * + 4 -` dibaca: operan mendorong pohon bersimpul satu, dan setiap operator menggabungkan dua pohon terbaru menjadi satu.

Kode 11.3 membangun pohon itu, mencetaknya dalam tiga notasi, dan mengevaluasinya; ia juga membangun kembali pohon urai pada **Gambar 11.1** (b) dari bentuk postfiksnya, `abc+*de+-`.

Kode 11.3 `expr_tree.cpp`: pohon ekspresi yang dibangun dari notasi postfiks, dicetak dalam notasi prefiks, infiks, dan postfiks, serta dievaluasi dengan penelusuran *postorder*.

```
// expr_tree.cpp -- an expression tree built from postfix with a stack, printed
// in prefix, infix and postfix notation, and evaluated.
#include <cstdio>
#include <stack>
#include <string>

struct Node {
    char data;              // an operator or a one-character operand
    Node *left, *right;
};

Node *createNode(char data) { return new Node{data, nullptr, nullptr}; }

bool isOperator(char c) { return c == '+' || c == '-' || c == '*' || c == '/'; }

Node *constructExpressionTree(const std::string &postfix, bool trace) {
    std::stack<Node *> st;
    for (char c : postfix) {
        Node *newNode = createNode(c);
        if (isOperator(c)) {                // the two newest trees become children
            newNode->right = st.top();      // popped first: the RIGHT operand
            st.pop();
            newNode->left = st.top();
            st.pop();
        }
        st.push(newNode);
        if (trace)
            std::printf("  read %c: stack holds %zu tree(s)\n", c, st.size());
    }
    return st.top();                        // exactly one tree is left
}

void displayPrefix(const Node *t) {         // preorder
    if (t == nullptr) return;
    std::printf("%c ", t->data);
    displayPrefix(t->left);
    displayPrefix(t->right);
}
```

Kode 11.3 (Lanjutan) *expr_tree.cpp: pohon ekspresi yang dibangun dari notasi postfiks, dicetak dalam notasi prefiks, infiks, dan postfiks, serta dievaluasi dengan penelusuran postorder.*

```
void displayInfix(const Node *t) {          // inorder, with parentheses
    if (t == nullptr) return;
    if (isOperator(t->data)) std::printf("( ");
    displayInfix(t->left);
    std::printf("%c ", t->data);
    displayInfix(t->right);
    if (isOperator(t->data)) std::printf(") ");
}

void displayPostfix(const Node *t) {        // postorder
    if (t == nullptr) return;
    displayPostfix(t->left);
    displayPostfix(t->right);
    std::printf("%c ", t->data);
}

double evaluate(const Node *t) {            // postorder again: operands first
    if (!isOperator(t->data)) return t->data - '0';
    double a = evaluate(t->left), b = evaluate(t->right);
    switch (t->data) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        default:  return a / b;
    }
}

void destroy(Node *t) {
    if (t == nullptr) return;
    destroy(t->left);
    destroy(t->right);
    delete t;
}

void show(const std::string &postfix, bool numeric) {
    std::printf("Postfix input: %s\n", postfix.c_str());
    Node *root = constructExpressionTree(postfix, numeric);
    std::printf("Prefix Notation:  ");
    displayPrefix(root);
    std::printf("\nInfix Notation:   ");
    displayInfix(root);
    std::printf("\nPostfix Notation: ");
    displayPostfix(root);
    if (numeric) std::printf("\nValue: %g", evaluate(root));
    std::printf("\n\n");
    destroy(root);
```

Kode 11.3 (Lanjutan) *expr_tree.cpp: pohon ekspresi yang dibangun dari notasi postfiks, dicetak dalam notasi prefiks, infiks, dan postfiks, serta dievaluasi dengan penelusuran postorder.*

```
}

int main() {
    show("325*+4-", true);          // 3 + 2 * 5 - 4
    show("abc+*de+-", false);       // (a * (b + c)) - (d + e)
    return 0;
}
```

```
Postfix input: 325*+4-
  read 3: stack holds 1 tree(s)
  read 2: stack holds 2 tree(s)
  read 5: stack holds 3 tree(s)
  read *: stack holds 2 tree(s)
  read +: stack holds 1 tree(s)
  read 4: stack holds 2 tree(s)
  read -: stack holds 1 tree(s)
Prefix Notation:  - + 3 * 2 5 4
Infix Notation:   ( ( 3 + ( 2 * 5 ) ) - 4 )
Postfix Notation: 3 2 5 * + 4 -
Value: 9

Postfix input: abc+*de+-
Prefix Notation:  - * a + b c + d e
Infix Notation:   ( ( a * ( b + c ) ) - ( d + e ) )
Postfix Notation: a b c + * d e + -
```

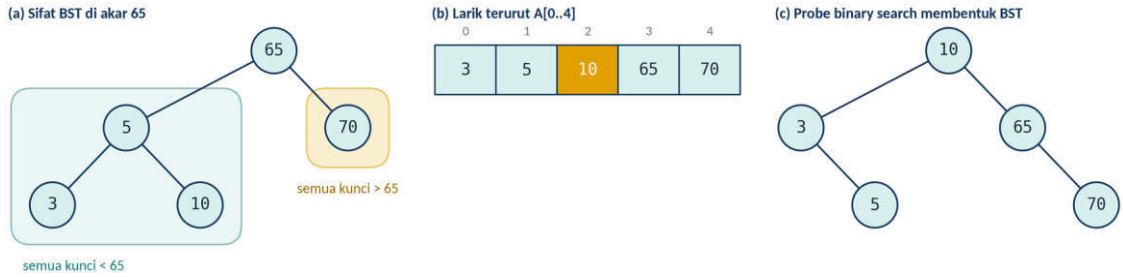
Nilai 9 dapat diperiksa dengan tangan: $2 \cdot 5 = 10$, $3 + 10 = 13$, $13 - 4 = 9$. Kompilator melakukan hal yang sama dalam skala besar: pengurai membangun pohon dari seluruh program, dan tahap-tahap berikutnya menelusurinya, secara *postorder* untuk menghitung tipe dan menghasilkan kode, secara *preorder* untuk mencetaknya.

11.7 Binary Search Tree

11.7.1 Sifat BST

Ingat kembali *binary search* pada larik terurut (Subbab 2.5). Probe pertama melihat elemen tengah; bila kuncinya lebih kecil, seluruh separuh kanan dibuang, dan probe berikutnya melihat bagian tengah separuh kiri. Gambarlah setiap kemungkinan barisan probe dan Anda memperoleh sebuah pohon biner: elemen tengah di akar, tengah separuh kiri sebagai anak kirinya, tengah separuh kanan sebagai anak kanannya, dan seterusnya. **Gambar 11.14** (b) dan (c) menggambarnya untuk lima kunci 3, 5, 10, 65, 70. Pada pohon itu setiap kunci di subpohon kiri sebuah simpul lebih kecil daripada kunci simpul itu, dan setiap kunci di subpohon kanan lebih besar. *Binary search tree* (pohon pencarian biner) adalah persis pohon ini, dikeluarkan dari larik dan dibangun dari simpul-simpul, sehingga ia dapat tumbuh dan menyusut.

> **DEFINISI.** Sebuah *binary search tree* (BST) adalah pohon biner yang setiap simpulnya x menyimpan sebuah kunci $x.key$, dan untuk setiap simpul x: setiap kunci di subpohon kiri x lebih kecil daripada $x.key$, dan setiap kunci di subpohon kanan x lebih besar daripada $x.key$.



Gambar 11.14 (a) Sifat BST di akar: setiap kunci subpohon kiri lebih kecil daripada 65 dan setiap kunci subpohon kanan lebih besar; (b) dan (c) probe-probe *binary search* pada larik terurut membentuk sebuah *binary search tree*.

Gagasan ini diterbitkan secara terpisah oleh beberapa orang sekitar tahun 1960, di antaranya P. F. Windley [3], A. D. Booth dan A. J. T. Colin, serta T. N. Hibbard [4], yang semuanya mencari cara menjaga sebuah tabel tetap terurut selagi tabel itu tumbuh; Knuth [8] menuturkan kisahnya.

Perhatikan kata-kata *setiap kunci di subpohon*. Tidak cukup bahwa *anak* kiri lebih kecil dan *anak* kanan lebih besar; syaratnya harus berlaku untuk setiap keturunan. Soal 11.8 memperlihatkan pohon yang lolos uji lokal di setiap simpul tetapi tetap bukan pohon pencarian. Sifat itu juga harus berlaku di *setiap* simpul, bukan hanya di akar: 5 lebih besar daripada setiap kunci di subpohon kirinya (3) dan lebih kecil daripada setiap kunci di subpohon kanannya (10).

Definisi di atas memakai ketaksamaan tegas, sehingga sebuah BST menyimpan setiap kunci paling banyak sekali, dan itulah yang diperlukan katalog kode buku yang unik atau sebuah himpunan. Pohon yang harus menyimpan duplikat menyimpan pencacah di setiap simpul, atau bersepakat mengirim kunci yang sama secara konsisten ke satu sisi (program laundry pada Subbab 11.12 mengirimnya ke kanan).

Sifat BST memiliki akibat langsung yang sudah kita lihat pada **Gambar 11.8**:

> **INORDER ITU TERURUT.** Penelusuran *inorder* pada *binary search tree* mengunjungi kunci-kuncinya dalam urutan naik. *Bukti* dengan induksi pada banyak simpul. Pohon kosong menghasilkan barisan kosong, yang terurut. Untuk pohon tak kosong berakar r, *inorder* mendaftar subpohon kiri, lalu r, lalu subpohon kanan. Menurut hipotesis induksi setiap subpohon terdaftar dalam urutan naik; menurut sifat BST setiap kunci pada daftar pertama lebih kecil daripada $r.key$ dan setiap kunci pada daftar terakhir lebih besar. Jadi gabungannya naik.

Inilah sebabnya BST adalah *kamus yang tetap terurut*: ia menjawab "apakah kunci ini ada?" seperti tabel *hash* (Bab 15), tetapi ia juga dapat mendaftar kuncinya secara berurutan, melaporkan yang terkecil dan terbesar, dan menemukan tetangga sebuah kunci, yang tidak sanggup dilakukan tabel *hash*.

11.7.2 Penyisipan: `insert(65, 5, 70, 3, 10)`

Untuk menyisipkan kunci z, berjalanlah turun dari akar seolah-olah mencari z: di setiap simpul belok ke kiri bila z lebih kecil daripada kunci simpul itu dan ke kanan bila lebih besar. Ketika perjalanan jatuh keluar dari pohon, pada sebuah subpohon kosong, simpul baru ditempatkan persis di sana, sebagai daun baru. Bila perjalanan bertemu simpul yang sudah memuat z, kunci itu duplikat dan tidak disisipkan lagi. Rumusan rekursifnya hanya tiga baris:

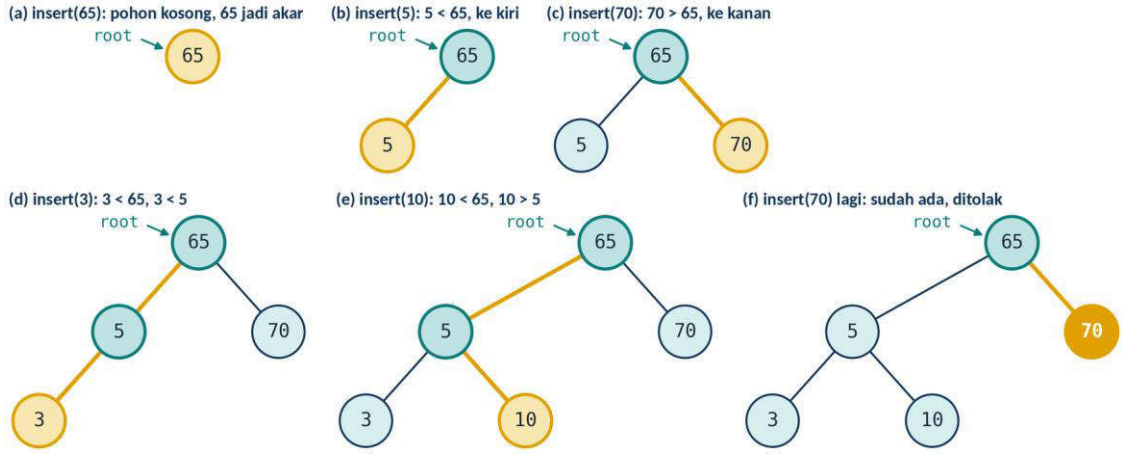
Pseudokode 11.5 `TREE-INSERT`: turun seperti pada pencarian dan gantungkan simpul baru pada subpohon kosong tempat perjalanan berakhir.

```
TREE-INSERT(x, z):              // sisipkan kunci z ke subpohon yang pointer akarnya x
 1.  if x = NIL
 2.      x ← a new node with key z, left NIL, right NIL
 3.  else if z < x.key
 4.      TREE-INSERT(x.left, z)
 5.  else if z > x.key
 6.      TREE-INSERT(x.right, z)
 7.  else
 8.      report "z has existed in the tree"
```

Baris 2 harus mengubah *pointer* yang tadinya NIL, yaitu medan `left` atau `right` milik induk, atau pointer akar itu sendiri bila pohon kosong. Karena itulah parameter x berarti "pointer ke subpohon", bukan salinannya. C mengungkapkan hal ini dengan pointer ke pointer, `Node **root`, persis teknik `init(Student **p)` pada Subbab 7.4.4; C++ juga dapat memakai referensi, `Node *&root`; sedangkan Java dan Python, yang tidak memiliki keduanya, menulis fungsi itu sedemikian rupa sehingga ia *mengembalikan* akar subpohon sesudah penyisipan dan membiarkan pemanggil

menyimpannya: `t.left = insert(t.left, z)`. Ketiga gaya itu muncul dalam program-program bab ini.

Sekarang contoh berjalannya. Mulai dari pohon kosong, sisipkan 65, 5, 70, 3, dan 10, dalam urutan itu; **Gambar 11.15** memperlihatkan pohon sesudah setiap panggilan, dengan lintasan yang diikuti berwarna hijau-biru dan kuning serta simpul baru berwarna kuning.



Gambar 11.15 Pohon sesudah setiap panggilan `insert(65, 5, 70, 3, 10)` dan satu `insert(70)` ulangan: simpul hijau-biru dan sisi kuning adalah lintasan pencarian, simpul kuning adalah daun baru, dan 70 yang kembar ditemukan lalu ditolak.

1. `insert(65)` (a). Pohon kosong, sehingga `root` bernilai NIL: baris 2 menciptakan simpul 65, dan simpul itu menjadi akar.
2. `insert(5)` (b). $5 < 65$, sehingga belok ke kiri. Subpohon kiri 65 kosong: 5 menjadi anak kiri 65.
3. `insert(70)` (c). $70 > 65$, belok ke kanan; subpohon kanan kosong: 70 menjadi anak kanan 65.
4. `insert(3)` (d). $3 < 65$, belok ke kiri menuju 5; $3 < 5$, belok ke kiri; kosong: 3 menjadi anak kiri 5.
5. `insert(10)` (e). $10 < 65$, belok ke kiri menuju 5; $10 > 5$, belok ke kanan; kosong: 10 menjadi anak kanan 5.
6. `insert(70)` **lagi** (f). $70 > 65$, belok ke kanan menuju 70; $70 = 70$: baris 8 melaporkan bahwa 70 sudah ada, dan pohon tidak berubah.

Periksa sifat BST pada hasilnya: setiap kunci di bawah 5 di sebelah kiri lebih kecil daripada 5 (3), di sebelah kanan lebih besar (10); segala sesuatu di bawah 65 di kiri (3, 5, 10) lebih kecil daripada 65, dan di kanan (70) lebih besar. Penelusuran *inorder*-nya 3, 5, 10, 65, 70, sebagaimana mestinya.

Kunci yang sama dalam urutan lain menghasilkan pohon lain. Menyisipkan 10, 5, 3, 65, 70 akan menjadikan 10 akarnya. Bentuk sebuah BST merekam *riwayat* penyisipannya, dan Subbab 11.8 memperlihatkan betapa pentingnya hal itu.

Penyisipan juga dapat ditulis tanpa rekursi, dengan sebuah pointer yang berjalan turun dan pointer *pengekor* ke induknya, seperti `deleteKey` mengekorkan `prev` di belakang `cur` pada Subbab 7.6.5. Eksperimen tinggi pada Subbab 11.8.2 memakai bentuk itu, dengan teknik pointer-ke-tautan dari Subbab 7.7.3, karena penyisipan rekursif ke pohon merosot bersimpul 32.000 akan menumpuk 32.000 panggilan.

Ongkos. Perjalanan mengunjungi satu simpul per tingkat, sehingga penyisipan ke pohon bertinggi h melakukan paling banyak $h + 1$ perbandingan dan kerja lain sebanyak $O(1)$: waktu $O(h)$.

11.7.3 Pencarian

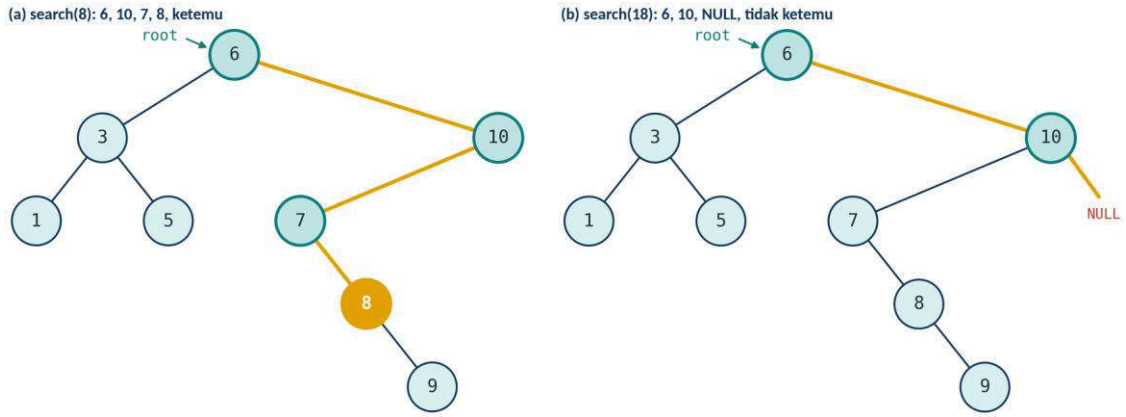
Pencarian adalah penyisipan tanpa menyisipkan. Di setiap simpul, bandingkan kunci dengan kunci simpul itu: sama berarti ketemu; lebih kecil berarti kunci itu hanya mungkin berada di subpohon kiri, sehingga belok ke kiri; lebih besar berarti belok ke kanan. Mencapai NIL berarti kunci itu tidak ada di pohon. **Pseudokode 11.6** memberikan bentuk rekursif dan iteratifnya; bentuk iteratif tidak memerlukan *stack*, karena setiap langkah berpindah ke satu anak dan tidak pernah perlu kembali.

Pseudokode 11.6 Pencarian pada *binary search tree*, secara rekursif dan iteratif; keduanya mengembalikan simpul yang memuat *k*, atau NIL.

```
TREE-SEARCH(x, k):
 1.  if x = NIL or k = x.key
 2.      return x
 3.  if k < x.key
 4.      return TREE-SEARCH(x.left, k)
 5.  else
 6.      return TREE-SEARCH(x.right, k)

ITERATIVE-TREE-SEARCH(x, k):
 1.  while x ≠ NIL and k ≠ x.key
 2.      if k < x.key
 3.          x ← x.left
 4.      else
 5.          x ← x.right
 6.  return x
```

Setiap perbandingan membuang satu subpohon utuh, sama seperti setiap probe *binary search* membuang separuh larik. **Gambar 11.16** menelusuri dua pencarian pada pohon yang dibangun dengan menyisipkan 6, 3, 10, 1, 5, 7, 8, 9.



Gambar 11.16 Dua pencarian: (a) `search(8)` mengikuti 6, 10, 7 dan menemukan 8; (b) `search(18)` mengikuti 6 dan 10, jatuh keluar dari pohon pada subpohon kanan 10 yang kosong, dan melaporkan bahwa 18 tidak ada.

- `search(8)`. Di akar, 6: $8 > 6$, sehingga 8 hanya mungkin berada di subpohon kanan; belok ke kanan. Di 10: $8 < 10$, belok ke kiri. Di 7: $8 > 7$, belok ke kanan. Di 8: sama, ketemu. Empat perbandingan, padahal pohon itu memiliki delapan simpul; subpohon berakar 3 dan subpohon kanan 10 yang kosong tidak pernah disentuh.
- `search(18)`. Di 6: $18 > 6$, belok ke kanan. Di 10: $18 > 10$, belok ke kanan. Subpohon kanan 10 kosong: 18 tidak ada di pohon. Dua perbandingan sudah cukup untuk membuktikannya.

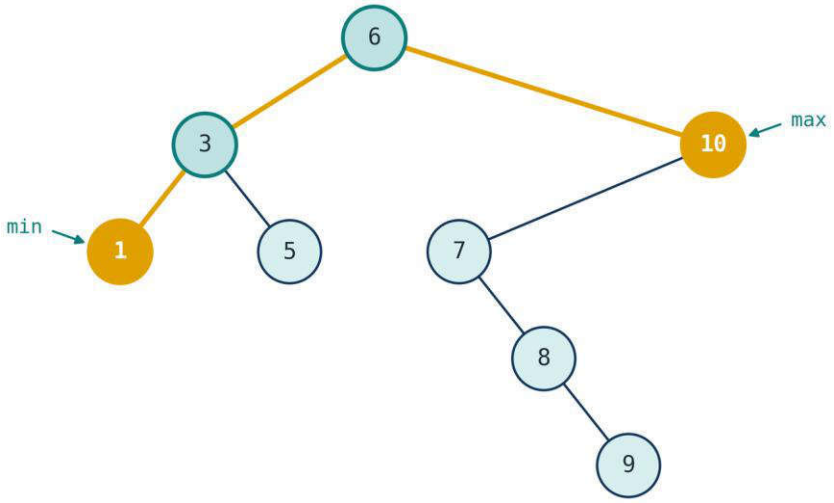
Ongkos. Satu perbandingan per tingkat pada satu lintasan akar-ke-daun: $O(h)$ dalam kasus terburuk, dan kasus terburuk itu adalah pencarian yang gagal dan turun sepanjang lintasan terpanjang.

11.7.4 Minimum dan maksimum

Kunci terkecil sebuah BST berada di simpul *paling kiri*: mulai dari akar, ikuti pointer `left` sampai pointer berikutnya bernilai NIL. Mengapa? Setiap simpul yang memiliki anak kiri mempunyai kunci yang lebih kecil di subpohon kirinya, sehingga ia tidak mungkin minimum; simpul paling kiri tidak memiliki anak kiri, dan setiap simpul lain adalah salah satu leluhurnya (yang kuncinya lebih besar, karena simpul paling kiri berada di subpohon kiri mereka) atau berada di subpohon kanan salah satu leluhur itu (lebih besar lagi). Secara simetris, kunci terbesar berada di simpul paling kanan. **Gambar 11.17** memperlihatkan kedua perjalanan itu.

Pseudokode 11.7 Minimum adalah simpul paling kiri dan maksimum adalah simpul paling kanan dari pohon yang tidak kosong.

```
TREE-MINIMUM(x):                    TREE-MAXIMUM(x):
 1.  while x.left ≠ NIL              1.  while x.right ≠ NIL
 2.      x ← x.left                  2.      x ← x.right
 3.  return x                        3.  return x
```



Gambar 11.17 Minimum, 1, dicapai dengan terus ke kiri dari akar sampai tidak ada anak kiri; maksimum, 10, dengan terus ke kanan sampai tidak ada anak kanan.

Perhatikan maksimum, 10, pada **Gambar 11.17**: ia bukan daun. Ia memiliki subpohon kiri (7, 8, 9), tetapi tidak memiliki anak kanan, dan hanya itulah yang diperlukan maksimum. Kedua perjalanan mengikuti satu lintasan: $O(h)$.

11.7.5 Menghitung simpul, daun, dan tinggi

Tiga besaran mengikuti definisi rekursif secara langsung, masing-masing sebuah perhitungan *postorder*: hitung dahulu jawaban untuk kedua subpohon, lalu gabungkan.

- **Banyak simpul** pohon kosong adalah 0; selain itu, banyaknya adalah banyak simpul di subpohon kiri ditambah banyak simpul di subpohon kanan ditambah 1 untuk akar.
- **Banyak daun** pohon kosong adalah 0; simpul tunggal tanpa anak adalah 1 daun; selain itu, banyaknya adalah jumlah dari kedua subpohon.

- **Tinggi** pohon kosong adalah -1; selain itu, tingginya satu lebih besar daripada yang lebih tinggi di antara kedua subpohon:

$$height(\text{NIL}) = -1, \qquad height(x) = 1 + \max(height(x.left),\ height(x.right)) \tag{11.9}$$

Kesepakatan $height(\text{NIL}) = -1$ itulah yang membuat Persamaan (11.9) benar untuk daun: $1 + \max(-1,-1) = 0$. Dengan kesepakatan lain yang juga lazim, tinggi dihitung dalam *simpul* dan bukan sisi, pohon kosong bertinggi 0 dan daun bertinggi 1; rekursinya sama dengan 0 pada kasus dasar, dan setiap jawaban menjadi satu lebih besar.

Telusuri Persamaan (11.9) pada pohon `insert(65, 5, 70, 3, 10)`: daun 3, 10, dan 70 bertinggi 0; simpul 5 bertinggi $1 + \max(0,0) = 1$; akar bertinggi $1 + \max(1,0) = 2$. Pada pohon **Gambar 11.16**, lintasan terpanjangnya 6, 10, 7, 8, 9, empat sisi, sehingga tingginya 4.

Ketiga fungsi itu mengunjungi setiap simpul sekali: masing-masing $\Theta(n)$, menurut Persamaan (11.8). Fungsi yang mencetak daun, dari kiri ke kanan, adalah perjalanan yang sama dengan mencetak sebagai ganti menghitung.

11.7.6 Mencetak pohon secara menyamping

Terminal mencetak teks baris demi baris, dari atas ke bawah, dan itu canggung untuk pohon yang tumbuh ke bawah. Trik sederhananya adalah mencetak pohon yang diputar 90 derajat berlawanan arah jarum jam, dengan akar di kiri dan setiap tingkat menjorok satu langkah lebih jauh ke kanan. Simpul paling kanan lalu harus muncul di baris *pertama*, sehingga penelusurannya adalah *inorder terbalik*: subpohon kanan, simpul, subpohon kiri, dengan setiap simpul menjorok empat spasi per tingkat kedalaman.

Pseudokode 11.8 Mencetak pohon secara menyamping: *inorder* terbalik, menjorok sesuai kedalaman.

```
PRINT-SIDEWAYS(x, depth):
 1.  if x ≠ NIL
 2.      PRINT-SIDEWAYS(x.right, depth + 1)
 3.      print 4 · depth spaces, then x.key
 4.      PRINT-SIDEWAYS(x.left, depth + 1)
```

Untuk membaca keluarannya, miringkan kepala Anda ke kiri: kolom paling kiri adalah akar, kolom berikutnya tingkat 1, dan seterusnya; anak kanan sebuah simpul adalah baris terdekat di atasnya yang satu kolom lebih ke kanan, dan anak kirinya baris terdekat di bawahnya. Subbab 11.7.7 memperlihatkan hasilnya untuk kedua pohon contoh.

11.7.7 Program lengkapnya

Kode 11.4 menyatukan Subbab 11.7.2 sampai 11.7.6. Fungsi `insert`-nya menerima alamat pointer yang harus diubah, `Node **root`, seperti diperlihatkan `insert(&(*root)->left, newData)`: `(*root)->left` adalah medan pointer kiri simpul saat ini, dan `&` mengambil alamatnya, sehingga panggilan rekursif dapat menyimpan simpul baru ke medan milik induk. Fungsi `main`-nya membangun contoh berjalan, termasuk 70 kembar yang ditolak, mencetak keempat penelusuran dan semua hitungan, mencetak pohon secara menyamping, lalu melepaskannya; kemudian ia membangun pohon **Gambar 11.16** dan menjalankan kedua pencarian pada gambar itu, sambil mencetak setiap keputusan.

Kode 11.4 `bst.cpp`: *binary search tree* dengan penyisipan, empat penelusuran, pencarian, hitungan simpul, hitungan daun, tinggi, minimum, maksimum, daftar daun, pencetakan menyamping, dan pelepasan semua simpul.

```
// bst.cpp -- a binary search tree: insert, traversals, search, count, height,
// minimum, maximum, leaves, printing sideways, and clearing.
#include <cstdio>
#include <queue>

typedef struct MyNode {
    int data;
    struct MyNode *left;    // subtree of smaller keys
    struct MyNode *right;   // subtree of larger keys
} Node;

void insert(Node **root, int newData) {     // root is the ADDRESS of a pointer
    if (*root == nullptr) {                 // an empty subtree: the node goes here
        Node *newNode = new Node;
        newNode->data = newData;
        newNode->left = nullptr;
        newNode->right = nullptr;
        *root = newNode;                    // changes the caller's pointer
        std::printf("%d has been inserted by insert()\n", newData);
    } else if (newData < (*root)->data) {
        insert(&(*root)->left, newData);
    } else if (newData > (*root)->data) {
        insert(&(*root)->right, newData);
    } else {
        std::printf("%d has existed in the tree\n", newData);
    }
}

void preOrder(const Node *root) {           // root, left, right
    if (root != nullptr) {
        std::printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}

void inOrder(const Node *root) {            // left, root, right
    if (root != nullptr) {
        inOrder(root->left);
        std::printf("%d ", root->data);
        inOrder(root->right);
    }
}
```

Kode 11.4 (Lanjutan) *bst.cpp: binary search tree dengan penyisipan, empat penelusuran, pencarian, hitungan simpul, hitungan daun, tinggi, minimum, maksimum, daftar daun, pencetakan menyamping, dan pelepasan semua simpul.*

```
void postOrder(const Node *root) {          // left, right, root
    if (root != nullptr) {
        postOrder(root->left);
        postOrder(root->right);
        std::printf("%d ", root->data);
    }
}

void levelOrder(Node *root) {               // breadth first, with a queue
    if (root == nullptr) {
        std::printf("Tree is empty.");
        return;
    }
    std::queue<Node *> q;
    q.push(root);
    while (!q.empty()) {
        Node *n = q.front();
        q.pop();
        std::printf("%d ", n->data);                    // visit the node
        if (n->left != nullptr) q.push(n->left);        // children join the back
        if (n->right != nullptr) q.push(n->right);
    }
}

Node *search(Node *root, int data) {        // prints the path it takes
    if (root == nullptr) {
        std::printf("  reached NULL: %d cannot be found by search()\n", data);
        return nullptr;
    } else if (data < root->data) {
        std::printf("  %d < %d, go left\n", data, root->data);
        return search(root->left, data);
    } else if (data > root->data) {
        std::printf("  %d > %d, go right\n", data, root->data);
        return search(root->right, data);
    } else {
        std::printf("  %d has been found by search()\n", data);
        return root;
    }
}

int count(const Node *root) {
    if (root == nullptr) return 0;
    return count(root->left) + count(root->right) + 1;
}
```

Kode 11.4 (Lanjutan) *bst.cpp: binary search tree dengan penyisipan, empat penelusuran, pencarian, hitungan simpul, hitungan daun, tinggi, minimum, maksimum, daftar daun, pencetakan menyamping, dan pelepasan semua simpul.*

```
int countLeaves(const Node *root) {
    if (root == nullptr) return 0;
    if (root->left == nullptr && root->right == nullptr) return 1;
    return countLeaves(root->left) + countLeaves(root->right);
}

int height(const Node *root) {              // edges on the longest downward path
    if (root == nullptr) return -1;         // the empty tree has height -1
    int u = height(root->left), v = height(root->right);
    return (u > v ? u : v) + 1;
}

Node *min(Node *root) {                     // the leftmost node
    if (root == nullptr) return nullptr;
    if (root->left == nullptr) return root;
    return min(root->left);
}

Node *max(Node *root) {                     // the rightmost node
    if (root == nullptr) return nullptr;
    if (root->right == nullptr) return root;
    return max(root->right);
}

void leaf(const Node *root) {               // prints the leaves, left to right
    if (root == nullptr) return;
    if (root->left == nullptr && root->right == nullptr)
        std::printf("%d ", root->data);
    leaf(root->left);
    leaf(root->right);
}

void printSideways(const Node *root, int depth) {   // right subtree on top
    if (root == nullptr) return;
    printSideways(root->right, depth + 1);
    std::printf("%*s%d\n", 4 * depth, "", root->data);
    printSideways(root->left, depth + 1);
}
```

Kode 11.4 (Lanjutan) *bst.cpp: binary search tree dengan penyisipan, empat penelusuran, pencarian, hitungan simpul, hitungan daun, tinggi, minimum, maksimum, daftar daun, pencetakan menyamping, dan pelepasan semua simpul.*

```
int clear(Node **root) {                    // postorder: children before parent
    if (*root == nullptr) return 0;
    int freed = clear(&(*root)->left) + clear(&(*root)->right);
    delete *root;
    *root = nullptr;
    return freed + 1;
}

void report(Node *root) {
    std::printf("PreOrder:   ");
    preOrder(root);
    std::printf("\nInOrder:    ");
    inOrder(root);
    std::printf("\nPostOrder:  ");
    postOrder(root);
    std::printf("\nLevelOrder: ");
    levelOrder(root);
    std::printf("\nNodes: %d, leaves: %d, height: %d, min: %d, max: %d\n",
                count(root), countLeaves(root), height(root),
                min(root)->data, max(root)->data);
    std::printf("Leaves: ");
    leaf(root);
    std::printf("\nThe tree, printed sideways (root on the left):\n");
    printSideways(root, 0);
}

int main() {
    std::printf("Create a root, it's an empty node...\n");
    Node *root = nullptr;
    int first[] = {65, 5, 70, 3, 10, 70};
    for (int x : first) {
        std::printf("Insert a node %d ... ", x);
        insert(&root, x);
    }
    std::printf("\n");
    report(root);
    std::printf("Clear the tree ... %d nodes deleted\n\n", clear(&root));
```

Kode 11.4 (Lanjutan) *bst.cpp: binary search tree dengan penyisipan, empat penelusuran, pencarian, hitungan simpul, hitungan daun, tinggi, minimum, maksimum, daftar daun, pencetakan menyamping, dan pelepasan semua simpul.*

```
    std::printf("Build a tree ...\n");
    int second[] = {6, 3, 10, 1, 5, 7, 8, 9};
    for (int x : second) insert(&root, x);
    std::printf("\n");
    report(root);
    std::printf("Searching 8 ...\n");
    search(root, 8);
    std::printf("Searching 18 ...\n");
    search(root, 18);
    std::printf("Clear the tree ... %d nodes deleted\n", clear(&root));
    return 0;
}
```

```
Create a root, it's an empty node...
Insert a node 65 ... 65 has been inserted by insert()
Insert a node 5 ... 5 has been inserted by insert()
Insert a node 70 ... 70 has been inserted by insert()
Insert a node 3 ... 3 has been inserted by insert()
Insert a node 10 ... 10 has been inserted by insert()
Insert a node 70 ... 70 has existed in the tree

PreOrder:   65 5 3 10 70
InOrder:    3 5 10 65 70
PostOrder:  3 10 5 70 65
LevelOrder: 65 5 70 3 10
Nodes: 5, leaves: 3, height: 2, min: 3, max: 70
Leaves: 3 10 70
The tree, printed sideways (root on the left):
    70
65
        10
    5
        3
Clear the tree ... 5 nodes deleted

Build a tree ...
6 has been inserted by insert()
3 has been inserted by insert()
10 has been inserted by insert()
1 has been inserted by insert()
5 has been inserted by insert()
7 has been inserted by insert()
8 has been inserted by insert()
9 has been inserted by insert()

PreOrder:   6 3 1 5 10 7 8 9
InOrder:    1 3 5 6 7 8 9 10
PostOrder:  1 5 3 9 8 7 10 6
LevelOrder: 6 3 10 1 5 7 8 9
Nodes: 8, leaves: 3, height: 4, min: 1, max: 10
Leaves: 1 5 9
The tree, printed sideways (root on the left):
    10
            9
        8
        7
6
        5
    3
        1
Searching 8 ...
  8 > 6, go right
  8 < 10, go left
  8 > 7, go right
  8 has been found by search()
Searching 18 ...
  18 > 6, go right
  18 > 10, go right
  reached NULL: 18 cannot be found by search()
Clear the tree ... 8 nodes deleted
```

Bacalah keluaran itu sambil melihat gambar-gambarnya. Penelusuran telusur-dalam pohon pertama adalah yang ada pada **Gambar 11.8**, dan urutan tingkatnya 65 5 70 3 10; hitungannya 5 simpul, 3 daun (3, 10, 70), dan tinggi 2; cetakan menyampingnya, bila dimiringkan, adalah **Gambar 11.15** (e). Untuk pohon kedua, urutan tingkat 6 3 10 1 5 7 8 9 adalah pohon **Gambar 11.16** yang dibaca tingkat demi tingkat; tingginya, 4, adalah lintasan 6, 10, 7, 8, 9; dan kedua jejak pencarian persis lintasan yang digambar pada gambar itu.

Dua perincian patut dicermati. Pertama, `clear` harus berupa penelusuran *postorder*: ia melepaskan kedua subpohon sebelum menghapus simpul itu sendiri, karena begitu simpul dihapus, medan `left` dan `right`-nya tidak dapat lagi dibaca. Sesudah itu ia mengisi pointer milik pemanggil dengan `nullptr`, sehingga pohon kembali menjadi pohon kosong yang sah. `clear` yang hanya mengisi `root = nullptr` tanpa menghapus apa pun dapat dikompilasi, berjalan, dan mencetak hal yang benar, tetapi membocorkan setiap simpul; Valgrind melaporkan "All heap blocks were freed – no leaks are possible" untuk **Kode 11.4**, dan itulah bukti bahwa versi kita tidak bocor. Kedua, `min(root)->data` menganggap pohon tidak kosong; pada pohon kosong `min` mengembalikan `nullptr` dan `->` akan menjadi perilaku tak terdefinisi (Subbab 7.11), sehingga `report` hanya dipanggil untuk pohon yang tidak kosong.

11.7.8 Pohon yang sama dalam Java dan Python

Java dan Python tidak memiliki pointer ke pointer, sehingga `insert` milik keduanya mengembalikan akar subpohon yang diterimanya, dan pemanggil menyimpan hasilnya: `root = insert(root, x)` di tingkat teratas, `t.left = insert(t.left, newData)` di dalam fungsi. Bila subpohonnya kosong, akar yang dikembalikan adalah simpul baru; bila tidak, ia simpul yang sama seperti sebelumnya, dan penugasan itu tidak mengubah apa pun. **Kode 11.5** adalah versi Java; `clear` cukup membuang referensi ke akar, dan *garbage collector* mengambil kembali setiap simpul yang tidak lagi terjangkau.

Kode 11.5 `Bst.java`: *binary search tree* dari **Kode 11.4** dalam Java; `insert` mengembalikan akar baru setiap subpohon.

```java
// Bst.java -- a binary search tree: insert, traversals, search, count, height,
// minimum, maximum, leaves, printing sideways, and clearing.
import java.util.ArrayDeque;

public class Bst {
    static class Node {
        int data;
        Node left, right;   // subtrees of smaller and of larger keys

        Node(int data) {
            this.data = data;
        }
    }

    private Node root;      // null means the tree is empty

    void insert(int newData) {
        root = insert(root, newData);
    }

    // Returns the root of the subtree after the insertion (Java has no Node **).
    private Node insert(Node t, int newData) {
        if (t == null) {
            System.out.println(newData + " has been inserted by insert()");
            return new Node(newData);
        } else if (newData < t.data) {
            t.left = insert(t.left, newData);
        } else if (newData > t.data) {
            t.right = insert(t.right, newData);
        } else {
            System.out.println(newData + " has existed in the tree");
        }
        return t;
    }

    static void preOrder(Node t) {
        if (t != null) {
            System.out.print(t.data + " ");
            preOrder(t.left);
            preOrder(t.right);
        }
    }
```

Kode 11.5 (Lanjutan) *Bst.java: binary search tree dari Kode 11.4 dalam Java; insert mengembalikan akar baru setiap subpohon.*

```
    static void inOrder(Node t) {
        if (t != null) {
            inOrder(t.left);
            System.out.print(t.data + " ");
            inOrder(t.right);
        }
    }

    static void postOrder(Node t) {
        if (t != null) {
            postOrder(t.left);
            postOrder(t.right);
            System.out.print(t.data + " ");
        }
    }

    static void levelOrder(Node t) {
        if (t == null) {
            System.out.print("Tree is empty.");
            return;
        }
        ArrayDeque<Node> q = new ArrayDeque<>();
        q.add(t);
        while (!q.isEmpty()) {
            Node n = q.remove();
            System.out.print(n.data + " ");
            if (n.left != null) q.add(n.left);
            if (n.right != null) q.add(n.right);
        }
    }

    static Node search(Node t, int data) {
        if (t == null) {
            System.out.println("  reached NULL: " + data
                    + " cannot be found by search()");
            return null;
        } else if (data < t.data) {
            System.out.println("  " + data + " < " + t.data + ", go left");
            return search(t.left, data);
        } else if (data > t.data) {
            System.out.println("  " + data + " > " + t.data + ", go right");
            return search(t.right, data);
        } else {
            System.out.println("  " + data + " has been found by search()");
            return t;
        }
```

Kode 11.5 (Lanjutan) *Bst.java: binary search tree dari Kode 11.4 dalam Java; insert mengembalikan akar baru setiap subpohon.*

```
    }

    static int count(Node t) {
        if (t == null) return 0;
        return count(t.left) + count(t.right) + 1;
    }

    static int countLeaves(Node t) {
        if (t == null) return 0;
        if (t.left == null && t.right == null) return 1;
        return countLeaves(t.left) + countLeaves(t.right);
    }

    static int height(Node t) {
        if (t == null) return -1;
        return Math.max(height(t.left), height(t.right)) + 1;
    }

    static Node min(Node t) {
        if (t == null) return null;
        return t.left == null ? t : min(t.left);
    }

    static Node max(Node t) {
        if (t == null) return null;
        return t.right == null ? t : max(t.right);
    }

    static void leaf(Node t) {
        if (t == null) return;
        if (t.left == null && t.right == null) System.out.print(t.data + " ");
        leaf(t.left);
        leaf(t.right);
    }

    static void printSideways(Node t, int depth) {
        if (t == null) return;
        printSideways(t.right, depth + 1);
        System.out.println(" ".repeat(4 * depth) + t.data);
        printSideways(t.left, depth + 1);
    }
```

Kode 11.5 (Lanjutan) *Bst.java: binary search tree dari Kode 11.4 dalam Java; insert mengembalikan akar baru setiap subpohon.*

```
    int clear() {                   // the garbage collector frees the nodes
        int n = count(root);
        root = null;
        return n;
    }

    void report() {
        System.out.print("PreOrder:   ");
        preOrder(root);
        System.out.print("\nInOrder:    ");
        inOrder(root);
        System.out.print("\nPostOrder:  ");
        postOrder(root);
        System.out.print("\nLevelOrder: ");
        levelOrder(root);
        System.out.printf("%nNodes: %d, leaves: %d, height: %d, min: %d, max: %d%n",
                count(root), countLeaves(root), height(root), min(root).data,
                max(root).data);
        System.out.print("Leaves: ");
        leaf(root);
        System.out.println("\nThe tree, printed sideways (root on the left):");
        printSideways(root, 0);
    }

    public static void main(String[] args) {
        System.out.println("Create a root, it's an empty node...");
        Bst tree = new Bst();
        for (int x : new int[] {65, 5, 70, 3, 10, 70}) {
            System.out.print("Insert a node " + x + " ... ");
            tree.insert(x);
        }
        System.out.println();
        tree.report();
        System.out.println("Clear the tree ... " + tree.clear()
                + " nodes deleted\n");
```

Kode 11.5 (Lanjutan) *Bst.java: binary search tree dari Kode 11.4 dalam Java; insert mengembalikan akar baru setiap subpohon.*

```
        System.out.println("Build a tree ...");
        for (int x : new int[] {6, 3, 10, 1, 5, 7, 8, 9}) tree.insert(x);
        System.out.println();
        tree.report();
        System.out.println("Searching 8 ...");
        search(tree.root, 8);
        System.out.println("Searching 18 ...");
        search(tree.root, 18);
        System.out.println("Clear the tree ... " + tree.clear() + " nodes deleted");
    }
}
```

Kode 11.6 adalah versi Python. Penelusuran-penelusurannya menambahkan ke sebuah list alih-alih langsung mencetak, sehingga pencetakan terkumpul di satu tempat; `__slots__` menetapkan tiga atribut sebuah simpul, seperti pada Subbab 7.5.9.

Kode 11.6 `bst.py`: *binary search tree* yang sama dalam Python.

```
# bst.py -- a binary search tree: insert, traversals, search, count, height,
# minimum, maximum, leaves, printing sideways, and clearing.
from collections import deque


class Node:
    __slots__ = ("data", "left", "right")

    def __init__(self, data):
        self.data = data
        self.left = None        # subtree of smaller keys
        self.right = None       # subtree of larger keys


def insert(t, new_data):
    """Returns the root of the subtree after the insertion."""
    if t is None:
        print(f"{new_data} has been inserted by insert()")
        return Node(new_data)
    if new_data < t.data:
        t.left = insert(t.left, new_data)
    elif new_data > t.data:
        t.right = insert(t.right, new_data)
    else:
        print(f"{new_data} has existed in the tree")
    return t


def pre_order(t, out):          # root, left, right
    if t is not None:
        out.append(t.data)
        pre_order(t.left, out)
        pre_order(t.right, out)


def in_order(t, out):           # left, root, right
    if t is not None:
        in_order(t.left, out)
        out.append(t.data)
        in_order(t.right, out)
```

Kode 11.6 (Lanjutan) *bst.py: binary search tree yang sama dalam Python.*

```
def post_order(t, out):         # left, right, root
    if t is not None:
        post_order(t.left, out)
        post_order(t.right, out)
        out.append(t.data)


def level_order(t, out):        # breadth first, with a queue
    if t is None:
        return
    q = deque([t])
    while q:
        n = q.popleft()
        out.append(n.data)
        if n.left is not None:
            q.append(n.left)
        if n.right is not None:
            q.append(n.right)


def search(t, data):
    if t is None:
        print(f"  reached NULL: {data} cannot be found by search()")
        return None
    if data < t.data:
        print(f"  {data} < {t.data}, go left")
        return search(t.left, data)
    if data > t.data:
        print(f"  {data} > {t.data}, go right")
        return search(t.right, data)
    print(f"  {data} has been found by search()")
    return t


def count(t):
    return 0 if t is None else count(t.left) + count(t.right) + 1


def count_leaves(t):
    if t is None:
        return 0
    if t.left is None and t.right is None:
        return 1
    return count_leaves(t.left) + count_leaves(t.right)
```

Kode 11.6 (Lanjutan) *bst.py: binary search tree yang sama dalam Python.*

```
def height(t):                  # the empty tree has height -1
    return -1 if t is None else max(height(t.left), height(t.right)) + 1


def tree_min(t):                # the leftmost node
    if t is None:
        return None
    return t if t.left is None else tree_min(t.left)


def tree_max(t):                # the rightmost node
    if t is None:
        return None
    return t if t.right is None else tree_max(t.right)


def leaves(t, out):
    if t is None:
        return
    if t.left is None and t.right is None:
        out.append(t.data)
    leaves(t.left, out)
    leaves(t.right, out)


def print_sideways(t, depth=0):
    if t is None:
        return
    print_sideways(t.right, depth + 1)
    print(" " * (4 * depth) + str(t.data))
    print_sideways(t.left, depth + 1)


def show(label, walk, t):
    out = []
    walk(t, out)
    print(label + "".join(f"{x} " for x in out))
```

Kode 11.6 (Lanjutan) *bst.py: binary search tree yang sama dalam Python.*

```
def report(root):
    show("PreOrder:   ", pre_order, root)
    show("InOrder:    ", in_order, root)
    show("PostOrder:  ", post_order, root)
    show("LevelOrder: ", level_order, root)
    print(f"Nodes: {count(root)}, leaves: {count_leaves(root)}, "
          f"height: {height(root)}, min: {tree_min(root).data}, "
          f"max: {tree_max(root).data}")
    show("Leaves: ", leaves, root)
    print("The tree, printed sideways (root on the left):")
    print_sideways(root)


def main():
    print("Create a root, it's an empty node...")
    root = None
    for x in (65, 5, 70, 3, 10, 70):
        print(f"Insert a node {x} ... ", end="")
        root = insert(root, x)
    print()
    report(root)
    print(f"Clear the tree ... {count(root)} nodes deleted\n")
    root = None                 # the garbage collector frees the nodes

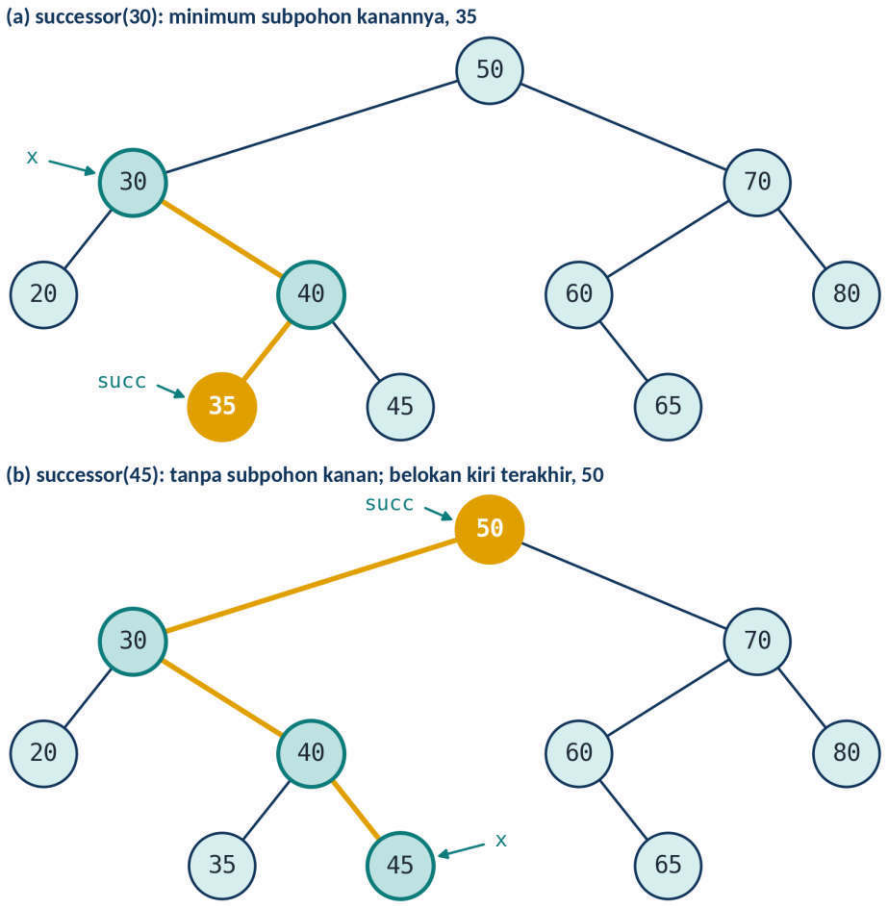
    print("Build a tree ...")
    for x in (6, 3, 10, 1, 5, 7, 8, 9):
        root = insert(root, x)
    print()
    report(root)
    print("Searching 8 ...")
    search(root, 8)
    print("Searching 18 ...")
    search(root, 18)
    print(f"Clear the tree ... {count(root)} nodes deleted")


if __name__ == "__main__":
    main()
```

Kedua program mencetak keluaran yang persis sama dengan **Kode 11.4**. Satu peringatan berlaku khususnya untuk versi Python: kedalaman rekursinya sama dengan tinggi pohon, dan batas rekursi bawaan CPython adalah 1.000 (Subbab 10.12.3). Menyisipkan 2.000 kunci dalam urutan terurut ke `insert` ini membangun rantai bertinggi 1.999 dan memunculkan `RecursionError`. Subbab 11.8 menjelaskan mengapa pohon seperti itu muncul dan bagaimana menghindarinya; penyisipan iteratif pada `height_experiment.py` tidak memiliki batas seperti itu.

11.7.9 Successor dan predecessor

Successor sebuah kunci x dalam BST adalah kunci terkecil yang lebih besar daripada x: kunci yang datang tepat sesudah x dalam barisan *inorder*. *Predecessor* adalah kunci terbesar yang lebih kecil daripada x. Pada pohon **Gambar 11.18**, yang barisan *inorder*-nya 20, 30, 35, 40, 45, 50, 60, 65, 70, 80, *successor* dari 30 adalah 35 dan *successor* dari 45 adalah 50. Keduanya dapat ditemukan dalam waktu $O(h)$ tanpa melihat kunci-kunci lain, dan ada dua kasus.



Gambar 11.18 Dua kasus *successor*: (a) 30 memiliki subpohon kanan, dan *successor*-nya adalah minimum subpohon itu, 35; (b) 45 tidak memiliki subpohon kanan, dan *successor*-nya adalah leluhur tempat lintasan dari akar terakhir kali berbelok ke kiri, 50.

- **Kasus 1: x memiliki subpohon kanan.** Setiap kunci yang lebih besar daripada x dan dekat dengan x terletak di subpohon itu, dan yang terkecil di antaranya adalah minimum subpohon itu: belok ke kanan sekali, lalu ke kiri sejauh mungkin. Pada **Gambar 11.18** (a), subpohon kanan 30 berakar 40, yang simpul paling kirinya 35.
- **Kasus 2: x tidak memiliki subpohon kanan.** Maka *successor*-nya berada di atas x. Berjalanlah naik pada lintasan dari x menuju akar: selama Anda tiba di sebuah simpul dari anak kanannya, simpul itu lebih kecil daripada x (Anda datang dari subpohon kanannya); pertama kali Anda tiba di sebuah simpul dari anak *kiri*-nya, simpul itu lebih besar daripada x dan itulah *successor*-nya. Dilihat dari akar, ia adalah simpul terakhir pada lintasan pencarian tempat lintasan itu *berbelok ke kiri*. Pada **Gambar 11.18** (b), lintasan menuju 45 melewati 50 (belok kiri), 30 (belok kanan), 40 (belok kanan), 45; belokan kiri terakhir terjadi di 50.
- Bila x tidak memiliki subpohon kanan dan lintasannya tidak pernah berbelok ke kiri, x adalah maksimum dan tidak memiliki *successor*.

Simpul-simpul kita tidak memiliki pointer ke induk, sehingga kita tidak dapat berjalan naik. Sebagai gantinya kita berjalan *turun*, sambil mengingat simpul terakhir tempat kita berbelok ke kiri, seperti dilakukan **Pseudokode 11.9**. *Predecessor* adalah bayangan cerminnya: maksimum subpohon kiri bila ada, selain itu simpul terakhir tempat lintasan pencarian berbelok ke kanan.

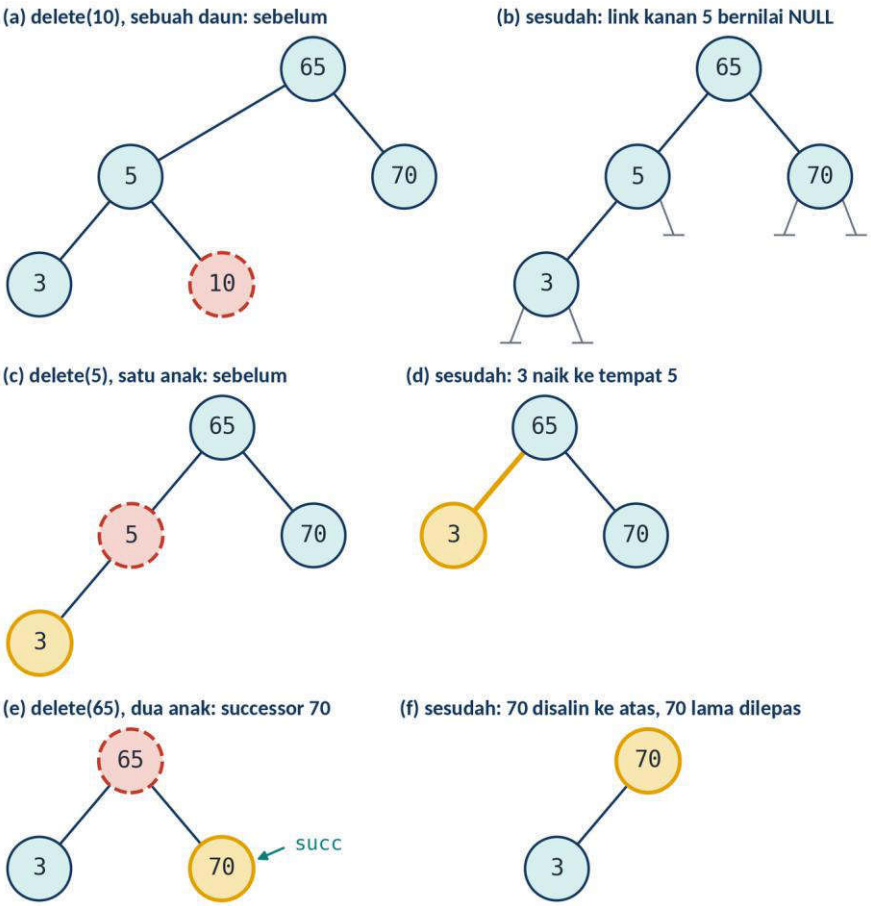
Pseudokode 11.9 `TREE-SUCCESSOR` tanpa pointer induk: cari k sambil mengingat belokan kiri terakhir.

```
TREE-SUCCESSOR(root, k):              // k ada di dalam pohon
 1.  cand ← NIL                       // simpul terakhir tempat lintasan belok kiri
 2.  x ← root
 3.  while x.key ≠ k
 4.      if k < x.key
 5.          cand ← x
 6.          x ← x.left
 7.      else x ← x.right
 8.  if x.right ≠ NIL
 9.      return TREE-MINIMUM(x.right)  // kasus 1
10.  return cand                      // kasus 2 (NIL bila k maksimum)
```

Pohon yang menyimpan pointer `parent` di setiap simpul (seperti pohon *red-black* di balik `std::map` dan `TreeMap`) dapat menemukan *successor* dari simpul yang sudah dipegangnya dengan berjalan naik, tanpa pencarian dari akar; Cormen dkk. [2] memberikan versi itu. Bagaimanapun, ongkosnya $O(h)$. Menjalani seluruh pohon dengan panggilan *successor* berulang, mulai dari minimum, hanya menghabiskan $\Theta(n)$ secara total, karena setiap sisi dilintasi paling banyak dua kali; beginilah `++it` pada iterator `std::map` bekerja.

11.7.10 Penghapusan: tiga kasus

Penghapusan adalah satu-satunya operasi BST yang benar-benar pelik, karena simpul yang dihapus mungkin memiliki anak yang harus tetap berada di pohon, dan sifat BST harus tetap berlaku sesudahnya. Pertama, cari simpul z yang memuat kunci itu. Lalu ada tiga kasus, menurut banyak anak z; **Gambar 11.19** menerapkannya bergiliran pada contoh berjalan.



Gambar 11.19 Tiga kasus penghapusan, diterapkan satu per satu pada pohon `insert(65, 5, 70, 3, 10)`: daun dilepas (a, b); simpul beranak satu digantikan oleh anaknya (c, d); simpul beranak dua mengambil kunci *successor*-nya, yang kemudian dihapus dari subpohon kanan (e, f).

Kasus 1: *z* adalah daun. Tidak ada apa pun yang tergantung di bawah *z*, sehingga cukup isi pointer induk ke *z* dengan NIL dan lepaskan simpulnya. Menghapus 10, anak kanan 5 (**Gambar 11.19** (a)), membuat pointer kanan 5 bernilai NIL (b). Barisan *inorder* kehilangan 10 dan tetap terurut.

Kasus 2: *z* memiliki tepat satu anak. Sambung-lewati *z*: jadikan induk *z* menunjuk langsung ke satu-satunya anak *z*, lalu lepaskan *z*. Anak itu naik satu tingkat, membawa seluruh subpohonnya. Sifat BST bertahan, karena setiap kunci di subpohon itu sudah berada di sisi yang sama dari setiap leluhur *z* seperti *z* sendiri. Menghapus 5, yang kini hanya memiliki anak kiri 3 ((c)), membuat pointer kiri 65 menunjuk ke 3 ((d)).

Kasus 3: *z* memiliki dua anak. Sekarang tidak satu pun anak dapat begitu saja mengambil tempat *z*, karena tidak ada tempat untuk menggantungkan anak yang lain. Triknya, yang berasal dari Hibbard [4], adalah mencari kunci yang dapat *menggantikan* kunci *z* tanpa merusak urutan: *successor z*, yaitu minimum subpohon kanannya. Kunci itu lebih besar daripada semua kunci di subpohon kiri *z* dan lebih kecil daripada semua kunci lain di subpohon kanan, sehingga ia pas persis di tempat *z*. Salin kunci *successor* ke *z*, lalu hapus *successor* itu dari subpohon kanan. Penghapusan kedua itu selalu mudah, karena *successor*, sebagai simpul paling kiri subpohon kanan, *tidak memiliki anak kiri*: ia termasuk kasus 1 atau kasus 2. Menghapus 65 ((e)): *successor*-nya 70, minimum subpohon kanannya; 70 disalin ke akar, dan 70 yang lama, sebuah daun, dilepas ((f)).

Pseudokode 11.10 menyatakan prosedurnya dalam gaya rekursif yang mengembalikan akar baru subpohon, gaya `insert` pada Java dan Python.

Pseudokode 11.10 `TREE-DELETE`, varian *successor*: daun dibuang, simpul beranak satu digantikan anaknya, dan simpul beranak dua mengambil kunci *successor*-nya.

```
TREE-DELETE(x, k):                  // kembalikan akar subpohon sesudah k dihapus
 1.  if x = NIL
 2.      return NIL                 // k tidak ada di pohon
 3.  if k < x.key
 4.      x.left ← TREE-DELETE(x.left, k)
 5.  else if k > x.key
 6.      x.right ← TREE-DELETE(x.right, k)
 7.  else if x.left = NIL and x.right = NIL      // kasus 1: daun
 8.      free x; return NIL
 9.  else if x.left = NIL or x.right = NIL       // kasus 2: satu anak
10.      c ← the non-NIL child of x
11.      free x; return c
12.  else                                       // kasus 3: dua anak
13.      s ← TREE-MINIMUM(x.right)              // successor
14.      x.key ← s.key
15.      x.right ← TREE-DELETE(x.right, s.key)
16.  return x
```

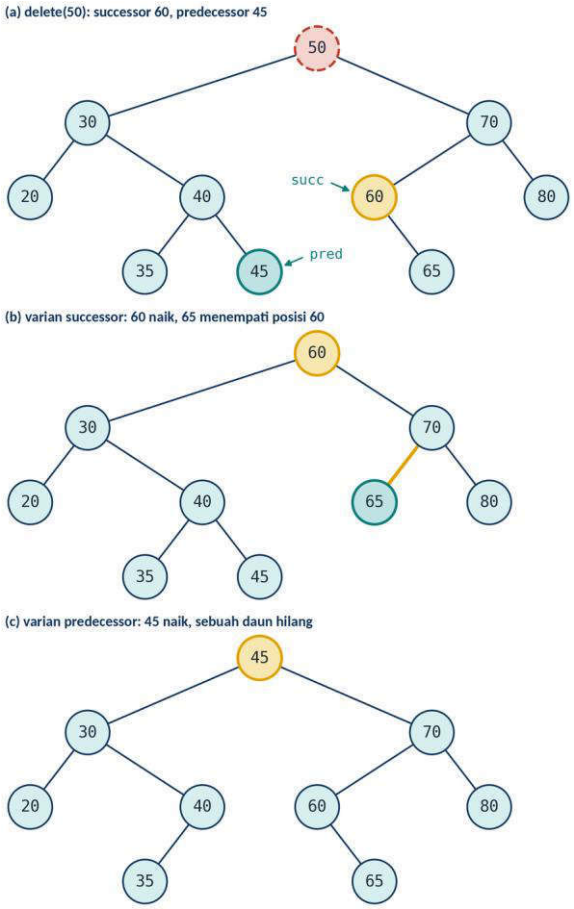
Rekursi itu mengurus pembukuan induk untuk kita: setiap panggilan mengembalikan akar subpohonnya *sesudah* penghapusan, dan pemanggil menyimpannya ke pointer yang tadi diikutinya (baris 4 dan 6). Ketika sebuah simpul dibuang, panggilan yang membuangnya mengembalikan NIL atau anak yang tersisa, dan nilai itu mendarat di pointer induk. Ketika akar sendiri yang dihapus, panggilan teratas mengembalikan akar baru. Tidak diperlukan kasus khusus untuk akar, sama seperti teknik pointer-ke-tautan pada Subbab 7.7.3 tidak memerlukannya untuk simpul pertama sebuah list.

Satu kehalusan lagi: pada kasus 3 simpul *z* tidak dilepaskan; *kunci*-nya ditimpa dan *simpul successor*-lah yang dilepaskan. Itu tidak masalah bila simpul hanya memuat kunci. Bila simpul

membawa rekaman besar, atau bila bagian lain program memegang pointer ke simpul-simpul tertentu (iterator `std::map` berjanji tetap sah untuk semua elemen yang tidak dihapus), menyalin tidak dapat diterima, dan simpul *successor* justru *dipindahkan* ke posisi *z* dengan menautkan ulang pointer. Cormen dkk. [2] menguraikan versi itu, dengan fungsi pembantu yang mereka namai TRANSPLANT; ongkosnya sama.

11.7.11 Dua varian: successor atau predecessor

Successor bukan satu-satunya pengganti yang mungkin pada kasus 3. *Predecessor* dari *z*, yaitu maksimum subpohon kirinya, sama pasnya: ia lebih kecil daripada semua kunci lain di subpohon kiri dan lebih besar daripada semua kunci di subpohon kanan. Ia tidak memiliki anak kanan, sehingga menghapusnya lagi-lagi termasuk kasus 1 atau kasus 2. **Gambar 11.20** menghapus akar 50 dari sebuah pohon bersimpul sepuluh dengan kedua cara.



Gambar 11.20 Menghapus simpul beranak dua dengan kedua cara: (a) akar 50 memiliki *successor* 60 dan *predecessor* 45; (b) dengan *successor*, 60 naik dan anak kanannya 65 menempati posisi lamanya (kasus 2); (c) dengan *predecessor*, 45 naik dan sebuah daun hilang (kasus 1).

- **Varian *successor*.** Minimum subpohon kanan 50 ditemukan dengan belok ke kanan menuju 70 lalu ke kiri menuju 60. Kali ini *successor*-nya bukan anak 50 tetapi dua tingkat di bawahnya, dan ia memiliki anak kanan, 65. Kuncinya menggantikan 50, dan menghapus 60 yang lama termasuk kasus 2: 65 naik ke posisi 60 sebagai anak kiri 70.
- **Varian *predecessor*.** Maksimum subpohon kiri 50 ditemukan dengan belok ke kiri menuju 30 lalu ke kanan menuju 40 dan 45. 45 menggantikan 50, dan menghapus 45 yang lama, sebuah daun, termasuk kasus 1.

Kedua hasil adalah *binary search tree* yang benar dengan kunci yang sama; barisan *inorder* keduanya sama. Varian mana yang sebaiknya dipakai program? Untuk satu penghapusan, tidak

ada bedanya. Untuk banyak penghapusan, bisa ada bedanya: program yang selalu memakai *successor* terus mengambil simpul dari subpohon kanan, dan eksperimen Eppinger [9] memperlihatkan bahwa barisan acak panjang berisi penyisipan dan penghapusan yang hanya memakai *successor* membuat pohon terukur lebih tinggi daripada pohon yang dibangun dengan penyisipan saja. Obat sederhana yang banyak dipakai adalah bergantian di antara kedua varian, atau memilih secara acak.

11.7.12 Program penghapusan

Kode 11.7 mengimplementasikan *successor* dan *predecessor* tanpa pointer induk (**Pseudokode 11.9** dan bayangan cerminnya) serta `deleteNode` dengan sebuah penanda yang memilih variannya. Fungsi `main`-nya memutar ulang **Gambar 11.19** pada contoh berjalan, menanyakan *successor* dan *predecessor* pada pohon bersimpul sepuluh di **Gambar 11.18**, menghapus akar pohon itu dengan setiap varian seperti pada **Gambar 11.20**, dan akhirnya mencoba menghapus kunci yang tidak ada.

Kode 11.7 `bst_delete.cpp`: *successor* dan *predecessor* tanpa pointer induk, serta penghapusan dalam ketiga kasus dengan varian *successor* dan varian *predecessor*.

```cpp
// bst_delete.cpp -- successor, predecessor, and deletion from a binary search
// tree in all three cases, with the successor variant and the predecessor variant.
#include <cstdio>

struct Node {
    int data;
    Node *left, *right;
};

Node *insert(Node *root, int x) {           // returns the new subtree root
    if (root == nullptr) return new Node{x, nullptr, nullptr};
    if (x < root->data) root->left = insert(root->left, x);
    else if (x > root->data) root->right = insert(root->right, x);
    return root;                            // a duplicate changes nothing
}

Node *minNode(Node *t) {
    while (t != nullptr && t->left != nullptr) t = t->left;
    return t;
}

Node *maxNode(Node *t) {
    while (t != nullptr && t->right != nullptr) t = t->right;
    return t;
}

// The node holding the smallest key larger than x (x must be in the tree).
Node *successor(Node *root, int x) {
    Node *cand = nullptr;                   // last ancestor where we went left
    Node *cur = root;
    while (cur != nullptr && cur->data != x) {
        if (x < cur->data) {
            cand = cur;
            cur = cur->left;
        } else {
            cur = cur->right;
        }
    }
    if (cur == nullptr) return nullptr;     // x is not in the tree
    if (cur->right != nullptr) return minNode(cur->right);
    return cand;
}
```

Kode 11.7 (Lanjutan) *bst_delete.cpp: successor dan predecessor tanpa pointer induk, serta penghapusan dalam ketiga kasus dengan varian successor dan varian predecessor.*

```
// The node holding the largest key smaller than x (x must be in the tree).
Node *predecessor(Node *root, int x) {
    Node *cand = nullptr;                   // last ancestor where we went right
    Node *cur = root;
    while (cur != nullptr && cur->data != x) {
        if (x > cur->data) {
            cand = cur;
            cur = cur->right;
        } else {
            cur = cur->left;
        }
    }
    if (cur == nullptr) return nullptr;
    if (cur->left != nullptr) return maxNode(cur->left);
    return cand;
}

// Deletes x and returns the new root of this subtree. With useSucc the
// two-children case borrows the successor, otherwise the predecessor.
Node *deleteNode(Node *root, int x, bool useSucc) {
    if (root == nullptr) {
        std::printf("  %d is not in the tree\n", x);
        return nullptr;
    }
    if (x < root->data) {
        root->left = deleteNode(root->left, x, useSucc);
    } else if (x > root->data) {
        root->right = deleteNode(root->right, x, useSucc);
    } else if (root->left == nullptr && root->right == nullptr) {
        std::printf("  case 1: %d is a leaf, unlink it\n", x);
        delete root;
        return nullptr;
    } else if (root->left == nullptr || root->right == nullptr) {
        Node *child = (root->left != nullptr) ? root->left : root->right;
        std::printf("  case 2: %d has one child, %d moves up\n", x, child->data);
        delete root;
        return child;
    } else if (useSucc) {
        Node *s = minNode(root->right);     // smallest key of the right subtree
        std::printf("  case 3: %d has two children, successor %d replaces it\n",
                    x, s->data);
        root->data = s->data;
        root->right = deleteNode(root->right, s->data, useSucc);
    } else {
        Node *p = maxNode(root->left);      // largest key of the left subtree
        std::printf("  case 3: %d has two children, predecessor %d replaces it\n",
```

Kode 11.7 (Lanjutan) *bst_delete.cpp: successor dan predecessor tanpa pointer induk, serta penghapusan dalam ketiga kasus dengan varian successor dan varian predecessor.*

```
                    x, p->data);
        root->data = p->data;
        root->left = deleteNode(root->left, p->data, useSucc);
    }
    return root;
}

void printSideways(const Node *t, int depth) {
    if (t == nullptr) return;
    printSideways(t->right, depth + 1);
    std::printf("%*s%d\n", 4 * depth + 4, "", t->data);
    printSideways(t->left, depth + 1);
}

void inOrder(const Node *t) {
    if (t == nullptr) return;
    inOrder(t->left);
    std::printf("%d ", t->data);
    inOrder(t->right);
}

void destroy(Node *t) {
    if (t == nullptr) return;
    destroy(t->left);
    destroy(t->right);
    delete t;
}

Node *build(const int *keys, int n) {
    Node *root = nullptr;
    for (int i = 0; i < n; ++i) root = insert(root, keys[i]);
    return root;
}

void show(const char *title, const Node *t) {
    std::printf("%s  inorder: ", title);
    inOrder(t);
    std::printf("\n");
    printSideways(t, 0);
}
```

Kode 11.7 (Lanjutan) *bst_delete.cpp: successor dan predecessor tanpa pointer induk, serta penghapusan dalam ketiga kasus dengan varian successor dan varian predecessor.*

```
int main() {
    int a[] = {65, 5, 70, 3, 10};
    Node *t = build(a, 5);
    show("Tree A:", t);
    int dels[] = {10, 5, 65};
    for (int x : dels) {
        std::printf("delete %d\n", x);
        t = deleteNode(t, x, true);
        show("after:", t);
    }
    destroy(t);

    int b[] = {50, 30, 70, 20, 40, 60, 80, 35, 45, 65};
    t = build(b, 10);
    show("\nTree B:", t);
    int qs[] = {45, 50, 35, 65, 80, 20};
    for (int x : qs) {
        Node *s = successor(t, x), *p = predecessor(t, x);
        std::printf("  key %2d: predecessor ", x);
        if (p != nullptr) std::printf("%2d", p->data); else std::printf("--");
        std::printf(", successor ");
        if (s != nullptr) std::printf("%2d\n", s->data); else std::printf("--\n");
    }
    std::printf("delete 50 (successor variant)\n");
    t = deleteNode(t, 50, true);
    show("after:", t);
    destroy(t);

    t = build(b, 10);
    std::printf("\nTree B again; delete 50 (predecessor variant)\n");
    t = deleteNode(t, 50, false);
    show("after:", t);
    std::printf("delete 99\n");
    t = deleteNode(t, 99, false);
    destroy(t);
    return 0;
}
```

```
Tree A:  inorder: 3 5 10 65 70
        70
    65
            10
        5
            3
delete 10
  case 1: 10 is a leaf, unlink it
after:  inorder: 3 5 65 70
        70
    65
        5
            3
delete 5
  case 2: 5 has one child, 3 moves up
after:  inorder: 3 65 70
        70
    65
        3
delete 65
  case 3: 65 has two children, successor 70 replaces it
  case 1: 70 is a leaf, unlink it
after:  inorder: 3 70
    70
        3

Tree B:  inorder: 20 30 35 40 45 50 60 65 70 80
            80
        70
                65
            60
    50
                45
            40
                35
        30
            20
  key 45: predecessor 40, successor 50
  key 50: predecessor 45, successor 60
  key 35: predecessor 30, successor 40
  key 65: predecessor 60, successor 70
  key 80: predecessor 70, successor --
  key 20: predecessor --, successor 30
delete 50 (successor variant)
  case 3: 50 has two children, successor 60 replaces it
  case 2: 60 has one child, 65 moves up
after:  inorder: 20 30 35 40 45 60 65 70 80
            80
        70
            65
    60
                45
```

```
            40
                35
        30
            20

Tree B again; delete 50 (predecessor variant)
  case 3: 50 has two children, predecessor 45 replaces it
  case 1: 45 is a leaf, unlink it
after:  inorder: 20 30 35 40 45 60 65 70 80
            80
        70
                65
            60
    45
            40
                35
        30
            20
delete 99
  99 is not in the tree
```

Baris-baris jejak menyebutkan kasus yang berlaku. Pada pohon pertama, menghapus 10, lalu 5, lalu 65 memicu kasus 1, 2, dan 3 dalam urutan itu, dan kasus 3 diakhiri dengan penghapusan kasus 1 atas 70 yang lama, persis seperti pada **Gambar 11.19**. Pada pohon kedua, keenam kueri mencakup setiap situasi: 45 dan 35 tidak memiliki subpohon kanan (kasus 2 pada **Pseudokode 11.9**), 50 dan 65 memilikinya, 80 tidak memiliki *successor*, dan 20 tidak memiliki *predecessor*. Kedua penghapusan 50 menghasilkan pohon **Gambar 11.20** (b) dan (c), dan baris terakhir memperlihatkan bahwa menghapus kunci yang tidak ada akan jatuh keluar dari pohon tanpa mengubah apa pun. Di bawah Valgrind program ini berakhir dengan semua blok *heap* terlepas dan tanpa galat; versi Java dan Python (`BstDelete.java`, `bst_delete.py`) mencetak keluaran yang sama.

11.7.13 Ongkos

Setiap operasi BST selain penelusuran mengikuti satu lintasan turun dari akar, mungkin disusul satu lintasan turun lagi (minimum sebuah subpohon pada penghapusan kasus 3), dan melakukan kerja konstan per simpul pada lintasan itu. Jadi:

$$T_{\text{cari}},\ T_{\text{sisip}},\ T_{\text{hapus}},\ T_{\text{min}},\ T_{\text{max}},\ T_{\text{succ}},\ T_{\text{pred}} \ = \ O(h) \tag{11.10}$$

dengan h tinggi pohon. **Tabel 11.3** memadukan Persamaan (11.10) dengan kedua tinggi ekstrem dari Subbab 11.5.3.

Tabel 11.3 Ongkos operasi-operasi *binary search tree* dengan n simpul dan tinggi h.

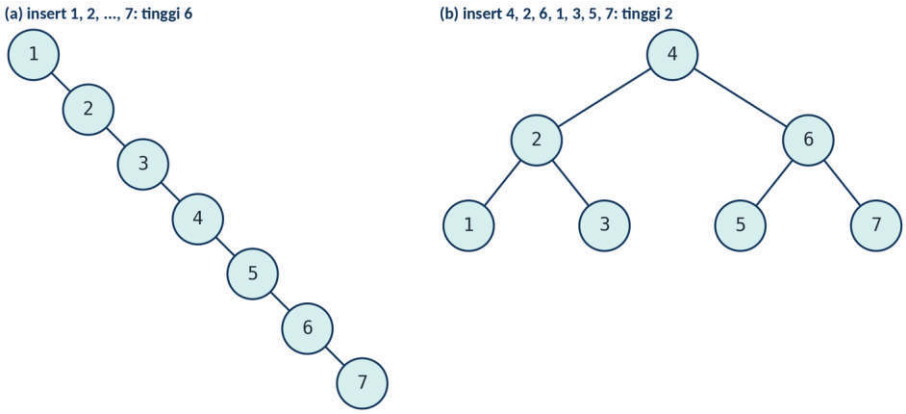
Operasi	Ongkos	Pohon seimbang, $h=\Theta(\lg n)$	Pohon merosot, $h=n-1$
cari, sisip, hapus	$O(h)$ (**Pseudokode 11.6**, **Pseudokode 11.5**, **Pseudokode 11.10**)	$O(\lg n)$	$O(n)$
minimum, maksimum	$O(h)$ (**Pseudokode 11.7**)	$O(\lg n)$	$O(n)$
successor, *predecessor*	$O(h)$ (**Pseudokode 11.9**)	$O(\lg n)$	$O(n)$
hitung simpul, daun, tinggi	$\Theta(n)$ (Persamaan (11.9))	$\Theta(n)$	$\Theta(n)$
penelusuran apa pun	waktu $\Theta(n)$ (Persamaan (11.8)), *stack* $O(h)$	$\Theta(n)$	$\Theta(n)$
membangun dari n kunci	n penyisipan	$O(n \lg n)$	$\Theta(n^2)$

Tabel itu menaruh seluruh persoalan dalam satu kolom: BST sebaik *binary search* bila tingginya logaritmik, dan tidak lebih baik daripada *linked list* pada Bab 7 bila tingginya linear. Tinggi itu ditentukan oleh urutan kedatangan kunci-kunci. Subbab berikutnya mengukurnya.

11.8 Pohon Seimbang dan Pohon Merosot

11.8.1 Urutan penyisipan menentukan bentuk

Sisipkan tujuh kunci 1, 2, …, 7 ke BST kosong dalam urutan naik. Setiap kunci baru lebih besar daripada setiap kunci yang sudah ada di pohon, sehingga setiap penyisipan berjalan ke kanan sampai ke bawah dan menggantungkan simpul baru sebagai anak kanan simpul sebelumnya. Hasilnya, **Gambar 11.21** (a), adalah rantai condong ke kanan bertinggi 6: sebuah *linked list* yang digambar miring, dan setiap pencarian di dalamnya adalah pencarian sekuensial dari Bab 2. Sisipkan tujuh kunci yang sama dalam urutan 4, 2, 6, 1, 3, 5, 7 dan Anda memperoleh pohon *perfect* pada **Gambar 11.21** (b), bertinggi 2, yang terendah yang mungkin menurut Persamaan (11.6).



Gambar 11.21 Tujuh kunci yang sama dalam dua urutan penyisipan: (a) dalam urutan terurut kunci-kunci itu membentuk rantai bertinggi 6; (b) bila yang tengah disisipkan lebih dahulu, terbentuk pohon *perfect* bertinggi 2.

Masukan terurut adalah kasus terburuk, dan dalam praktik justru kasus yang *lazim*: rekaman datang dalam urutan waktu pembuatannya, kode dibagikan dalam urutan naik, sebuah berkas sering kali sudah terurut. Berapa banyak dari $n!$ urutan penyisipan yang menghasilkan rantai? Kunci pertama harus kunci terkecil atau terbesar di antara yang tersisa (bila tidak, ia memiliki kunci di kedua sisinya dan pohon akan bercabang di akar), dan hal yang sama berlaku untuk setiap kunci berikutnya di antara kunci-kunci yang masih akan datang, sampai tinggal satu kunci. Itu berarti dua pilihan untuk setiap kunci dari $n-1$ kunci pertama:

$$\#\{\text{urutan bertinggi } n-1\} \;=\; 2^{n-1} \tag{11.11}$$

Untuk $n = 7$ itu 64 dari $7! = 5040$ urutan, sekitar 1,3%. Untuk urutan acak, pohon merosot itu jarang. Pertanyaannya adalah seperti apa pohon acak yang *khas*, dan jawabannya: baik.

11.8.2 Tinggi yang terukur

Teori *binary search tree* acak menyatakan bahwa bila semua $n!$ urutan penyisipan sama mungkinnya, kedalaman rata-rata sebuah simpul adalah

$$\bar{d}(n) \;=\; 2\left(1+\frac{1}{n}\right)H_n - 4 \;\approx\; 2\ln n \;\approx\; 1.39\lg n \tag{11.12}$$

dengan $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n + 0.577$ bilangan harmonik ke-n [8]. Jadi BST acak rata-rata hanya sekitar 39% lebih dalam daripada pohon terbaik yang mungkin. *Tinggi*-nya, kedalaman simpul terdalamnya, lebih besar, sekitar $4.311\ln n$ untuk n besar [10], dan itu tetap logaritmik.

Kode 11.8 memeriksa kedua klaim itu dan mengukur kasus terburuknya. Bagian 1 menyisipkan kunci $0, 1, \ldots, n-1$ dalam urutan acak (benih yang tetap membuat jalannya dapat diulang) untuk n dari 2^{10} sampai 2^{20} dan melaporkan tinggi, kedalaman rata-rata, nilai Persamaan (11.12), dan waktunya. Bagian 2 menyisipkan kunci yang sama dalam urutan terurut dan dalam urutan acak lalu membandingkannya. Penyisipannya iteratif, dengan teknik pointer-ke-tautan dari Subbab 7.7.3, dan tinggi diukur tingkat demi tingkat, sehingga rantai bersimpul 32.000 tidak dapat meluapkan *call stack*.

Kode 11.8 `height_experiment.cpp`: tinggi dan kedalaman rata-rata *binary search tree* yang dibangun dari urutan penyisipan acak dan terurut, beserta waktu pembangunannya.

```
// height_experiment.cpp -- how tall does a binary search tree grow? Keys inserted
// in random order against keys inserted in sorted order, measured.
#include <algorithm>
#include <chrono>
#include <cmath>
#include <cstdio>
#include <numeric>
#include <random>
#include <vector>

struct Node {
    int key;
    Node *left, *right;
};

void insert(Node *&root, int key) {        // iterative: no recursion depth limit
    Node **link = &root;
    while (*link != nullptr)
        link = (key < (*link)->key) ? &(*link)->left : &(*link)->right;
    *link = new Node{key, nullptr, nullptr};
}

// Height and total depth by level-order, with no recursion.
void measure(Node *root, int &height, double &totalDepth) {
    std::vector<Node *> level{root}, next;
    height = -1;
    totalDepth = 0;
    while (!level.empty()) {
        ++height;
        totalDepth += (double) height * level.size();
        next.clear();
        for (Node *n : level) {
            if (n->left) next.push_back(n->left);
            if (n->right) next.push_back(n->right);
        }
        level.swap(next);
    }
}
```

Kode 11.8 (Lanjutan) *height_experiment.cpp: tinggi dan kedalaman rata-rata binary search tree yang dibangun dari urutan penyisipan acak dan terurut, beserta waktu pembangunannya.*

```
void destroy(Node *root) {                  // iterative, with an explicit stack
    std::vector<Node *> st;
    if (root) st.push_back(root);
    while (!st.empty()) {
        Node *n = st.back();
        st.pop_back();
        if (n->left) st.push_back(n->left);
        if (n->right) st.push_back(n->right);
        delete n;
    }
}

double buildMs(const std::vector<int> &keys, int &h, double &avg) {
    auto t0 = std::chrono::steady_clock::now();
    Node *root = nullptr;
    for (int k : keys) insert(root, k);
    auto t1 = std::chrono::steady_clock::now();
    double total;
    measure(root, h, total);
    avg = total / keys.size();
    destroy(root);
    return std::chrono::duration<double, std::milli>(t1 - t0).count();
}

int main() {
    std::mt19937 rng(2026);
    std::printf("Part 1: keys 0..n-1 inserted in random order\n");
    std::printf("%9s %7s %8s %10s %10s %9s\n",
                "n", "floor", "height", "avg depth", "expected", "build ms");
    std::printf("%9s %7s %8s %10s %10s\n", "", "lg n", "", "", "avg depth");
    for (int n = 1024; n <= (1 << 20); n *= 4) {
        std::vector<int> keys(n);
        std::iota(keys.begin(), keys.end(), 0);
        std::shuffle(keys.begin(), keys.end(), rng);
        int h;
        double avg;
        double ms = buildMs(keys, h, avg);
        double H = 0;                       // harmonic number H_n
        for (int i = 1; i <= n; ++i) H += 1.0 / i;
        double expected = 2.0 * (1.0 + 1.0 / n) * H - 4.0;
        std::printf("%9d %7d %8d %10.2f %10.2f %9.1f\n",
                    n, (int) std::floor(std::log2(n)), h, avg, expected, ms);
    }
```

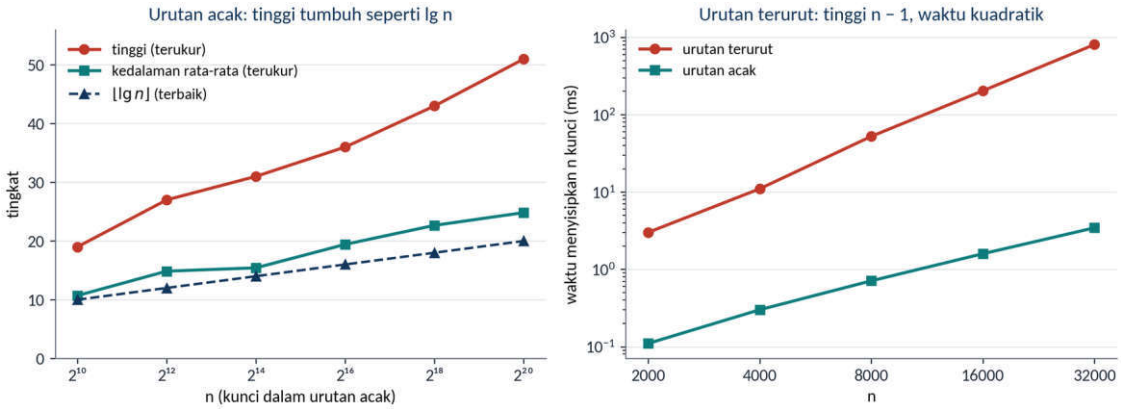
Kode 11.8 (Lanjutan) *height_experiment.cpp: tinggi dan kedalaman rata-rata binary search tree yang dibangun dari urutan penyisipan acak dan terurut, beserta waktu pembangunannya.*

```
    std::printf("\nPart 2: the same keys in sorted order against random order\n");
    std::printf("%9s %14s %12s %14s %12s\n",
                "n", "sorted height", "sorted ms", "random height", "random ms");
    for (int n = 2000; n <= 32000; n *= 2) {
        std::vector<int> keys(n);
        std::iota(keys.begin(), keys.end(), 0);
        int hs, hr;
        double avg;
        double ms = buildMs(keys, hs, avg);
        std::shuffle(keys.begin(), keys.end(), rng);
        double mr = buildMs(keys, hr, avg);
        std::printf("%9d %14d %12.1f %14d %12.2f\n", n, hs, ms, hr, mr);
    }
    return 0;
}
```

```
Part 1: keys 0..n-1 inserted in random order
        n   floor   height  avg depth   expected  build ms
             lg n                      avg depth
     1024      10       19      10.70      11.03       0.1
     4096      12       27      14.86      13.79       0.5
    16384      14       31      15.42      16.56       2.1
    65536      16       36      19.42      19.34      17.0
   262144      18       43      22.66      22.11      83.2
  1048576      20       51      24.83      24.88     530.1

Part 2: the same keys in sorted order against random order
        n  sorted height    sorted ms  random height    random ms
     2000           1999          3.0             23         0.11
     4000           3999         11.0             27         0.30
     8000           7999         52.4             31         0.71
    16000          15999        203.6             32         1.59
    32000          31999        806.0             35         3.44
```

Gambar 11.22 memplot jalannya program itu. Waktu-waktunya bergantung pada mesin (jalannya ini memakai server bersama Intel Xeon 2,1 GHz); tingginya tidak, karena benih itu menetapkan urutan penyisipan.



Gambar 11.22 Hasil pengukuran **Kode 11.8**: dalam urutan acak, tinggi dan kedalaman rata-rata tumbuh seiring lg$n$ (kiri); dalam urutan terurut pohon menjadi rantai, dan waktu membangunnya menjadi empat kali lipat pada setiap penggandaan n (kanan, skala logaritmik).

- **Urutan acak itu logaritmik.** Setiap kali n dikalikan 4, lg$n$ bertambah 2. Sepanjang lima langkah dari 2^{10} sampai 2^{20}, lg$n$ bertambah 10 dan kedalaman rata-rata yang terukur bertambah $24.83 - 10.70 = 14.13$, rasio sekitar 1,39 yang diramalkan Persamaan (11.12). Langkah-langkah tunggal bervariasi, karena rumus itu adalah rata-rata atas semua urutan penyisipan sedangkan setiap jalannya program hanyalah satu urutan, tetapi setiap nilai terukur berada dalam jarak sekitar satu tingkat dari rumusnya. Tingginya tumbuh lebih cepat daripada lg$n$ tetapi tetap kecil: sejuta kunci menghasilkan pohon bertinggi 51, dibandingkan dengan minimum yang mungkin, 19, dan tinggi rantai, 1.048.575. Pencarian apa pun di dalamnya memerlukan paling banyak 52 perbandingan.
- **Urutan terurut itu linear.** Tingginya tepat $n - 1$ setiap kali, sesuai ramalan. Membangun pohonnya memerlukan $0 + 1 + 2 + \cdots + (n-1) = n(n-1)/2$ langkah, $\Theta(n^2)$, dan waktunya memperlihatkan hal itu: setiap penggandaan n mengalikan waktunya sekitar 4 kali (3,0; 11,0; 52,4; 203,6; 806 ms). Pada $n = 32{,}000$ pembangunan terurut sekitar 230 kali lebih lambat daripada pembangunan acak, yang memerlukan 3,44 ms, dan selisihnya tumbuh tanpa batas.

Versi Java dan Python dalam folder kode (`HeightExperiment.java`, `height_experiment.py`) memperlihatkan bentuk yang sama dengan waktu masing-masing. Dalam satu jalannya Python, pembangunan terurut 32.000 kunci memerlukan sekitar 16 detik berbanding 50 ms untuk pembangunan acak; di Java, 1,4 detik berbanding 4 ms.

11.8.3 Arti "seimbang"

Sebuah keluarga pohon disebut **seimbang** (*balanced*) bila tingginya $O(\lg n)$. Beberapa syarat yang tepat menjaminnya; yang paling terkenal adalah syarat yang dijaga pohon AVL:

> **DEFINISI.** Sebuah pohon biner disebut *seimbang-tinggi* (*height-balanced*) bila, di setiap simpul, tinggi subpohon kiri dan kanan berselisih paling banyak 1. Selisih $height(x.left) - height(x.right)$ disebut *faktor keseimbangan* (*balance factor*) x; pada pohon seimbang-tinggi nilainya -1, 0, atau $+1$ di mana-mana.

Setiap pohon *perfect* dan setiap pohon *complete* seimbang-tinggi; rantai pada **Gambar 11.21** (a) tidak (di simpul 1 subpohon kirinya bertinggi -1 dan subpohon kanannya bertinggi 5). Bab 12 menulis fungsi yang memeriksa syarat ini dengan menghitung tinggi menurut Persamaan (11.9). Subbab 11.9.2 memperlihatkan mengapa syarat itu memaksa tinggi yang logaritmik.

Ada tiga cara untuk hidup berdampingan dengan kenyataan bahwa bentuk BST biasa bergantung pada masukannya.

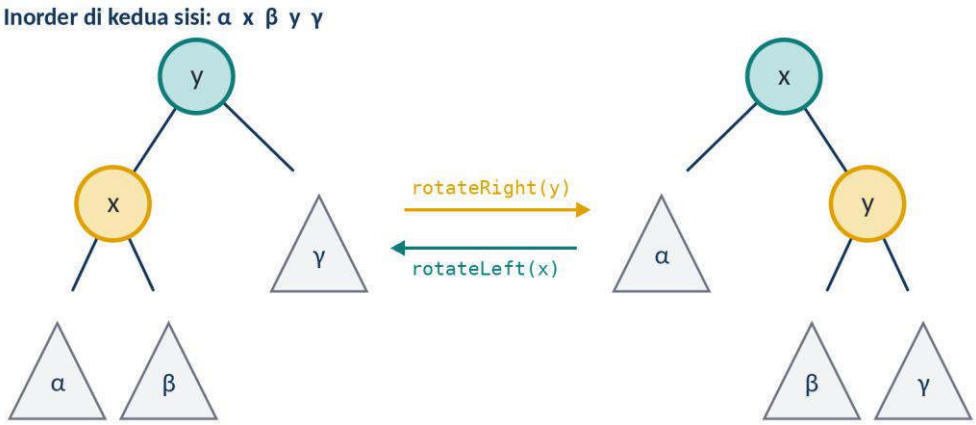
1. **Berharap masukannya acak.** Sering kali masuk akal, dan **Kode 11.8** memperlihatkan betapa baiknya urutan acak. Mengocok sekelompok kunci sebelum menyisipkannya membuat harapan itu menjadi kenyataan. Tetapi program tidak dapat mengocok kunci yang datang satu per satu, dan pihak yang berniat buruk dapat mengirim kunci terurut dengan sengaja.
2. **Membangun ulang bila pohonnya sudah diketahui.** Bila semua kunci tersedia sekaligus, urutkan lalu bangun pohon dari tengah ke luar: kunci tengah menjadi akar, dan kedua separuhnya dibangun dengan cara yang sama, secara rekursif. Hasilnya memiliki tinggi minimum $\lfloor \lg n \rfloor$ dan pembangunannya memerlukan $O(n)$ sesudah pengurutan; Soal 11.10 menuliskannya.
3. **Menyeimbangkan ulang pada setiap perubahan.** Jadikan setiap penyisipan dan penghapusan memperbaiki bentuk secara lokal sehingga tingginya tetap $O(\lg n)$ apa pun masukannya. Itulah yang dilakukan pohon penyeimbang-diri, dan itulah yang dilakukan *map* terurut di setiap pustaka baku.

11.9 Sekilas tentang Pohon Penyeimbang-Diri

Subbab ini adalah pratinjau: ia memperlihatkan satu operasi yang menjadi dasar semua pohon penyeimbang-diri, menjalankan salah satunya pada masukan yang merusak BST biasa, dan menyebutkan dua keluarga yang dipakai pustaka-pustaka baku. Algoritme lengkapnya termasuk kajian algoritme yang lebih lanjut; Cormen dkk. [2] menyajikannya secara rinci.

11.9.1 Rotasi

Sebuah *rotasi* mengubah bentuk bagian kecil sebuah BST dalam waktu $O(1)$ tanpa merusak sifat BST. **Gambar 11.23** memperlihatkan kedua arahnya. Pada gambar kiri, x adalah anak kiri y; α dan β adalah subpohon x dan γ adalah subpohon kanan y. *Rotasi kanan di y* menjadikan x akar bagian ini, dengan y sebagai anak kanannya; β, yang dalam urutan kunci berada di antara x dan y, berpindah menjadi subpohon kiri y. *Rotasi kiri di x* membatalkannya.

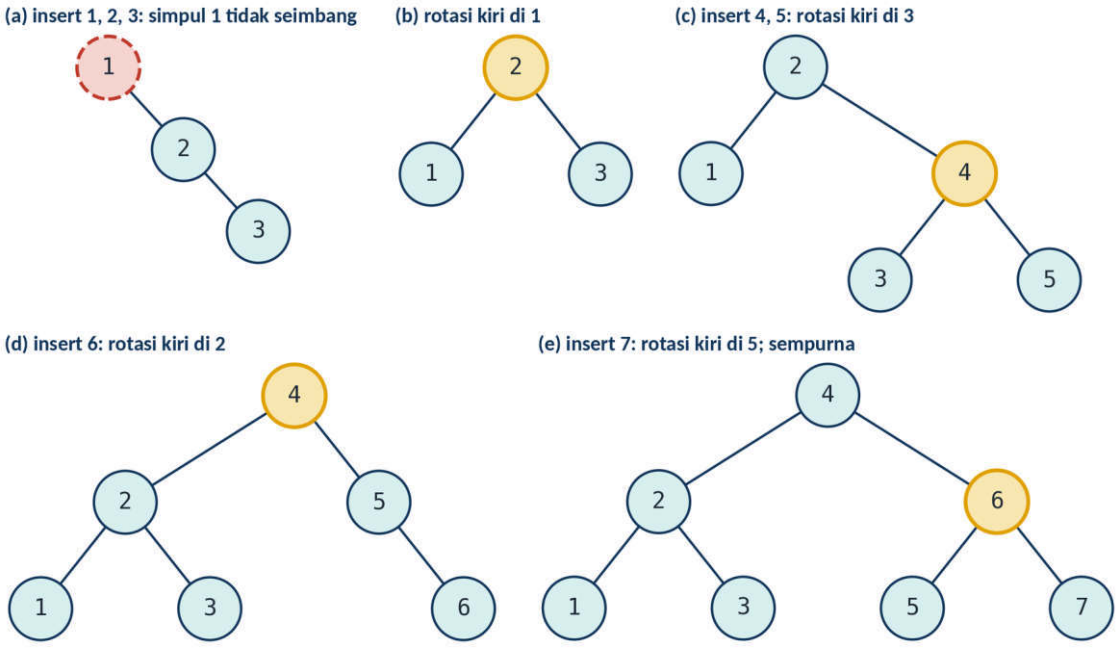


Gambar 11.23 Rotasi kanan dan rotasi kiri: tiga pointer berubah, subpohon di satu sisi turun satu tingkat dan di sisi lain naik satu tingkat, dan barisan *inorder* $\alpha, x, \beta, y, \gamma$ tidak berubah.

Rotasi mempertahankan sifat BST karena ia mempertahankan barisan *inorder*: pada kedua sisi barisannya $\alpha, x, \beta, y, \gamma$. Dan ia hanya mengubah tiga pointer (`y->left`, `x->right`, dan pointer dari atas yang tadinya menunjuk y dan kini menunjuk x), sehingga ongkosnya $O(1)$. Yang diubahnya adalah tinggi: pada rotasi kanan, segala sesuatu di α naik satu tingkat dan segala sesuatu di γ turun satu tingkat. Pohon penyeimbang-diri mendeteksi, sesudah penyisipan atau penghapusan, tempat di mana satu sisi menjadi terlalu dalam, lalu merotasi di sana.

11.9.2 Pohon AVL

BST penyeimbang-diri yang pertama diterbitkan pada 1962 oleh Georgy Adelson-Velsky dan Evgenii Landis [5], dan menyandang inisial mereka. **Pohon AVL** menjaga setiap simpul tetap seimbang-tinggi (Subbab 11.8.3). Setiap simpul menyimpan tingginya; sesudah penyisipan turun dan menambahkan daun, rekursinya berjalan naik kembali, menghitung ulang tinggi, dan di mana pun faktor keseimbangan menjadi $+2$ atau -2 ia memperbaiki simpul itu dengan satu rotasi atau, bila bagian yang dalam berada di sisi dalam, dengan dua rotasi (pertama di anaknya, lalu di simpul itu). **Gambar 11.24** menyisipkan 1, 2, …, 7 dalam urutan terurut, masukan yang menghasilkan **Gambar 11.21** (a).



Gambar 11.24 Menyisipkan 1 sampai 7 dalam urutan terurut ke pohon AVL: (a) sesudah 3, simpul 1 berfaktor keseimbangan −2; (b) rotasi kiri di 1 memperbaikinya; (c) sampai (e) tiga rotasi kiri lagi menjaga pohon tetap seimbang, dan hasilnya *perfect*.

Mengapa seimbang-tinggi berakibat tinggi yang logaritmik? Misalkan $N(h)$ adalah banyak simpul *paling sedikit* yang dapat dimiliki pohon AVL bertinggi h. Pohon seperti itu memiliki akar dan dua subpohon, satu bertinggi $h-1$ dan, agar simpulnya sesedikit mungkin, yang lain bertinggi $h-2$, keduanya sendiri sejarang mungkin: $N(h) = N(h-1) + N(h-2) + 1$, dengan $N(0) = 1$ dan $N(1) = 2$. Inilah rekurens Fibonacci dari Subbab 10.5, ditambah satu, dan memang $N(h) = F_{h+3} - 1$. Bilangan Fibonacci tumbuh seperti φ^h dengan $\varphi \approx 1.618$, sehingga $n \geq N(h)$ memaksa h paling banyak sekitar $\log_{\varphi} n$. Batas tepatnya [8] adalah

$$h < 1.4405 \lg(n+2) - 0.3277 \qquad (11.13)$$

Pohon AVL tidak pernah lebih dari 44% lebih tinggi daripada pohon terbaik yang mungkin, apa pun urutan kedatangan kuncinya. **Kode 11.9** mengimplementasikan penyisipan AVL (penghapusan serupa tetapi lebih panjang) dan menyuapinya kunci terurut.

Kode 11.9 `avl_first_look.cpp`: penyisipan AVL dengan rotasi tunggal dan ganda, disuapi kunci terurut yang mengubah *binary search tree* biasa menjadi rantai.

```cpp
// avl_first_look.cpp -- a first look at a self-balancing tree: AVL insertion
// with rotations, fed the sorted keys that ruin a plain binary search tree.
#include <cmath>
#include <cstdio>

struct Node {
    int key;
    int h;                  // height of the subtree rooted here
    Node *left, *right;
};

bool verbose = true;
int ht(const Node *t) { return t == nullptr ? -1 : t->h; }
void fix(Node *t) {            // recompute the height from the children
    int a = ht(t->left), b = ht(t->right);
    t->h = 1 + (a > b ? a : b);
}
int balance(const Node *t) { return ht(t->left) - ht(t->right); }

Node *rotateRight(Node *y) {    // y's left child x becomes the subtree root
    Node *x = y->left;
    y->left = x->right;
    x->right = y;
    fix(y);
    fix(x);
    return x;
}

Node *rotateLeft(Node *x) {     // x's right child y becomes the subtree root
    Node *y = x->right;
    x->right = y->left;
    y->left = x;
    fix(x);
    fix(y);
    return y;
}
```

Kode 11.9 (Lanjutan) *avl_first_look.cpp: penyisipan AVL dengan rotasi tunggal dan ganda, disuapi kunci terurut yang mengubah binary search tree biasa menjadi rantai.*

```
Node *insert(Node *t, int key) {
    if (t == nullptr) return new Node{key, 0, nullptr, nullptr};
    if (key < t->key) t->left = insert(t->left, key);
    else if (key > t->key) t->right = insert(t->right, key);
    else return t;
    fix(t);
    int b = balance(t);
    if (b > 1 && key > t->left->key) t->left = rotateLeft(t->left);    // left-right
    if (b < -1 && key < t->right->key) t->right = rotateRight(t->right);
    if (b > 1) {
        if (verbose)
            std::printf("  insert %d: right rotation at %d\n", key, t->key);
        return rotateRight(t);
    }
    if (b < -1) {
        if (verbose) std::printf("  insert %d: left rotation at %d\n", key, t->key);
        return rotateLeft(t);
    }
    return t;
}

void printSideways(const Node *t, int depth) {
    if (t == nullptr) return;
    printSideways(t->right, depth + 1);
    std::printf("%*s%d\n", 4 * depth + 4, "", t->key);
    printSideways(t->left, depth + 1);
}

void destroy(Node *t) {
    if (t == nullptr) return;
    destroy(t->left);
    destroy(t->right);
    delete t;
}

int main() {
    Node *root = nullptr;
    std::printf("AVL tree, keys 1..7 inserted in sorted order:\n");
    for (int k = 1; k <= 7; ++k) root = insert(root, k);
    printSideways(root, 0);
    std::printf("height %d (a plain BST would have height 6)\n\n", ht(root));
    destroy(root);
```

Kode 11.9 (Lanjutan) *avl_first_look.cpp: penyisipan AVL dengan rotasi tunggal dan ganda, disuapi kunci terurut yang mengubah binary search tree biasa menjadi rantai.*

```
    verbose = false;
    std::printf("%9s %12s %12s %14s\n", "n", "AVL height", "plain BST",
                "1.44 lg(n+2)");
    for (int n = 1000; n <= 1000000; n *= 10) {
        root = nullptr;
        for (int k = 1; k <= n; ++k) root = insert(root, k);
        std::printf("%9d %12d %12d %14.1f\n", n, ht(root), n - 1,
                    1.44 * std::log2(n + 2.0));
        destroy(root);
    }
    return 0;
}
```

```
AVL tree, keys 1..7 inserted in sorted order:
  insert 3: left rotation at 1
  insert 5: left rotation at 3
  insert 6: left rotation at 2
  insert 7: left rotation at 5
            7
        6
            5
    4
            3
        2
            1
height 2 (a plain BST would have height 6)

        n   AVL height    plain BST   1.44 lg(n+2)
     1000            9          999           14.4
    10000           13         9999           19.1
   100000           16        99999           23.9
  1000000           19       999999           28.7
```

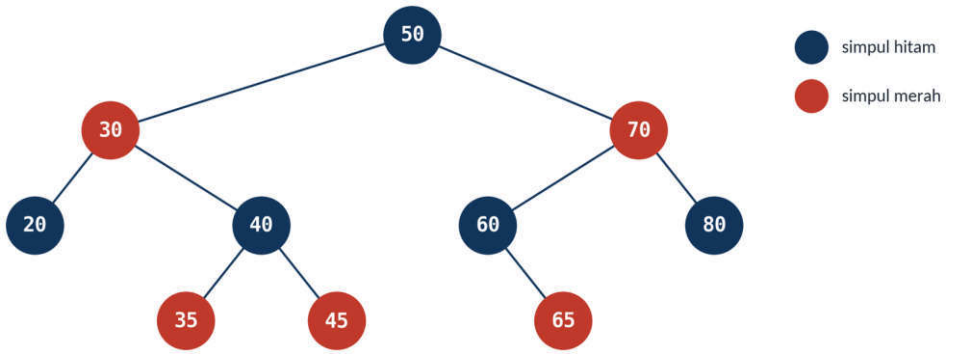
Rotasi-rotasi yang dicetak untuk 1 sampai 7 adalah rotasi pada **Gambar 11.24**, dan pohon yang dihasilkan *perfect*. Tabelnya membandingkan, untuk masukan terurut, tinggi pohon AVL dengan tinggi BST biasa $n-1$ (diukur pada **Kode 11.8**) dan dengan batas Persamaan (11.13). Dengan sejuta kunci terurut pohon AVL bertinggi 19, yang sama dengan $\lfloor \lg n \rfloor$: masukan terurut, kasus terburuk bagi BST biasa, kebetulan menjadi kasus mudah bagi pohon AVL. (Versi Java dan Python, `AvlFirstLook.java` dan `avl_first_look.py`, mencetak hal yang sama.)

11.9.3 Pohon red-black

Keluarga besar yang lain adalah **pohon *red-black***. Rudolf Bayer memperkenalkan gagasannya pada 1972 sebagai "*symmetric binary B-tree*" [6], dan Leonidas Guibas serta Robert Sedgewick memberinya warna dan namanya pada 1978 [7]. Setiap simpul membawa satu bit tambahan, warnanya, dan pohon itu mematuhi lima aturan [2]:

1. Setiap simpul berwarna merah atau hitam.
2. Akar berwarna hitam.
3. Setiap daun (pointer NIL) berwarna hitam.
4. Simpul merah memiliki dua anak hitam: tidak ada dua merah berturut-turut pada lintasan mana pun.
5. Untuk setiap simpul, semua lintasan dari simpul itu turun ke daun NIL memuat banyak simpul hitam yang sama, yang disebut *black-height*-nya.

Gambar 11.25 mewarnai pohon bersimpul sepuluh dari **Gambar 11.18** sebagai pohon *red-black* yang sah: setiap lintasan dari akar ke NIL melewati tepat dua simpul hitam, dan tidak ada simpul merah yang beranak merah.



Gambar 11.25 Pohon **Gambar 11.18** yang diwarnai sebagai pohon *red-black*: akar hitam, simpul merah beranak hitam, dan setiap lintasan akar-ke-NIL memuat dua simpul hitam.

Aturan 4 dan 5 membatasi tingginya. Menurut aturan 5 banyak simpul hitam sama pada setiap lintasan, dan menurut aturan 4 paling banyak ada simpul merah sebanyak simpul hitam pada lintasan mana pun, sehingga lintasan terpanjang paling banyak dua kali lintasan terpendek. Sebuah argumen hitungan lalu menghasilkan [2]

$$h \leq 2\lg(n+1) \tag{11.14}$$

yang merupakan jaminan lebih lemah daripada Persamaan (11.13): pohon *red-black* dapat sedikit lebih tinggi daripada pohon AVL. Sebagai gantinya, ia memerlukan lebih sedikit rotasi untuk memperbaiki diri: paling banyak dua rotasi per penyisipan dan paling banyak tiga per penghapusan, ditambah beberapa pewarnaan ulang, sedangkan penghapusan AVL mungkin merotasi di setiap tingkat dalam perjalanan naik. Itulah yang membuat pohon *red-black* menjadi pilihan lazim bagi pustaka serbaguna, tempat penyisipan dan penghapusan sama seringnya dengan pencarian: `std::map` dan `std::set` di ketiga pustaka baku C++ utama [11], `TreeMap` dan `TreeSet` di Java [12], dan pustaka pohon seimbang milik kernel Linux sendiri [13] semuanya adalah pohon *red-black*. Pohon AVL, yang lebih rendah, lebih disukai bila pencarian mendominasi. **Tabel 11.4** membandingkan ketiga jenis pohon yang dijumpai dalam bab ini.

Tabel 11.4 *Binary search tree* biasa, AVL, dan *red-black* dibandingkan.

	BST biasa	Pohon AVL	Pohon *red-black*
Data tambahan per simpul	tidak ada	tinggi (atau faktor keseimbangan)	satu bit warna
Tinggi, kasus terburuk	$n-1$	$<1.4405\lg(n+2)$ (Persamaan (11.13))	$\leq 2\lg(n+1)$ (Persamaan (11.14))
Cari, sisip, hapus	$O(n)$ terburuk, $O(\lg n)$ rata-rata	$O(\lg n)$ terburuk	$O(\lg n)$ terburuk
Rotasi per penyisipan	0	paling banyak 2	paling banyak 2
Rotasi per penghapusan	0	sampai $O(\lg n)$	paling banyak 3
Pemakaian khas	pengajaran, kunci acak	beban kerja dominan-pencarian	`std::map`, `TreeMap`, kernel

11.9.4 Pohon seimbang lainnya

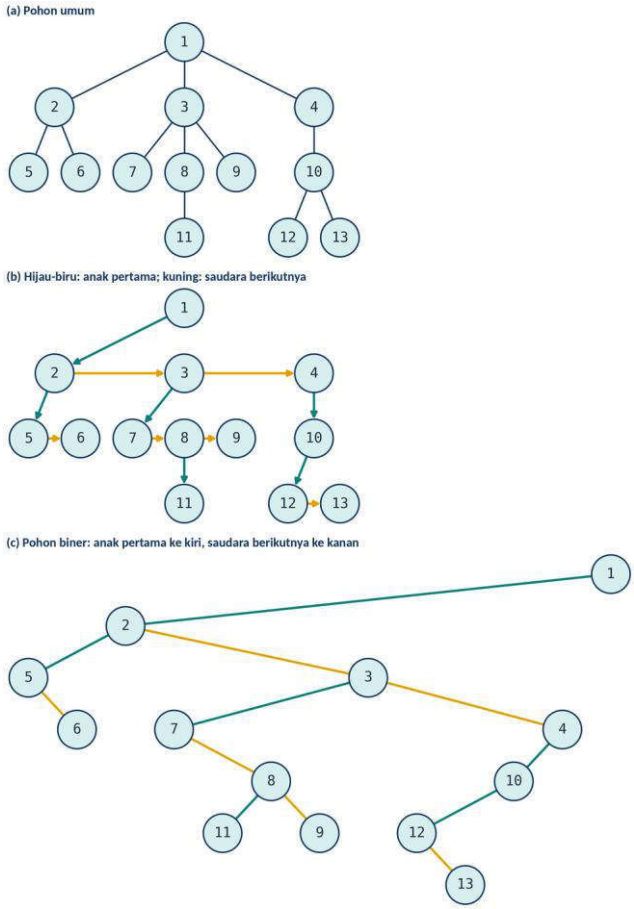
Tiga anggota keluarga lainnya patut dikenal namanya. **B-tree**, yang diperkenalkan Bayer dan McCreight pada 1972 [14], memberi setiap simpul ratusan kunci dan anak, sehingga pohon bermiliar kunci hanya sedalam empat atau lima tingkat; karena setiap tingkat berongkos satu pembacaan diska atau SSD, B-tree menjadi struktur indeks hampir setiap basis data, termasuk

indeks PostgreSQL pada Subbab 11.12 [15]. **Splay tree** (Sleator dan Tarjan [16]) sama sekali tidak menyimpan informasi keseimbangan, tetapi memindahkan setiap simpul yang diakses ke akar dengan rotasi; ongkos barisan operasi yang panjang rata-rata $O(\lg n)$ per operasi. **Treap** (Seidel dan Aragon [17]) memberi setiap simpul prioritas acak dan menjaga pohonnya tetap berupa *heap* menurut prioritas, sehingga bentuknya sama dengan BST acak (Subbab 11.8.2) apa pun urutan penyisipannya. *Heap* pada Bab 14 adalah separuh lain dari gagasan itu.

11.10 Pohon Umum: Konversi dan Rekonstruksi

11.10.1 Anak pertama, saudara berikutnya

Subbab 11.4.3 berjanji bahwa pohon apa pun dapat disimpan dengan dua pointer per simpul. Beri setiap simpul pointer ke **anak pertamanya** dan pointer ke **saudara berikutnya** (anak berikutnya dari induk yang sama). Sekarang bacalah kedua pointer itu sebagai pointer `left` dan `right` sebuah pohon biner: anak pertama menjadi anak kiri, dan saudara berikutnya menjadi anak kanan. Dengan kata lain: anak pertama sebuah simpul menjadi anak kirinya, anak kedua menjadi anak kanan dari anak kiri itu ("cucu kanan" simpul tersebut), anak ketiga menjadi anak kanan dari anak kedua, dan seterusnya menuruni rantai pointer kanan. **Gambar 11.26** melakukan konversi ini pada pohon umum bersimpul tiga belas.



Gambar 11.26 Mengubah pohon umum menjadi pohon biner: (a) pohon umum; (b) setiap simpul hanya menyimpan tautan ke anak pertamanya (hijau-biru) dan ke saudara berikutnya (kuning); (c) tautan-tautan yang sama digambar sebagai pohon biner, anak pertama di kiri dan saudara berikutnya di kanan.

Konversi ini tidak menghilangkan apa pun: pohon umumnya dapat dibaca kembali dengan mengikuti, dari setiap simpul, pointer kirinya lalu rantai pointer kanan dari sana, yang mendaftar anak-anak simpul itu secara berurutan. Dan penelusuran-penelusurannya berpadanan tepat:

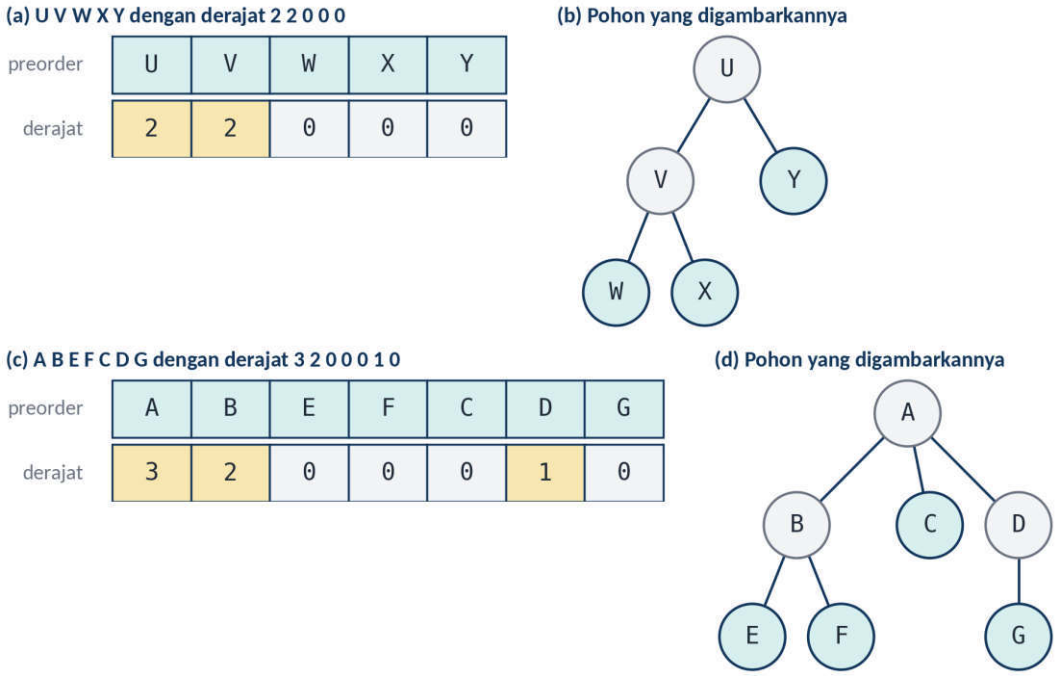
- ***preorder*** pohon umum (sebuah simpul, lalu subpohon setiap anaknya secara berurutan) sama dengan ***preorder*** pohon biner;
- ***postorder*** pohon umum (subpohon setiap anak secara berurutan, lalu simpulnya) sama dengan ***inorder*** pohon biner.

Untuk melihat fakta kedua, perhatikan bahwa dalam pohon biner subpohon kiri sebuah simpul memuat tepat keturunan-keturunannya dalam pohon umum, dan subpohon kanannya memuat saudara-saudaranya yang lebih belakang beserta keturunan mereka. Penelusuran *inorder* karena itu mendaftar semua keturunan sebuah simpul, lalu simpul itu, lalu saudara-saudaranya yang lebih belakang: dan itu persis *postorder* dalam pohon umum. *Postorder* pohon biner tidak berpadanan dengan urutan alami apa pun pada pohon umum, sehingga program di bawah hanya menampilkannya sebagai pembanding.

11.10.2 Membangun kembali pohon dari preorder dan derajat

Daftar *preorder* saja tidak menentukan sebuah pohon: daftar A, B, C bisa berupa akar dengan dua anak atau rantai tiga simpul. Tetapi daftar *preorder* bersama *derajat* setiap simpul (banyak anaknya) menentukannya. Aturannya rekursif, seperti segala sesuatu dalam bab ini: baca simpul berikutnya dan derajatnya d dari daftar; ia adalah akar subpohon berikutnya; lalu bangun d subpohonnya, satu demi satu, dari sisa daftar.

Cara kedua untuk membangun kembali pohon bekerja dari ujung, dan cara ini nyaman dikerjakan dengan tangan. Pindai daftar dari kanan ke kiri, cari simpul yang derajatnya lebih dari 0. Bila Anda menemukan simpul berderajat d, d subpohon tepat di sebelah kanannya (masing-masing sudah tersusun, atau berupa satu daun) adalah anak-anaknya: pasangkan mereka dan ganti mereka dalam daftar dengan pohon gabungannya. Lanjutkan ke kiri sampai akar mendapatkan anak-anaknya. **Gambar 11.27** menerapkannya pada dua daftar.



Gambar 11.27 Dua pohon yang dibangun kembali dari daftar *preorder* dan derajat setiap simpulnya: U V W X Y dengan derajat 2 2 0 0 0 (a, b), serta A B E F C D G dengan derajat 3 2 0 0 0 1 0 (c, d).

- Untuk **U V W X Y** dengan derajat **2 2 0 0 0**, derajat bukan-nol pertama dari kanan adalah milik V, yaitu 2: anak-anaknya adalah dua simpul di sebelah kanannya, W dan X. Buang keduanya; daftar kini U V(W,X) Y. Derajat bukan-nol berikutnya adalah milik U, 2: anak-anaknya adalah V (beserta subpohonnya) dan Y. Pohonnya U(V(W,X),Y).
- Untuk **A B E F C D G** dengan derajat **3 2 0 0 0 1 0**: D berderajat 1, sehingga G anaknya; B berderajat 2, sehingga E dan F anak-anaknya; A berderajat 3, sehingga anak-anaknya B, C, dan D. Pohonnya A(B(E,F),C,D(G)). Sebagai pemeriksaan, jumlah derajatnya $3 + 2 + 1 = 6 = 7 - 1$, banyak sisi, sesuai tuntutan Persamaan (11.1): setiap sisi terhitung sekali, sebagai anak dari suatu simpul.

Preorder beserta derajat adalah cara ringkas untuk *menyimpan* pohon dalam berkas atau mengirimkannya lewat jaringan: satu baris label, satu baris bilangan, dan penerimanya membangun kembali bentuk yang persis sama. Soal 11.6 memperlihatkan pasangan daftar kedua yang menentukan pohon biner: *preorder* bersama *inorder*.

11.10.3 Programnya

Kode 11.10 menyimpan pohon **Gambar 11.26** dengan pointer anak pertama dan saudara berikutnya, mencetaknya dalam notasi kurung, menelusurinya secara *preorder* dan *postorder*, mengubahnya menjadi pohon biner dan menelusurinya dengan tiga cara, lalu membangun kembali kedua pohon **Gambar 11.27** dari daftar *preorder* dan derajatnya.

Kode 11.10 `general_tree.cpp`: pohon umum dalam bentuk anak-pertama/saudara-berikutnya, konversinya menjadi pohon biner, dan pohon-pohon yang dibangun kembali dari daftar *preorder* dan derajat.

```cpp
// general_tree.cpp -- a general tree in first-child/next-sibling form, its
// conversion to a binary tree, and a tree rebuilt from preorder plus degrees.
#include <cstdio>
#include <string>

struct GTN {                    // General Tree Node
    std::string data;
    GTN *firstChild = nullptr;
    GTN *nextSibling = nullptr;
    explicit GTN(const std::string &d) : data(d) {}
};

struct BTN {                    // Binary Tree Node
    std::string data;
    BTN *left = nullptr, *right = nullptr;
    explicit BTN(const std::string &d) : data(d) {}
};

GTN *addChild(GTN *parent, const std::string &d) {   // appends a last child
    GTN *c = new GTN(d);
    if (parent->firstChild == nullptr) {
        parent->firstChild = c;
    } else {
        GTN *s = parent->firstChild;
        while (s->nextSibling != nullptr) s = s->nextSibling;
        s->nextSibling = c;
    }
    return c;
}

void gtPreOrder(const GTN *t) {         // the node, then each child's subtree
    std::printf("%s ", t->data.c_str());
    for (GTN *c = t->firstChild; c != nullptr; c = c->nextSibling) gtPreOrder(c);
}

void gtPostOrder(const GTN *t) {        // each child's subtree, then the node
    for (GTN *c = t->firstChild; c != nullptr; c = c->nextSibling) gtPostOrder(c);
    std::printf("%s ", t->data.c_str());
}
```

Kode 11.10 (Lanjutan) *general_tree.cpp: pohon umum dalam bentuk anak-pertama/saudara-berikutnya, konversinya menjadi pohon biner, dan pohon-pohon yang dibangun kembali dari daftar preorder dan derajat.*

```
void parenthesis(const GTN *t) {        // e.g. A(B(D,E),C)
    std::printf("%s", t->data.c_str());
    if (t->firstChild == nullptr) return;
    std::printf("(");
    for (GTN *c = t->firstChild; c != nullptr; c = c->nextSibling) {
        parenthesis(c);
        if (c->nextSibling != nullptr) std::printf(",");
    }
    std::printf(")");
}

// First child -> left, next sibling -> right.
BTN *convertToBinaryTree(const GTN *node) {
    if (node == nullptr) return nullptr;
    BTN *b = new BTN(node->data);
    b->left = convertToBinaryTree(node->firstChild);
    b->right = convertToBinaryTree(node->nextSibling);
    return b;
}

void btPreOrder(const BTN *t) {
    if (t == nullptr) return;
    std::printf("%s ", t->data.c_str());
    btPreOrder(t->left);
    btPreOrder(t->right);
}

void btInOrder(const BTN *t) {
    if (t == nullptr) return;
    btInOrder(t->left);
    std::printf("%s ", t->data.c_str());
    btInOrder(t->right);
}

void btPostOrder(const BTN *t) {
    if (t == nullptr) return;
    btPostOrder(t->left);
    btPostOrder(t->right);
    std::printf("%s ", t->data.c_str());
}
```

Kode 11.10 (Lanjutan) *general_tree.cpp: pohon umum dalam bentuk anak-pertama/saudara-berikutnya, konversinya menjadi pohon biner, dan pohon-pohon yang dibangun kembali dari daftar preorder dan derajat.*

```
// Rebuilds a general tree from its preorder listing and every node's degree.
GTN *buildFromPreorder(const std::string *pre, const int *deg, int &index) {
    GTN *node = new GTN(pre[index]);
    int children = deg[index];
    ++index;                              // the next unread position
    for (int i = 0; i < children; ++i) {
        GTN *c = buildFromPreorder(pre, deg, index);
        if (node->firstChild == nullptr) {
            node->firstChild = c;
        } else {
            GTN *s = node->firstChild;
            while (s->nextSibling != nullptr) s = s->nextSibling;
            s->nextSibling = c;
        }
    }
    return node;
}

void destroy(GTN *t) {
    if (t == nullptr) return;
    destroy(t->firstChild);
    destroy(t->nextSibling);
    delete t;
}

void destroy(BTN *t) {
    if (t == nullptr) return;
    destroy(t->left);
    destroy(t->right);
    delete t;
}

int main() {
    GTN *root = new GTN("1");
    GTN *n2 = addChild(root, "2");
    GTN *n3 = addChild(root, "3");
    GTN *n4 = addChild(root, "4");
    addChild(n2, "5");
    addChild(n2, "6");
    addChild(n3, "7");
    addChild(addChild(n3, "8"), "11");
    addChild(n3, "9");
    GTN *n10 = addChild(n4, "10");
    addChild(n10, "12");
    addChild(n10, "13");
```

Kode 11.10 (Lanjutan) *general_tree.cpp: pohon umum dalam bentuk anak-pertama/saudara-berikutnya, konversinya menjadi pohon biner, dan pohon-pohon yang dibangun kembali dari daftar preorder dan derajat.*

```
    std::printf("General tree: ");
    parenthesis(root);
    std::printf("\n  PreOrder:  ");
    gtPreOrder(root);
    std::printf("\n  PostOrder: ");
    gtPostOrder(root);

    BTN *b = convertToBinaryTree(root);
    std::printf("\nConverted binary tree:\n  PreOrder:  ");
    btPreOrder(b);
    std::printf("\n  InOrder:   ");
    btInOrder(b);
    std::printf("\n  PostOrder: ");
    btPostOrder(b);
    std::printf("\n");
    destroy(b);
    destroy(root);

    std::string pre1[] = {"U", "V", "W", "X", "Y"};
    int deg1[] = {2, 2, 0, 0, 0};
    std::string pre2[] = {"A", "B", "E", "F", "C", "D", "G"};
    int deg2[] = {3, 2, 0, 0, 0, 1, 0};
    int index = 0;
    GTN *t1 = buildFromPreorder(pre1, deg1, index);
    std::printf("\nFrom U V W X Y with degrees 2 2 0 0 0: ");
    parenthesis(t1);
    index = 0;
    GTN *t2 = buildFromPreorder(pre2, deg2, index);
    std::printf("\nFrom A B E F C D G with degrees 3 2 0 0 0 1 0: ");
    parenthesis(t2);
    std::printf("\n  PostOrder: ");
    gtPostOrder(t2);
    std::printf("\n");
    destroy(t1);
    destroy(t2);
    return 0;
}
```

```
General tree: 1(2(5,6),3(7,8(11),9),4(10(12,13)))
  PreOrder:  1 2 5 6 3 7 8 11 9 4 10 12 13
  PostOrder: 5 6 2 7 11 8 9 3 12 13 10 4 1
Converted binary tree:
  PreOrder:  1 2 5 6 3 7 8 11 9 4 10 12 13
  InOrder:   5 6 2 7 11 8 9 3 12 13 10 4 1
  PostOrder: 6 5 11 9 8 7 13 12 10 4 3 2 1

From U V W X Y with degrees 2 2 0 0 0: U(V(W,X),Y)
From A B E F C D G with degrees 3 2 0 0 0 1 0: A(B(E,F),C,D(G))
  PostOrder: E F B C G D A
```

Keluaran itu menegaskan kedua padanan pada Subbab 11.10.1: kedua baris *preorder* identik, dan baris *postorder* pohon umum sama dengan baris *inorder* pohon biner, 5 6 2 7 11 8 9 3 12 13 10 4 1. *Postorder* pohon biner, 6 5 11 9 8 7 13 12 10 4 3 2 1, berbeda, sesuai dugaan. Kedua pohon yang dibangun kembali tercetak dalam notasi kurung sebagai U(V(W,X),Y) dan A(B(E,F),C,D(G)), yaitu pohon-pohon pada **Gambar 11.27**.

11.11 Pohon Pencarian dalam Pustaka Baku

Program produksi jarang menulis pohon pencariannya sendiri; mereka memakai pohon seimbang yang disediakan bahasanya. Ketiga bahasa dalam buku ini menyediakan hal yang sangat berbeda.

11.11.1 C++: `std::set` dan `std::map`

`std::set<K>` menyimpan himpunan kunci unik dalam urutan terurut, dan `std::map<K, V>` menyimpan pasangan kunci-nilai yang diurutkan menurut kunci. Standar C++ tidak menyebutkan bagaimana keduanya diimplementasikan, tetapi mewajibkan pencarian, penyisipan, dan penghapusan $O(\lg n)$ serta iterasi terurut, dan dalam praktik ketiga implementasi utama (GNU libstdc++, LLVM libc++, dan milik Microsoft) memakai pohon *red-black* [11]. Setiap operasi pada Subbab 11.7 memiliki nama:

- `insert` dan `m[key] = value` menyisipkan; `find` mencari dan mengembalikan `end()` bila kuncinya tidak ada; `erase` menghapus;
- iterasi dari `begin()` sampai `end()` adalah penelusuran *inorder*, sehingga kunci-kuncinya keluar terurut, dan `++it` serta `--it` adalah *successor* dan *predecessor*;
- `begin()` dan `rbegin()` memberikan minimum dan maksimum;
- `lower_bound(k)` mengembalikan kunci pertama yang $\geq k$ dan `upper_bound(k)` kunci pertama yang $> k$, pencarian batas dari Subbab 2.6.3 pada pohon; bersama-sama keduanya menjawab kueri rentang.

Kode 11.11 menyimpan katalog Pustaka Ceria dari Bab 1 dalam `std::map` yang berkunci judul.

Kode 11.11 `std_map.cpp`: katalog Pustaka Ceria dalam `std::map`, dengan daftar terurut, pencarian, kueri rentang, *successor* dan *predecessor*, minimum dan maksimum, serta penghapusan; dan `std::set` yang menolak duplikat.

```
// std_map.cpp -- the Pustaka Ceria catalog in std::map, an ordered map that the
// standard libraries implement as a red-black tree; and std::set.
#include <cstdio>
#include <iterator>
#include <map>
#include <set>
#include <string>

int main() {
    std::map<std::string, std::string> byTitle;     // title -> book code
    const char *books[][2] = {
        {"B1001", "Laskar Pelangi"},     {"B1002", "Bumi Manusia"},
        {"B1003", "Filosofi Kopi"},      {"B1004", "Sejarah Nusantara"},
        {"B1005", "Algoritma & Struktur Data"},
        {"B1006", "Cerita Rakyat Nusantara"},
        {"B1007", "Fisika Dasar"},       {"B1008", "Negeri 5 Menara"}};
    for (auto &b : books) byTitle[b[1]] = b[0];     // O(lg n) each

    std::printf("Catalog by title (an inorder walk):\n");
    for (const auto &[title, code] : byTitle)
        std::printf("  %-27s %s\n", title.c_str(), code.c_str());

    for (const char *t : {"Fisika Dasar", "Ronggeng Dukuh Paruk"}) {
        auto it = byTitle.find(t);                  // a BST search
        std::printf("find \"%s\": %s\n", t,
                    it != byTitle.end() ? it->second.c_str()
                                        : "not in the catalog");
    }

    std::printf("Titles from \"F\" up to but not including \"N\":\n ");
    auto lo = byTitle.lower_bound("F"), hi = byTitle.lower_bound("N");
    for (auto it = lo; it != hi; ++it) std::printf(" [%s]", it->first.c_str());
    std::printf("\n");

    auto it = byTitle.find("Filosofi Kopi");
    std::printf("After \"Filosofi Kopi\" comes \"%s\"\n",
                std::next(it)->first.c_str());
    std::printf("Before \"Filosofi Kopi\" comes \"%s\"\n",
                std::prev(it)->first.c_str());
    std::printf("First title \"%s\", last title \"%s\"\n",
                byTitle.begin()->first.c_str(), byTitle.rbegin()->first.c_str());

    std::size_t removed = byTitle.erase("Bumi Manusia");
    std::printf("erase \"Bumi Manusia\": %zu removed, %zu titles left\n",
                removed, byTitle.size());
```

Kode 11.11 (Lanjutan) *std_map.cpp: katalog Pustaka Ceria dalam std::map, dengan daftar terurut, pencarian, kueri rentang, successor dan predecessor, minimum dan maksimum, serta penghapusan; dan std::set yang menolak duplikat.*

```
    std::set<int> keys;
    for (int x : {65, 5, 70, 3, 10}) keys.insert(x);
    std::printf("set from 65 5 70 3 10:");
    for (int x : keys) std::printf(" %d", x);
    bool added = keys.insert(70).second;            // false: already present
    std::printf("\ninsert 70 again: %s\n", added ? "added" : "rejected");
    return 0;
}
```

```
Catalog by title (an inorder walk):
  Algoritma & Struktur Data   B1005
  Bumi Manusia                B1002
  Cerita Rakyat Nusantara     B1006
  Filosofi Kopi               B1003
  Fisika Dasar                B1007
  Laskar Pelangi              B1001
  Negeri 5 Menara             B1008
  Sejarah Nusantara           B1004
find "Fisika Dasar": B1007
find "Ronggeng Dukuh Paruk": not in the catalog
Titles from "F" up to but not including "N":
  [Filosofi Kopi] [Fisika Dasar] [Laskar Pelangi]
After "Filosofi Kopi" comes "Fisika Dasar"
Before "Filosofi Kopi" comes "Cerita Rakyat Nusantara"
First title "Algoritma & Struktur Data", last title "Sejarah Nusantara"
erase "Bumi Manusia": 1 removed, 7 titles left
set from 65 5 70 3 10: 3 5 10 65 70
insert 70 again: rejected
```

Buku-buku itu disisipkan dalam urutan kodenya, B1001 sampai B1008, dan keluar dalam urutan abjad judul, karena iterasi adalah perjalanan *inorder*. Kueri rentang dari “F” sampai sebelum “N” memakai dua pencarian `lower_bound`, masing-masing $O(\lg n)$, ditambah satu langkah per judul yang dilaporkan. `std::multimap` dan `std::multiset` membolehkan kunci berulang. (C++ juga memiliki `std::unordered_map`, sebuah tabel *hash*, Bab 15: lebih cepat untuk pencarian tepat, tetapi tanpa urutan, tanpa kueri rentang, dan tanpa tetangga.)

11.11.2 Java: `TreeMap` dan `TreeSet`

`TreeMap<K, V>` di Java adalah, menurut dokumentasinya, “*a Red-Black tree based `NavigableMap` implementation*” yang “*provides guaranteed log(n) time cost for the `containsKey`, `get`, `put` and `remove` operations*”, dengan algoritme yang diadaptasi dari Cormen dkk. [12]. `TreeSet<K>` adalah himpunan yang ditopang sebuah `TreeMap`. Antarmuka `NavigableMap` menamai kueri-kueri tetangga secara langsung: `firstKey` dan `lastKey` (minimum, maksimum), `higherKey` dan `lowerKey` (*successor* dan *predecessor* dari kunci apa pun, ada atau tidak), `ceilingKey` dan `floorKey` (kunci terkecil yang $\geq k$, kunci terbesar yang $\leq k$), serta `subMap`, `headMap`, dan `tailMap` untuk rentang. **Kode 11.12** adalah **Kode 11.11** dalam Java dan mencetak keluaran yang sama.

Kode 11.12 `CatalogTreeMap.java`: katalog yang sama dalam `java.util.TreeMap`, dengan `subMap`, `higherKey`, `lowerKey`, `firstKey`, dan `lastKey`; serta sebuah `TreeSet`.

```
// CatalogTreeMap.java -- the Pustaka Ceria catalog in java.util.TreeMap, a
// red-black tree; and TreeSet.
import java.util.Map;
import java.util.TreeMap;
import java.util.TreeSet;

public class CatalogTreeMap {
    public static void main(String[] args) {
        TreeMap<String, String> byTitle = new TreeMap<>();     // title -> book code
        String[][] books = {
            {"B1001", "Laskar Pelangi"},     {"B1002", "Bumi Manusia"},
            {"B1003", "Filosofi Kopi"},      {"B1004", "Sejarah Nusantara"},
            {"B1005", "Algoritma & Struktur Data"},
            {"B1006", "Cerita Rakyat Nusantara"},
            {"B1007", "Fisika Dasar"},       {"B1008", "Negeri 5 Menara"}};
        for (String[] b : books) byTitle.put(b[1], b[0]);      // O(lg n) each

        System.out.println("Catalog by title (an inorder walk):");
        for (Map.Entry<String, String> e : byTitle.entrySet())
            System.out.printf("  %-27s %s%n", e.getKey(), e.getValue());

        for (String t : new String[] {"Fisika Dasar", "Ronggeng Dukuh Paruk"}) {
            String code = byTitle.get(t);                      // a BST search
            System.out.println("find \"" + t + "\": "
                    + (code != null ? code : "not in the catalog"));
        }

        System.out.println("Titles from \"F\" up to but not including \"N\":");
        StringBuilder sb = new StringBuilder(" ");
        for (String t : byTitle.subMap("F", "N").keySet())
            sb.append(" [").append(t).append(']');
        System.out.println(sb);

        System.out.println("After \"Filosofi Kopi\" comes \""
                + byTitle.higherKey("Filosofi Kopi") + "\"");
        System.out.println("Before \"Filosofi Kopi\" comes \""
                + byTitle.lowerKey("Filosofi Kopi") + "\"");
        System.out.println("First title \"" + byTitle.firstKey()
                + "\", last title \"" + byTitle.lastKey() + "\"");

        int removed = byTitle.remove("Bumi Manusia") != null ? 1 : 0;
        System.out.println("erase \"Bumi Manusia\": " + removed + " removed, "
                + byTitle.size() + " titles left");
```

Kode 11.12 (Lanjutan) *CatalogTreeMap.java: katalog yang sama dalam java.util.TreeMap, dengan subMap, higherKey, lowerKey, firstKey, dan lastKey; serta sebuah TreeSet.*

```
        TreeSet<Integer> keys = new TreeSet<>();
        for (int x : new int[] {65, 5, 70, 3, 10}) keys.add(x);
        System.out.print("set from 65 5 70 3 10:");
        for (int x : keys) System.out.print(" " + x);
        boolean added = keys.add(70);            // false: already present
        System.out.println("\ninsert 70 again: " + (added ? "added" : "rejected"));
    }
}
```

Ketika `std::map` memerlukan iterator dan `std::next`, `TreeMap` menjawab `higherKey("Filosofi Kopi")` dalam satu panggilan. Seperti pada `std::map`, alternatif berbasis *hash*-nya adalah `HashMap`, yang tidak berurutan.

11.11.3 Python: tanpa pohon pencarian bawaan

Pustaka baku Python tidak memiliki pohon pencarian seimbang. `dict` dan `set` miliknya adalah tabel *hash* (Bab 15): pencarian tepat dalam waktu rata-rata $O(1)$, dan `dict` mengingat urutan penyisipan, tetapi keduanya tidak terurut dan tidak dapat menjawab "kunci berikutnya sesudah yang ini" atau "semua kunci antara F dan N" tanpa memindai atau mengurutkan semuanya. Program Python memperoleh perilaku terurut dengan dua cara.

List terurut dengan `bisect`. Modul `bisect` melakukan *binary search* pada list terurut: `bisect_left(a, x)` adalah batas bawah pada Subbab 2.6.3, dan `insort(a, x)` menyisipkan x di tempatnya [18]. Pencariannya $O(\lg n)$; penyisipan dan penghapusannya $O(n)$, karena list harus bergeser, persis seperti temuan Subbab 7.3 untuk larik. Kedengarannya buruk, tetapi pergeseran itu hanyalah satu `memmove` pointer di C, dan untuk list sampai beberapa ratus ribu elemen biasanya lebih cepat daripada pohon yang ditulis dalam Python. **Kode 11.13** adalah katalog itu lagi, dengan list judul terurut untuk urutannya dan `dict` untuk kodenya, dan mencetak keluaran yang sama dengan **Kode 11.11**.

Kode 11.13 `library_trees.py`: katalog yang dijaga terurut dengan list terurut dan modul `bisect`, dengan `dict` untuk kodenya.

```
# library_trees.py -- Python has no built-in search tree. The Pustaka Ceria
# catalog kept in order with a sorted list and the bisect module instead.
from bisect import bisect_left, insort

BOOKS = [("B1001", "Laskar Pelangi"), ("B1002", "Bumi Manusia"),
         ("B1003", "Filosofi Kopi"), ("B1004", "Sejarah Nusantara"),
         ("B1005", "Algoritma & Struktur Data"),
         ("B1006", "Cerita Rakyat Nusantara"),
         ("B1007", "Fisika Dasar"), ("B1008", "Negeri 5 Menara")]


def main():
    titles = []                     # always sorted
    code_of = {}                    # title -> book code (a hash table, Chapter 15)
    for code, title in BOOKS:
        insort(titles, title)       # binary search, then an O(n) shift
        code_of[title] = code

    print("Catalog by title (an inorder walk):")
    for t in titles:
        print(f"  {t:<27} {code_of[t]}")

    for t in ("Fisika Dasar", "Ronggeng Dukuh Paruk"):
        i = bisect_left(titles, t)
        found = i < len(titles) and titles[i] == t
        print(f'find "{t}": {code_of[t] if found else "not in the catalog"}')

    print('Titles from "F" up to but not including "N":')
    lo, hi = bisect_left(titles, "F"), bisect_left(titles, "N")
    print(" " + "".join(f" [{t}]" for t in titles[lo:hi]))

    i = bisect_left(titles, "Filosofi Kopi")
    print(f'After "Filosofi Kopi" comes "{titles[i + 1]}"')
    print(f'Before "Filosofi Kopi" comes "{titles[i - 1]}"')
    print(f'First title "{titles[0]}", last title "{titles[-1]}"')

    i = bisect_left(titles, "Bumi Manusia")
    removed = 0
    if i < len(titles) and titles[i] == "Bumi Manusia":
        del titles[i]               # another O(n) shift
        del code_of["Bumi Manusia"]
        removed = 1
    print(f'erase "Bumi Manusia": {removed} removed, {len(titles)} titles left')
```

Kode 11.13 (Lanjutan) *library_trees.py: katalog yang dijaga terurut dengan list terurut dan modul bisect, dengan dict untuk kodenya.*

```
    keys = sorted({65, 5, 70, 3, 10})
    print("set from 65 5 70 3 10:" + "".join(f" {x}" for x in keys))
    added = 70 not in keys
    print(f"insert 70 again: {'added' if added else 'rejected'}")


if __name__ == "__main__":
    main()
```

Paket `sortedcontainers`. Paket pihak ketiga `sortedcontainers` (versi 2.4.0, dipasang dengan `pip install sortedcontainers`) menyediakan `SortedList`, `SortedDict`, dan `SortedSet` [19]. Menariknya, paket ini juga bukan pohon: ia menyimpan list berisi sublist terurut yang masing-masing sekitar seribu elemen, sehingga penyisipan murah (hanya satu sublist pendek yang bergeser) dan tata letaknya tetap ramah memori seperti larik. Antarmukanya menawarkan persis operasi-operasi pohon: `irange` untuk rentang, `bisect_left` dan `bisect_right` untuk tetangga, `peekitem` untuk kunci terkecil ke-*i*. **Kode 11.14** memperlihatkannya pada katalog yang sama.

Kode 11.14 `sorted_catalog.py`: katalog dalam `SortedDict` dari paket `sortedcontainers`.

```
# sorted_catalog.py -- the same catalog with the third-party sortedcontainers
# package (pip install sortedcontainers), whose SortedDict keeps keys in order.
from sortedcontainers import SortedDict

catalog = SortedDict({"Laskar Pelangi": "B1001", "Bumi Manusia": "B1002",
                      "Filosofi Kopi": "B1003", "Sejarah Nusantara": "B1004",
                      "Algoritma & Struktur Data": "B1005",
                      "Cerita Rakyat Nusantara": "B1006",
                      "Fisika Dasar": "B1007", "Negeri 5 Menara": "B1008"})

print("First two titles:", list(catalog.keys()[:2]))
print("From F up to N:", list(catalog.irange("F", "N", inclusive=(True, False))))
i = catalog.bisect_right("Filosofi Kopi")            # position after the key
print("After Filosofi Kopi:", catalog.peekitem(i))
catalog["Ronggeng Dukuh Paruk"] = "B1009"            # stays in order
print("Last title:", catalog.peekitem(-1))
```

```
First two titles: ['Algoritma & Struktur Data', 'Bumi Manusia']
From F up to N: ['Filosofi Kopi', 'Fisika Dasar', 'Laskar Pelangi']
After Filosofi Kopi: ('Fisika Dasar', 'B1007')
Last title: ('Sejarah Nusantara', 'B1004')
```

Tabel 11.5 merangkum ketiga pustaka itu.

Tabel 11.5 Himpunan dan *map* terurut dalam pustaka baku C++, Java, dan Python, beserta alternatif berbasis *hash*-nya.

Kebutuhan	C++	Java	Python
Himpunan terurut	`std::set` (*red-black*)	`TreeSet` (*red-black*)	`list` terurut + `bisect`; `SortedSet` (pihak ketiga)
Map terurut	`std::map` (*red-black*)	`TreeMap` (*red-black*)	`SortedDict` (pihak ketiga)
Successor, *predecessor*	`std::next`, `std::prev`, `upper_bound`	`higherKey`, `lowerKey`	`bisect_right`, `bisect_left`
Kueri rentang	`lower_bound` .. `lower_bound`	`subMap`	irisan di antara dua panggilan `bisect_left`; `irange`
Tak terurut (*hash*)	`std::unordered_map`	`HashMap`	`dict`

11.12 Profitina dalam Praktik

Profitina adalah platform visibilitas AI (GEO): ia memindai situs web sebuah merek, mengukur seberapa siap situs itu dikutip mesin penjawab AI, dari sinyal-sinyal seperti konten yang dapat diekstrak mesin, data terstruktur, dan kredibilitas teknis, lalu membantu memperbaiki temuannya [20]. Servernya adalah aplikasi Python yang dibangun dengan FastAPI di atas PostgreSQL. Pohon ada di mana-mana dalam pekerjaan itu, karena hampir segala sesuatu yang dibaca pemindai adalah pohon. Kita melukiskan bentuk setiap masalah dan golongan struktur yang cocok untuknya; pilihan implementasi produk itu sendiri adalah urusannya sendiri.

- **Halaman web adalah pohon.** Peramban, *crawler*, dan mesin penjawab semuanya mengubah HTML menjadi *Document Object Model* (DOM), pohon umum yang setiap elemennya adalah simpul yang anak-anaknya adalah elemen-elemen yang bersarang di dalamnya [21]. "Urutan dokumen", urutan kemunculan teks, adalah *preorder* pohon itu. Dapat tidaknya konten utama sebuah halaman diekstrak dengan rapi sebagian besar adalah pertanyaan tentang bentuk

pohonnya: apakah teks artikel itu berupa subpohon yang padat, atau tersebar di antara ratusan simpul bersaudara berisi navigasi dan iklan?

- **Judul-judul membentuk kerangka.** Judul `h1` sampai `h6` sebuah halaman mendefinisikan pohon kerangka, daftar isi dalam notasi tingkat pada Subbab 11.3. Halaman yang melompat dari `h1` langsung ke `h4`, atau memiliki tiga elemen `h1`, memiliki kerangka yang cacat, dan pembaca maupun mesin sama-sama kehilangan strukturnya. Memeriksanya adalah perjalanan *preorder* dengan *stack* berisi judul yang sedang terbuka, persis pola **Pseudokode 11.3** yang mendorong ketika turun lebih dalam dan mengambil ketika naik kembali.
- **Data terstruktur adalah pohon.** JSON-LD schema.org yang disematkan sebuah situs untuk mesin penjawab adalah objek bersarang: sebuah `Organization` dengan `address`, yang memiliki `streetAddress`, dan seterusnya. JSON adalah notasi kurung dengan kurung kurawal, dan memvalidasinya adalah penelusuran rekursif yang memeriksa setiap subpohon terhadap tipe yang diharapkan pada kedalaman itu.
- **Situs adalah pohon tautan.** Mulai dari halaman beranda dan mengikuti tautan secara melebar (urutan tingkat, **Pseudokode 11.2**) memberi setiap halaman sebuah *kedalaman perayapan*, tingkatnya dalam pohon yang terbentuk. Halaman yang terletak banyak klik di dalam ditemukan *crawler* lebih belakangan, atau tidak sama sekali; pemindai yang melaporkan kedalaman perayapan sedang melaporkan kedalaman simpul.
- **Utas diskusi adalah pohon.** Dalam komunitas *real-time*, sebuah balasan tergantung di bawah unggahan yang dijawabnya, dan balasan atas balasan tergantung di bawahnya lagi. Menampilkan sebuah utas adalah penelusuran *preorder* yang dicetak dalam notasi tingkat: setiap unggahan menjorok sesuai kedalamannya.
- **Data terurut tinggal di B-tree.** Tren skor mingguan sebuah merek dan ekspor "pemindaian antara Juli dan September" adalah kueri rentang atas waktu. Di PostgreSQL kueri seperti itu dijawab oleh indeks B-tree (Subbab 11.9.4) pada kolom cap waktu [15]: dua kali turun ke ujung-ujung rentang dan pemindaian terurut di antaranya, bentuk yang sama dengan kueri rentang pada program di bawah.

Versi yang dapat dijalankan dari gagasan-gagasan ini memakai studi kasus Profitina Laundry. Kedua belas pesanannya dalam sehari (pesanan pada Subbab 2.13.2) masing-masing siap diambil selang waktu tertentu sesudah datang: satu jam ditambah 20 menit per kilogram. Papan pengambilan menyimpan pesanan dalam BST berkunci waktu siap, dan menjawab pertanyaan-pertanyaan di meja layanan: pesanan mana yang siap antara 10:00 dan 12:59, pesanan mana yang siap berikutnya pada pukul 13:05, dan pesanan mana yang terakhir sudah siap. **Kode 11.15** mengimplementasikan papan itu; kueri rentangnya hanya mengunjungi subpohon yang mungkin memuat waktu dalam rentang, `ceiling` dan `floorNode` adalah pencarian bergaya *successor* dari Subbab 11.7.9 untuk kunci yang tidak harus ada di pohon, dan pesanan yang sudah diambil dibuang dengan varian *successor* penghapusan.

Kode 11.15 `laundry_tree.cpp`: papan pengambilan Profitina Laundry, sebuah *binary search tree* berisi pesanan berkunci waktu siap, dengan kueri rentang yang dipangkas, *ceiling*, *floor*, dan penghapusan.

```cpp
// laundry_tree.cpp -- Profitina Laundry's pickup board: orders in a binary search
// tree keyed by the time they are ready, with range, floor and ceiling queries.
#include <cstdio>
#include <string>

struct Order {
    int number;
    std::string customer;
    int ready;              // minutes after midnight
};

struct Node {
    Order order;
    Node *left, *right;
};

std::string hhmm(int m) {
    char buf[16];
    std::snprintf(buf, sizeof buf, "%02d:%02d", m / 60 % 24, m % 60);
    return buf;
}

Node *insert(Node *t, const Order &o) {
    if (t == nullptr) return new Node{o, nullptr, nullptr};
    if (o.ready < t->order.ready) t->left = insert(t->left, o);
    else t->right = insert(t->right, o);        // equal times go right
    return t;
}

void board(const Node *t) {                     // inorder: earliest first
    if (t == nullptr) return;
    board(t->left);
    std::printf("  %s  #%d %s\n", hhmm(t->order.ready).c_str(), t->order.number,
                t->order.customer.c_str());
    board(t->right);
}
```

Kode 11.15 (Lanjutan) *laundry_tree.cpp: papan pengambilan Profitina Laundry, sebuah binary search tree berisi pesanan berkunci waktu siap, dengan kueri rentang yang dipangkas, ceiling, floor, dan penghapusan.*

```
// Orders ready in [lo, hi], visiting only subtrees that can hold such times.
void range(const Node *t, int lo, int hi, int &visited) {
    if (t == nullptr) return;
    ++visited;
    if (lo < t->order.ready) range(t->left, lo, hi, visited);
    if (lo <= t->order.ready && t->order.ready <= hi)
        std::printf("  %s  #%d %s\n", hhmm(t->order.ready).c_str(), t->order.number,
                    t->order.customer.c_str());
    if (t->order.ready < hi) range(t->right, lo, hi, visited);
}

const Node *ceiling(const Node *t, int x) {    // earliest order ready at or after x
    const Node *best = nullptr;
    while (t != nullptr) {
        if (t->order.ready >= x) {
            best = t;                           // a candidate; try earlier
            t = t->left;
        } else {
            t = t->right;
        }
    }
    return best;
}

const Node *floorNode(const Node *t, int x) {  // latest order ready at or before x
    const Node *best = nullptr;
    while (t != nullptr) {
        if (t->order.ready <= x) {
            best = t;
            t = t->right;
        } else {
            t = t->left;
        }
    }
    return best;
}
```

Kode 11.15 (Lanjutan) *laundry_tree.cpp: papan pengambilan Profitina Laundry, sebuah binary search tree berisi pesanan berkunci waktu siap, dengan kueri rentang yang dipangkas, ceiling, floor, dan penghapusan.*

```
Node *remove(Node *t, int ready) {              // successor variant
    if (t == nullptr) return nullptr;
    if (ready < t->order.ready) {
        t->left = remove(t->left, ready);
    } else if (ready > t->order.ready) {
        t->right = remove(t->right, ready);
    } else if (t->left == nullptr || t->right == nullptr) {
        Node *child = (t->left != nullptr) ? t->left : t->right;
        delete t;
        return child;
    } else {
        Node *s = t->right;
        while (s->left != nullptr) s = s->left;
        t->order = s->order;
        t->right = remove(t->right, s->order.ready);
    }
    return t;
}

int height(const Node *t) {
    if (t == nullptr) return -1;
    int a = height(t->left), b = height(t->right);
    return (a > b ? a : b) + 1;
}

int count(const Node *t) {
    return t == nullptr ? 0 : count(t->left) + count(t->right) + 1;
}

void destroy(Node *t) {
    if (t == nullptr) return;
    destroy(t->left);
    destroy(t->right);
    delete t;
}
```

Kode 11.15 (Lanjutan) *laundry_tree.cpp: papan pengambilan Profitina Laundry, sebuah binary search tree berisi pesanan berkunci waktu siap, dengan kueri rentang yang dipangkas, ceiling, floor, dan penghapusan.*

```
int main() {
    struct Row {
        int number;
        const char *received, *customer;
        double kg;
    };
    Row in[] = {
        {1001, "07:45", "Sari", 3.5},  {1002, "08:10", "Budi", 5.0},
        {1003, "08:55", "Dewi", 2.0},  {1004, "09:30", "Andi", 7.5},
        {1005, "10:05", "Rina", 4.0},  {1006, "10:40", "Joko", 6.0},
        {1007, "11:15", "Maya", 3.0},  {1008, "11:50", "Tono", 8.0},
        {1009, "12:20", "Lina", 2.5},  {1010, "13:05", "Agus", 5.5},
        {1011, "14:30", "Fitri", 4.5}, {1012, "15:10", "Hadi", 3.0}};
    Node *root = nullptr;
    for (auto &r : in) {                        // ready = received + 60 + 20 per kg
        int h, m;
        std::sscanf(r.received, "%d:%d", &h, &m);
        int ready = h * 60 + m + 60 + (int) (20 * r.kg);
        root = insert(root, Order{r.number, r.customer, ready});
    }
    std::printf("Pickup board (%d orders, tree height %d):\n",
                count(root), height(root));
    board(root);

    int visited = 0;
    std::printf("Ready between 10:00 and 12:59:\n");
    range(root, 600, 779, visited);
    std::printf("  (%d of %d nodes visited)\n", visited, count(root));

    int now = 13 * 60 + 5;
    const Node *c = ceiling(root, now), *f = floorNode(root, now);
    std::printf("At %s the next order to be ready is #%d %s at %s\n",
                hhmm(now).c_str(), c->order.number, c->order.customer.c_str(),
                hhmm(c->order.ready).c_str());
    std::printf("The last order already ready is #%d %s at %s\n",
                f->order.number, f->order.customer.c_str(),
                hhmm(f->order.ready).c_str());
```

Kode 11.15 (Lanjutan) *laundry_tree.cpp: papan pengambilan Profitina Laundry, sebuah binary search tree berisi pesanan berkunci waktu siap, dengan kueri rentang yang dipangkas, ceiling, floor, dan penghapusan.*

```
    std::printf("Picked up: #%d %s and #1003 Dewi\n", f->order.number,
                f->order.customer.c_str());
    root = remove(root, f->order.ready);
    root = remove(root, 10 * 60 + 35);
    std::printf("Pickup board (%d orders, tree height %d):\n",
                count(root), height(root));
    board(root);
    destroy(root);
    return 0;
}
```

```
Pickup board (12 orders, tree height 7):
  09:55  #1001 Sari
  10:35  #1003 Dewi
  10:50  #1002 Budi
  12:25  #1005 Rina
  13:00  #1004 Andi
  13:15  #1007 Maya
  13:40  #1006 Joko
  14:10  #1009 Lina
  15:30  #1008 Tono
  15:55  #1010 Agus
  17:00  #1011 Fitri
  17:10  #1012 Hadi
Ready between 10:00 and 12:59:
  10:35  #1003 Dewi
  10:50  #1002 Budi
  12:25  #1005 Rina
  (5 of 12 nodes visited)
At 13:05 the next order to be ready is #1007 Maya at 13:15
The last order already ready is #1004 Andi at 13:00
Picked up: #1004 Andi and #1003 Dewi
Pickup board (10 orders, tree height 7):
  09:55  #1001 Sari
  10:50  #1002 Budi
  12:25  #1005 Rina
  13:15  #1007 Maya
  13:40  #1006 Joko
  14:10  #1009 Lina
  15:30  #1008 Tono
  15:55  #1010 Agus
  17:00  #1011 Fitri
  17:10  #1012 Hadi
```

Gambar 11.28 menggambar pohon papan itu dan menandai simpul-simpul yang dikunjungi kueri rentang.



Gambar 11.28 Pohon papan pengambilan, berkunci waktu siap: kueri rentang 10:00 sampai 12:59 mengunjungi lima simpul (hijau-biru dan kuning) dan melaporkan tiga (kuning); tujuh simpul lainnya tidak pernah disentuh.

Dua pelajaran muncul dari jalannya program ini. Kueri rentang mengunjungi 5 dari 12 simpul: di 13:00 ia tidak belok ke kanan, karena tidak ada apa pun di subpohon itu yang dapat siap sebelum 12:59, dan di 09:55 ia tidak belok ke kiri. Dan tinggi pohonnya 7, padahal 12 simpul muat dalam pohon bertinggi 3 menurut Persamaan (11.6). Penyebabnya adalah peringatan Subbab 11.8

dalam situasi nyata: pesanan datang dalam urutan waktu, dan waktu siap sebagian besar mengikuti waktu datang, sehingga kunci-kuncinya datang hampir terurut dan pohonnya condong ke kanan. Untuk papan berisi selusin pesanan hal itu tidak penting; untuk pesanan setahun, penting. Sistem sungguhan akan menyimpan papan itu dalam struktur seimbang, `std::map` atau `TreeMap` berkunci waktu siap, yang `lower_bound`, `subMap`, `ceilingKey`, dan `floorKey`-nya persis kueri-kueri pada program ini. Versi Java dan Python (`LaundryTree.java`, `laundry_tree.py`) mencetak keluaran yang sama.

11.13 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Mencampuradukkan kedalaman dan tinggi. Kedalaman menghitung sisi turun dari akar ke sebuah simpul; tinggi menghitung sisi dari sebuah simpul turun ke daun terdalamnya. Akar berkedalaman 0 dan bertinggi setinggi pohon; daun bertinggi 0. Periksa kesepakatan yang dipakai sebuah teks: sebagian menghitung tingkat dari 1 dan tinggi dalam simpul, yang menambahkan 1 pada setiap jawaban (Subbab 11.2.3 dan 11.7.5).

Memeriksa sifat BST hanya antara induk dan anak. Setiap kunci di *subpohon* kiri harus lebih kecil, bukan hanya anak kirinya. Pohon 10(5(2, 12), 15) lolos setiap uji induk-anak namun bukan BST. Periksalah dengan batas-batas yang diturunkan dari para leluhur (Soal 11.8).

Kehilangan perubahan pada pointer. `void insert(Node *root, int x)` menerima *salinan* pointer; menugaskan simpul baru kepadanya mengubah salinan itu, dan pohonnya tidak pernah tumbuh. Lewatkan `Node **root` atau `Node *&root`, atau kembalikan akar subpohon yang baru lalu simpan: `t->left = insert(t->left, x)` (Subbab 11.7.2). Di Java dan Python, melupakan penugasan `root = insert(root, x)` menghilangkan simpul pertama dengan cara yang sama.

Menghapus simpul beranak dua dengan menyambung-lewati. "Ganti simpul dengan anaknya" dari kasus 2 tidak berhasil bila ada dua anak. Ganti kuncinya dengan kunci *successor* (atau *predecessor*) dan hapus simpul itu dari subpohonnya, tempat ia paling banyak memiliki satu anak (Subbab 11.7.10).

Melupakan kasus dasar, atau mendereferensi NIL. Setiap fungsi pohon rekursif diawali `if (root == nullptr)`. `min(root)->data` pada pohon kosong, atau `root->left->data` pada simpul tanpa anak kiri, adalah perilaku tak terdefinisi (Subbab 7.11).

Melepaskan pohon dalam urutan yang salah, atau tidak sama sekali. Hapus anak-anak sebelum induknya (*postorder*); menghapus induk lebih dahulu menghilangkan pointer ke anak-anaknya. Mengisi `root = nullptr` tanpa menghapus apa pun membocorkan setiap simpul. Jalankan program pohon di bawah Valgrind atau AddressSanitizer (Subbab 11.7.7).

Menganggap BST biasa selalu $O(\lg n)$. Ongkosnya $O(h)$, dan h bisa bernilai $n-1$: masukan terurut atau hampir terurut, yang lazim, mengubah BST menjadi list dan pembangunannya menjadi pekerjaan $\Theta(n^2)$ (Subbab 11.8.2). Pakailah pohon seimbang (`std::map`, `TreeMap`) bila urutan masukan tidak dalam kendali Anda.

Merekursi pada pohon yang mungkin merosot. Kedalaman rekursi sama dengan tinggi. Rantai sejuta simpul meluapkan *stack* C++, dan batas bawaan Python berhenti di sekitar seribu (Subbab 10.12). Pakailah penyisipan dan pencarian iteratif, serta *stack* eksplisit untuk penelusuran, bila pohonnya tidak diketahui seimbang (Subbab 11.6.4).

Memakai penelusuran untuk pekerjaan yang memerlukan penelusuran lain. Menghapus memerlukan *postorder*; menyalin dan menyimpan memerlukan *preorder*; keluaran terurut

memerlukan *inorder*; "yang terdekat ke akar lebih dahulu" memerlukan urutan tingkat dengan *queue*.

Mencampuradukkan nama-nama bentuk. "*Full*", "*complete*", dan "*perfect*" berarti hal yang berbeda di buku yang berbeda (Subbab 11.5.2); sebutkan maksud Anda dengan syaratnya, bukan hanya dengan namanya.

11.14 Ringkasan Bab

- **Pohon** adalah akar dengan nol subpohon atau lebih yang saling lepas; ia nonlinear dan hierarkis. Pohon dengan n simpul memiliki $n-1$ sisi (Persamaan (11.1)) dan tepat satu lintasan dari akar ke setiap simpul. Kosakatanya (induk, anak, saudara, daun, simpul internal, leluhur, keturunan, lintasan, tingkat, kedalaman, tinggi, derajat, subpohon, ukuran) diringkas pada **Tabel 11.1**.
- Pohon dapat digambar dengan tingkat-tingkatnya, sebagai diagram Venn, dalam notasi tingkat (indentasi), atau dalam notasi kurung seperti `(A(B(D,E(I,J)),C(F,G,H)))`.
- **Representasi.** Pohon biner muat dalam larik dengan anak di $2i+1$ dan $2i+2$ (Persamaan (11.2)), ideal untuk pohon *complete* dan boros untuk yang lain; representasi dinamis memakai pointer `left` dan `right` dan memiliki $n+1$ pointer nol (Persamaan (11.3)). Pohon apa pun dapat disimpan dengan pointer anak pertama dan saudara berikutnya, yang menjadikannya pohon biner.
- **Pohon biner** dapat berbentuk *full*, *perfect*, *complete*, atau condong (**Tabel 11.2**). Tingkat ℓ memuat paling banyak 2^{ℓ} simpul (Persamaan (11.4)), pohon bertinggi h paling banyak $2^{h+1}-1$ (Persamaan (11.5)), sehingga $h \geq \lfloor \lg n \rfloor$ (Persamaan (11.6)); pohon *full* memiliki satu daun lebih banyak daripada simpul internalnya (Persamaan (11.7)).
- **Penelusuran.** *Preorder*, *inorder*, dan *postorder* (**Pseudokode 11.1**) adalah tiga saat sebuah Euler tour melewati sebuah simpul; urutan tingkat memakai *queue* (**Pseudokode 11.2**); keempatnya dapat berjalan dengan *stack* atau *queue* eksplisit (**Pseudokode 11.3**) dan semuanya memerlukan waktu $\Theta(n)$ (Persamaan (11.8)). Tiga urutan telusur-dalam pada pohon ekspresi menghasilkan notasi prefiks, infiks, dan postfiks, dan notasi postfiks membangun pohonnya dengan *stack* (**Pseudokode 11.4**).
- **Binary search tree** menyimpan kunci yang lebih kecil di subpohon kiri dan yang lebih besar di subpohon kanan setiap simpul, sehingga *inorder* terurut. Penyisipan (**Pseudokode 11.5**, contoh berjalan `insert(65, 5, 70, 3, 10)`), pencarian (**Pseudokode 11.6**), minimum dan maksimum (**Pseudokode 11.7**), *successor* dan *predecessor* (**Pseudokode 11.9**), serta penghapusan (**Pseudokode 11.10**: daun dilepas, simpul beranak satu digantikan anaknya, simpul beranak dua mengambil kunci *successor* atau *predecessor*-nya) semuanya berongkos $O(h)$ (Persamaan (11.10), **Tabel 11.3**). Hitung simpul, daun, dan tinggi (Persamaan (11.9)) berongkos $\Theta(n)$; mencetak menyamping adalah *inorder* terbalik (**Pseudokode 11.8**).
- **Bentuk.** Urutan penyisipan acak menghasilkan kedalaman rata-rata sekitar $1.39 \lg n$ (Persamaan (11.12)); masukan terurut, dan seluruhnya 2^{n-1} urutan (Persamaan (11.11)), menghasilkan rantai bertinggi $n-1$, yang terukur pada **Kode 11.8** sebagai pembangunan $\Theta(n^2)$.
- **Pohon penyeimbang-diri** memperbaiki bentuknya dengan rotasi $O(1)$ yang mempertahankan barisan *inorder*. Pohon AVL menjaga setiap faktor keseimbangan dalam $\{-1,0,1\}$ dan bertinggi kurang dari $1.4405 \lg(n+2)$ (Persamaan (11.13)); pohon *red-black* menjaga lima aturan warna dan bertinggi paling banyak $2 \lg(n+1)$ (Persamaan (11.14)) dengan rotasi yang lebih sedikit (**Tabel 11.4**).

- **Pohon umum** dapat diubah menjadi pohon biner (anak pertama di kiri, saudara berikutnya di kanan), dengan *preorder* umum sama dengan *preorder* biner dan *postorder* umum sama dengan *inorder* biner; daftar *preorder* beserta derajat menentukan sebuah pohon.
- **Pustaka.** `std::set` dan `std::map` di C++ serta `TreeSet` dan `TreeMap` di Java adalah pohon *red-black* dengan *successor*, *predecessor*, dan kueri rentang; Python tidak memiliki pohon pencarian bawaan dan memakai list terurut dengan `bisect` atau paket pihak ketiga `sortedcontainers` (**Tabel 11.5**).

11.15 Soal Latihan

11.1 Untuk pohon `(P(Q(T,U(W)),R,S(V)))` dalam notasi kurung, sebutkan: akarnya; simpul internal dan daunnya; anak-anak Q; saudara-saudara R; leluhur W; keturunan Q; kedalaman U dan tinggi Q; tinggi dan derajat pohon; ukurannya dan banyak sisinya.

11.2 Gambarlah pohon pada Soal 11.1 beserta tingkat-tingkatnya, lalu uraikan diagram Venn dan notasi tingkat (indentasi)-nya. Simpul mana saja yang berada di setiap tingkat?

11.3 Sisipkan 40, 20, 60, 10, 30, 50, 70 ke *binary search tree* kosong. Gambarlah pohonnya sesudah setiap penyisipan, dan sebutkan tingginya serta penelusuran *preorder*, *inorder*, *postorder*, dan urutan tingkatnya. Berapa tingginya bila kunci yang sama disisipkan dalam urutan naik?

11.4 Simpan pohon yang dibangun pada Soal 11.3 dalam larik seperti pada Subbab 11.4.1. Kunci mana yang berada di slot 5, dan di mana induknya menurut Persamaan (11.2)? Simpan pohon `insert(65, 5, 70, 3, 10)` dengan cara yang sama: berapa slot yang kosong, dan mengapa?

11.5 (a) Dalam sebuah pohon biner, misalkan n_0, n_1, dan n_2 adalah banyak simpul yang memiliki 0, 1, dan 2 anak. Buktikan bahwa $n_0 = n_2 + 1$, apa pun bentuknya. (b) Di antara dua nilai berapakah banyak simpul pohon biner *complete* bertinggi h harus berada? Simpulkan bahwa tingginya $\lfloor \lg n \rfloor$.

11.6 Sebuah pohon biner memiliki *preorder* A B D G C E F H dan *inorder* D G B A E C H F. Gambarlah pohonnya, sebutkan *postorder*-nya, dan tulislah `Node *buildPreIn(...)`, yang membangun kembali pohon biner dari daftar *preorder* dan *inorder*-nya. Lalu perlihatkan dengan pohon dua simpul bahwa *preorder* dan *postorder* bersama-sama *tidak* menentukan sebuah pohon biner.

11.7 Pada pohon bersimpul sepuluh di **Gambar 11.18** (disisipkan sebagai 50, 30, 70, 20, 40, 60, 80, 35, 45, 65), hapuslah 30, pertama dengan varian *successor* lalu, mulai lagi dari pohon semula, dengan varian *predecessor*. Gambarlah kedua hasilnya. Kemudian hapuslah 70 dari pohon semula dengan setiap varian.

11.8 Tulislah `bool isBST(const Node *t, long long lo, long long hi)`, yang memutuskan apakah sebuah pohon biner merupakan *binary search tree*. Jelaskan mengapa pemeriksaan "anak kiri setiap simpul lebih kecil dan anak kanannya lebih besar" tidak cukup, dengan memakai pohon 10(5(2, 12), 15).

11.9 Tulislah fungsi yang mengembalikan simpul yang memuat kunci terkecil ke-k sebuah BST ($k = 1$ adalah minimum), atau `nullptr` bila pohon memiliki kurang dari k simpul. Berapa waktu jalannya?

11.10 Tulislah `Node *buildBalanced(const std::vector<int> &a, int lo, int hi)`, yang membangun BST bertinggi minimum dari kunci terurut `a[lo..hi]`. Berapa waktu jalannya, dan berapa tinggi yang dihasilkannya untuk $n = 7$, 1000, dan 1,000,000 kunci?

11.11 *Leluhur bersama terendah* (*lowest common ancestor*) dua kunci x dan y dalam BST adalah simpul terdalam yang memuat keduanya di subpohonnya (sebuah simpul terhitung berada di subpohonnya sendiri). Tulislah fungsi yang menemukannya dalam waktu $O(h)$ tanpa rekursi, dan sebutkan leluhur bersama terendah dari (35, 45), (35, 65), (20, 35), dan (60, 65) pada pohon Soal 11.7.

11.12 Gambarlah setiap *binary search tree* yang memuat kunci 1, 2, 3, dan untuk masing-masing daftarlah urutan penyisipan yang menghasilkannya. Periksalah Persamaan (11.11) untuk $n = 3$.

11.13 Bangunlah pohon ekspresi dari ekspresi postfiks `8 2 / 3 - 4 5 * +` dengan **Pseudokode 11.4**, sambil menggambar *stack* sesudah setiap simbol. Sebutkan bentuk prefiks dan infiks berkurung lengkapnya serta nilainya.

11.14 `bst.cpp` (**Kode 11.4**) bekerja untuk pohon-pohon dalam bab ini. Apa yang terjadi bila `main`-nya menyisipkan kunci 1 sampai 1.000.000 dalam urutan naik lalu memanggil `height`? Taksirlah memori *stack* yang diperlukan bila setiap bingkai panggilan memakan 48 bita, dan bandingkan dengan batas lazim 8 MB di Linux. Apa yang terjadi pada `bst.py` dengan 2.000 kunci naik? Usulkan dua perbaikan.

11.15 Untuk setiap tugas Pustaka Ceria, pilihlah struktur di antara larik terurut dengan *binary search* (Bab 2), tabel *hash* (`std::unordered_map`, Bab 15), dan pohon pencarian seimbang (`std::map`), lalu beri alasannya: (a) mencari buku berdasarkan kode tepatnya, berkali-kali per detik, dengan penambahan yang sering; (b) menampilkan katalog dalam urutan abjad judul sesudah setiap penambahan; (c) mendaftar semua judul yang diawali “Ne”; (d) daftar acuan tetap berisi 5.000 tajuk subjek, dimuat setahun sekali dan dicari terus-menerus.

11.16 Sisipkan 10, 20, 15 ke pohon AVL kosong. Di simpul mana faktor keseimbangan menjadi ± 2, mengapa satu rotasi tidak cukup, dan dua rotasi mana yang memperbaikinya? Gambarlah hasilnya.

11.16 Penyelesaian Soal Latihan

11.1 Akarnya P. Simpul internal: P, Q, U, S; daun: T, W, R, V. Anak-anak Q adalah T dan U; saudara-saudara R adalah Q dan S; leluhur W adalah U, Q, dan P; keturunan Q adalah T, U, dan W. U berkedalaman 2; Q bertinggi 2 (lintasan Q, U, W). Pohon itu bertinggi 3 (P, Q, U, W) dan berderajat 3 (P memiliki tiga anak). Ia memiliki 9 simpul dan, menurut Persamaan (11.1), 8 sisi.

11.2 Tingkat 0: P. Tingkat 1: Q, R, S. Tingkat 2: T, U, V. Tingkat 3: W. Dalam diagram Venn, daerah P memuat tiga daerah saling lepas Q, R, dan S; Q memuat T dan daerah U, yang memuat W; S memuat V; R adalah satu daerah kecil. Dalam notasi tingkat, satu simpul per baris dalam urutan *preorder*, menjorok sesuai kedalaman: P; Q (satu langkah); T dan U (dua langkah); W (tiga langkah); R (satu langkah); S (satu langkah); V (dua langkah). Notasi kurungnya adalah yang diberikan, `(P(Q(T,U(W)),R,S(V)))`.

11.3 40 menjadi akar; 20 ke kiri 40 dan 60 ke kanan; 10 ke kiri 20, 30 ke kanan 20, 50 ke kiri 60, dan 70 ke kanan 60. Hasilnya pohon *perfect* bertinggi 2. *Preorder*: 40 20 10 30 60 50 70. *Inorder*: 10 20 30 40 50 60 70. *Postorder*: 10 30 20 50 70 60 40. Urutan tingkat: 40 20 60 10 30 50 70. Dalam urutan naik kunci-kunci itu membangun rantai condong ke kanan bertinggi 6.

11.4 Lariknya 40, 20, 60, 10, 30, 50, 70 di slot 0 sampai 6. Slot 5 memuat 50; induknya berada di $\lfloor (5-1)/2 \rfloor = 2$, yang memuat 60, benar. Pohon `insert(65, 5, 70, 3, 10)` tersimpan sebagai

65, 5, 70, 3, 10 di slot 0 sampai 4 tanpa slot kosong, karena pohon itu *complete*: tingkat terakhirnya (3, 10) terisi dari kiri.

11.5 (a) Pohon itu memiliki $n = n_0 + n_1 + n_2$ simpul dan, dengan menghitung sisi setiap simpul ke anak-anaknya, $n_1 + 2n_2$ sisi. Menurut Persamaan (11.1) banyaknya $n - 1$, sehingga $n_0 + n_1 + n_2 - 1 = n_1 + 2n_2$, yang menghasilkan $n_0 = n_2 + 1$. Persamaan (11.7) adalah kasus khusus $n_1 = 0$. (b) Tingkat 0 sampai $h - 1$ penuh, yaitu $2^h - 1$ simpul menurut Persamaan (11.5), dan tingkat terakhir memuat antara 1 dan 2^h simpul, sehingga $2^h \le n \le 2^{h+1} - 1$. Dengan mengambil logaritma, $h \le \lg n < h + 1$, sehingga $h = \lfloor \lg n \rfloor$.

11.6 Kunci *preorder* pertama, A, adalah akarnya. Dalam daftar *inorder*, kunci-kunci di kiri A (D G B) membentuk subpohon kiri dan yang di kanannya (E C H F) subpohon kanan. Ulangi pada setiap bagian: kunci *preorder* berikutnya, B, adalah akar bagian kiri, dan D G berada di kirinya dalam *inorder*, sehingga B hanya memiliki subpohon kiri, berakar kunci *preorder* berikutnya, D, yang memiliki G di kanannya. Di sebelah kanan, C adalah akar dengan E di kirinya dan H F di kanannya; F adalah akar bagian itu dengan H di kirinya. Jadi A beranak B dan C; B beranak kiri D; D beranak kanan G; C beranak E dan F; F beranak kiri H. *Postorder*-nya G D B E H F C A, dan tingginya 3. Fungsi `buildPreIn` pada **Kode 11.16** melakukan persis hal ini: sebuah indeks bersama menyusuri daftar *preorder*, dan posisi setiap akar dalam daftar *inorder* membelah kunci-kunci yang tersisa. Program itu mencetak *postorder* dan tinggi pohon yang dibangun kembali, dan menegaskan penyelesaian dengan tangan tadi.

Kode 11.16 `exercises.cpp`: penyelesaian Soal 11.6 serta 11.8 sampai 11.11.

```cpp
// exercises.cpp -- solutions to the programming exercises of Chapter 11.
#include <climits>
#include <cstdio>
#include <string>
#include <vector>

struct Node {
    int data;
    Node *left, *right;
};

Node *insert(Node *t, int x) {
    if (t == nullptr) return new Node{x, nullptr, nullptr};
    if (x < t->data) t->left = insert(t->left, x);
    else if (x > t->data) t->right = insert(t->right, x);
    return t;
}

int height(const Node *t) {
    if (t == nullptr) return -1;
    int a = height(t->left), b = height(t->right);
    return (a > b ? a : b) + 1;
}

void destroy(Node *t) {
    if (t == nullptr) return;
    destroy(t->left);
    destroy(t->right);
    delete t;
}

void postOrderChars(const Node *t) {
    if (t == nullptr) return;
    postOrderChars(t->left);
    postOrderChars(t->right);
    std::printf("%c ", (char) t->data);
}
```

Kode 11.16 (Lanjutan) *exercises.cpp: penyelesaian Soal 11.6 serta 11.8 sampai 11.11.*

```
// Exercise 11.6: rebuild a tree from its preorder and inorder listings.
Node *buildPreIn(const std::string &pre, const std::string &in, int &p,
                 int lo, int hi) {
    if (lo > hi) return nullptr;            // no keys for this subtree
    char rootKey = pre[p++];                // the next preorder key is this root
    int m = (int) in.find(rootKey, lo);     // inorder: left | root | right
    Node *t = new Node{rootKey, nullptr, nullptr};
    t->left = buildPreIn(pre, in, p, lo, m - 1);
    t->right = buildPreIn(pre, in, p, m + 1, hi);
    return t;
}

// Exercise 11.8: is every key inside the open interval its ancestors allow?
bool isBST(const Node *t, long long lo, long long hi) {
    if (t == nullptr) return true;
    if (t->data <= lo || t->data >= hi) return false;
    return isBST(t->left, lo, t->data) && isBST(t->right, t->data, hi);
}

bool isBSTLocalOnly(const Node *t) {        // the tempting but WRONG check
    if (t == nullptr) return true;
    if (t->left != nullptr && t->left->data > t->data) return false;
    if (t->right != nullptr && t->right->data < t->data) return false;
    return isBSTLocalOnly(t->left) && isBSTLocalOnly(t->right);
}

// Exercise 11.9: the k-th smallest key (k counts from 1), by an inorder walk.
const Node *kthSmallest(const Node *t, int &k) {
    if (t == nullptr) return nullptr;
    const Node *r = kthSmallest(t->left, k);
    if (r != nullptr) return r;
    if (--k == 0) return t;                 // t is the k-th node visited
    return kthSmallest(t->right, k);
}

// Exercise 11.10: a balanced BST from sorted keys a[lo..hi].
Node *buildBalanced(const std::vector<int> &a, int lo, int hi) {
    if (lo > hi) return nullptr;
    int mid = lo + (hi - lo) / 2;           // the middle key becomes the root
    return new Node{a[mid], buildBalanced(a, lo, mid - 1),
                    buildBalanced(a, mid + 1, hi)};
}
```

Kode 11.16 (Lanjutan) *exercises.cpp: penyelesaian Soal 11.6 serta 11.8 sampai 11.11.*

```
// Exercise 11.11: lowest common ancestor of keys x and y in a BST.
const Node *lca(const Node *t, int x, int y) {
    while (t != nullptr) {
        if (x < t->data && y < t->data) t = t->left;         // both on the left
        else if (x > t->data && y > t->data) t = t->right;   // both on the right
        else return t;                                       // they split here
    }
    return nullptr;
}

int main() {
    int p = 0;
    std::string pre = "ABDGCEFH", in = "DGBAECHF";
    Node *t = buildPreIn(pre, in, p, 0, (int) in.size() - 1);
    std::printf("11.6  rebuilt tree, postorder: ");
    postOrderChars(t);
    std::printf("(height %d)\n", height(t));
    destroy(t);

    int keys[] = {50, 30, 70, 20, 40, 60, 80, 35, 45, 65};
    Node *b = nullptr;
    for (int x : keys) b = insert(b, x);
    Node *bad = new Node{10, new Node{5, new Node{2, nullptr, nullptr},
                                         new Node{12, nullptr, nullptr}},
                             new Node{15, nullptr, nullptr}};
    std::printf("11.8  tree B: isBST %s; bad tree: isBST %s, local check says %s\n",
                isBST(b, LLONG_MIN, LLONG_MAX) ? "true" : "false",
                isBST(bad, LLONG_MIN, LLONG_MAX) ? "true" : "false",
                isBSTLocalOnly(bad) ? "true" : "false");
    destroy(bad);

    std::printf("11.9  tree B:");
    for (int k : {1, 4, 10, 11}) {
        int kk = k;
        const Node *r = kthSmallest(b, kk);
        if (r != nullptr) std::printf(" k=%d -> %d;", k, r->data);
        else std::printf(" k=%d -> none", k);
    }
    std::printf("\n");
```

Kode 11.16 (Lanjutan) *exercises.cpp: penyelesaian Soal 11.6 serta 11.8 sampai 11.11.*

```
    for (int n : {7, 1000, 1000000}) {
        std::vector<int> a(n);
        for (int i = 0; i < n; ++i) a[i] = i + 1;
        Node *r = buildBalanced(a, 0, n - 1);
        std::printf("11.10 n=%d: root %d, height %d\n", n, r->data, height(r));
        destroy(r);
    }

    std::printf("11.11 tree B:");
    int pairs[][2] = {{35, 45}, {35, 65}, {20, 35}, {60, 65}};
    for (auto &q : pairs)
        std::printf(" lca(%d,%d)=%d", q[0], q[1], lca(b, q[0], q[1])->data);
    std::printf("\n");
    destroy(b);
    return 0;
}
```

```
11.6  rebuilt tree, postorder: G D B E H F C A (height 3)
11.8  tree B: isBST true; bad tree: isBST false, local check says true
11.9  tree B: k=1 -> 20; k=4 -> 40; k=10 -> 80; k=11 -> none
11.10 n=7: root 4, height 2
11.10 n=1000: root 500, height 9
11.10 n=1000000: root 500000, height 19
11.11 tree B: lca(35,45)=40 lca(35,65)=50 lca(20,35)=30 lca(60,65)=60
```

Untuk bagian kedua: pohon "A dengan anak kiri B" dan "A dengan anak kanan B" sama-sama memiliki *preorder* A B dan *postorder* B A, sehingga kedua daftar itu tidak dapat membedakannya. (*Inorder* dapat: B A untuk yang pertama dan A B untuk yang kedua.)

11.7 Menghapus 30 (dua anak: 20, dan 40 yang beranak 35 dan 45). *Varian successor:* minimum subpohon kanan adalah 35, sebuah daun; 35 menggantikan 30 dan daun itu dibuang, sehingga subpohon kiri 50 menjadi 35 dengan anak 20 dan 40, dan 40 hanya menyisakan anak kanan 45. *Varian predecessor:* maksimum subpohon kiri adalah 20, sebuah daun; 20 menggantikan 30 dan daun itu dibuang, sehingga 20 tidak memiliki anak kiri dan beranak kanan 40 (dengan 35 dan 45). Menghapus 70 dari pohon semula (dua anak: 60 yang beranak kanan 65, dan 80). *Varian successor: successor*-nya 80, sebuah daun; 80 menggantikan 70 dan posisi kanan lama 70 menjadi kosong, sehingga 80 beranak kiri 60, yang tetap beranak kanan 65. *Varian predecessor: predecessor*-nya 65, maksimum subpohon kiri, sebuah daun; 65 menggantikan 70, dan 65 beranak 60 dan 80. Keempat hasil memiliki barisan *inorder* yang terurut.

11.8 `isBST` pada **Kode 11.16** menurunkan selang terbuka (lo, hi) yang harus memuat setiap kunci subpohon saat ini; belok ke kiri mempersempit *hi* menjadi kunci simpul itu, belok ke kanan mempersempit *lo*. Pohon 10(5(2, 12), 15) gagal di 12: ia berada di subpohon kiri 10, sehingga harus di bawah 10, tetapi nilainya 12. Pemeriksaan lokal hanya melihat bahwa $12 > 5$ (anak kanan 5 yang benar) dan $5 < 10$ (anak kiri 10 yang benar), lalu menerima pohon itu; keluaran memperlihatkan `isBST false` dan `local check says true`. Tipe `long long` dengan `LLONG_MIN` dan `LLONG_MAX` membuat batas awalnya lebih lebar daripada kunci `int` mana pun.

11.9 `kthSmallest` melakukan penelusuran *inorder* dan menghitung kunjungan mundur dari k; simpul tempat hitungan mencapai 0 adalah jawabannya, dan pencarian segera berhenti. Ia mengunjungi k kunci terkecil ditambah simpul-simpul pada lintasan menuju kunci-kunci itu, sehingga waktunya $O(h + k)$, dan $\Theta(n)$ dalam kasus terburuk. Pada Pohon B keluaran memberikan 20, 40, 80 untuk $k = 1,4,10$ dan tidak ada simpul untuk $k = 11$. (Menyimpan ukuran subpohon di setiap simpul mengurangi waktunya menjadi $O(h)$, gagasan *order-statistic tree* pada Cormen dkk. [2].)

11.10 `buildBalanced` menjadikan kunci tengah sebagai akar dan membangun kedua separuhnya secara rekursif; setiap kunci ditempatkan sekali dengan kerja $O(1)$, sehingga waktunya $\Theta(n)$ sesudah kunci-kunci diurutkan. Rekursinya selalu membagi dua, sehingga kedalamannya sekitar $\lg n$, dan pohonnya memiliki tinggi minimum menurut Persamaan (11.6): keluaran memperlihatkan tinggi 2 untuk 7 kunci, 9 untuk 1.000, dan 19 untuk 1.000.000, yaitu $\lfloor \lg n \rfloor$.

11.11 Berjalanlah turun dari akar: bila kedua kunci lebih kecil daripada kunci saat ini, leluhur bersamanya berada di subpohon kiri; bila keduanya lebih besar, di subpohon kanan; selain itu kedua kunci berada di sisi yang berbeda (atau salah satunya adalah kunci saat ini), dan simpul saat ini adalah leluhur bersama terendahnya. Cara ini mengikuti satu lintasan, $O(h)$, dan tidak memerlukan rekursi. Keluaran memberikan lca(35, 45) = 40, lca(35, 65) = 50, lca(20, 35) = 30, dan lca(60, 65) = 60 (65 berada di subpohon 60).

11.12 Ada lima pohon. 1, 2, 3 menghasilkan rantai ke kanan; 3, 2, 1 rantai ke kiri; 1, 3, 2 menghasilkan 1 dengan anak kanan 3, yang beranak kiri 2; 3, 1, 2 menghasilkan 3 dengan anak kiri 1, yang beranak kanan 2; dan baik 2, 1, 3 maupun 2, 3, 1 menghasilkan pohon *perfect* berakar 2. Empat dari enam urutan menghasilkan pohon bertinggi $n-1=2$, dan $2^{3-1}=4$, sesuai Persamaan (11.11). (Secara umum banyak pohon biner dengan n simpul adalah bilangan Catalan $\frac{1}{n+1}\binom{2n}{n}$, yang bernilai 5 untuk $n=3$.)

11.13 Membaca `8 2 / 3 - 4 5 * +`: dorong 8, dorong 2; `/` mengambil 2 (kanan) dan 8 (kiri) lalu mendorong pohon 8 / 2; dorong 3; `-` menggabungkan (8 / 2) dan 3; dorong 4, dorong 5; `*` menggabungkan 4 dan 5; `+` menggabungkan ((8 / 2) - 3) dan (4 * 5). Akarnya `+`, dengan subpohon kiri `-` (yang subpohon kirinya `/` atas 8 dan 2, dan anak kanannya 3) dan subpohon kanan `*` atas 4 dan 5. Prefiks: `+ - / 8 2 3 * 4 5`. Infiks: `( ( ( 8 / 2 ) - 3 ) + ( 4 * 5 ) )`. Nilainya: $8/2=4$, $4-3=1$, $4\cdot 5=20$, $1+20=21$.

11.14 Kunci naik membangun rantai ke kanan bertinggi 999.999. Setiap `insert` lalu merekursi sekali per tingkat, sehingga penyisipan-penyisipan terakhir, serta `height`, `count`, dan `clear`, menumpuk sampai sejuta panggilan. Dengan 48 bita per bingkai, itu sekitar 48 MB *stack*, enam kali batas lazim 8 MB di Linux (dan 48 kali bawaan 1 MB di Windows), sehingga program mogok karena *stack overflow* (Subbab 10.12.1); jauh sebelum itu, waktu penyisipan $\Theta(n^2)$ pada Subbab 11.8.2 akan memakan berjam-jam. Pada `bst.py` batas rekursi sekitar 1.000 tercapai kira-kira pada kunci ke-1.000, dan Python memunculkan `RecursionError`. Perbaikannya: tulis penyisipan, pencarian, dan tinggi secara iteratif (seperti `height_experiment.cpp`); atau, lebih baik, jaga pohon tetap seimbang (pohon AVL, atau `std::map`), sehingga kedalamannya tidak pernah melebihi sekitar $1.44\lg n$.

11.15 (a) Tabel *hash*: pencarian tepat dalam waktu rata-rata $O(1)$ dan penambahan yang murah; tidak ada bagian tugas itu yang memerlukan urutan. (b) Pohon pencarian seimbang: setiap penambahan $O(\lg n)$ dan daftar abjadnya adalah perjalanan *inorder*; larik terurut akan membayar pergeseran $\Theta(n)$ per penambahan, dan tabel *hash* harus mengurutkan untuk setiap daftar. (c) Pohon pencarian seimbang (atau larik terurut): judul-judul yang diawali "Ne" adalah judul dari `lower_bound("Ne")` sampai sebelum `lower_bound("Nf")`, dua pencarian $O(\lg n)$ ditambah satu langkah per hasil; tabel *hash* tidak dapat menjawabnya tanpa memindai semuanya. (d) Larik terurut dengan *binary search*: datanya tidak pernah berubah, sehingga ongkos penyisipan tidak relevan, dan larik tidak memiliki beban pointer serta paling ramah *cache* (Subbab 7.3.2).

11.16 Sesudah menyisipkan 10 dan 20, pohonnya 10 dengan anak kanan 20. Menyisipkan 15 berjalan ke kanan 10 lalu ke kiri 20. Sekarang simpul 10 memiliki subpohon kiri bertinggi -1 dan subpohon kanan bertinggi 1: faktor keseimbangan -2. Bagian yang dalam, 15, berada di sisi *dalam* (subpohon kiri dari anak kanan 10), sehingga satu rotasi kiri di 10 hanya memindahkan masalahnya: 20 akan menjadi akar dengan anak kiri 10, dan 15 menjadi anak kanan 10, menghasilkan faktor keseimbangan $+2$ di 20. Perbaikannya adalah rotasi ganda pada baris "*right-left*" di **Kode 11.9**: pertama rotasi kanan di 20, yang menjadikan 15 anak kanan 10 dengan anak kanan 20 (sebuah rantai lurus), lalu rotasi kiri di 10. Hasilnya 15 dengan anak 10 dan 20, bertinggi 1.

Daftar Pustaka

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, sec. 2.3.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 12–13.

[3] P. F. Windley, "Trees, forests and rearranging," *The Computer Journal*, vol. 3, no. 2, pp. 84–88, 1960.

[4] T. N. Hibbard, "Some combinatorial properties of certain trees with applications to searching and sorting," *Journal of the ACM*, vol. 9, no. 1, pp. 13–28, Jan. 1962.

[5] G. M. Adelson-Velsky and E. M. Landis, "An algorithm for the organization of information," *Soviet Mathematics Doklady*, vol. 3, pp. 1259–1263, 1962.

[6] R. Bayer, "Symmetric binary B-trees: Data structure and maintenance algorithms," *Acta Informatica*, vol. 1, no. 4, pp. 290–306, 1972.

[7] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. 19th Annual Symposium on Foundations of Computer Science*, Ann Arbor, MI, USA, Oct. 1978, pp. 8–21.

[8] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998, secs. 6.2.2–6.2.3.

[9] J. L. Eppinger, "An empirical study of insertion and deletion in binary search trees," *Communications of the ACM*, vol. 26, no. 9, pp. 663–669, Sep. 1983.

[10] B. Reed, "The height of a random binary search tree," *Journal of the ACM*, vol. 50, no. 3, pp. 306–332, May 2003.

[11] GNU Project, "libstdc++-v3/include/bits/stl_tree.h," GCC source repository. [Online]. Available: https://gcc.gnu.org/git/?p=gcc.git;a=blob;f=libstdc%2B%2B-v3/include/bits/stl_tree.h. [Accessed: 24-Sep-2026].

[12] Oracle, "Class TreeMap<K,V>," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/TreeMap.html. [Accessed: 24-Sep-2026].

[13] The Linux Kernel Documentation, "Red-black Trees (rbtree) in Linux." [Online]. Available: https://docs.kernel.org/core-api/rbtree.html. [Accessed: 24-Sep-2026].

[14] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173–189, 1972.

[15] PostgreSQL Global Development Group, "B-Tree Indexes," *PostgreSQL Documentation* (current version). [Online]. Available: https://www.postgresql.org/docs/current/btree.html. [Accessed: 24-Sep-2026].

[16] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM*, vol. 32, no. 3, pp. 652–686, Jul. 1985.

[17] R. Seidel and C. R. Aragon, "Randomized search trees," *Algorithmica*, vol. 16, no. 4–5, pp. 464–497, 1996.

[18] Python Software Foundation, "bisect — Array bisection algorithm," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/bisect.html. [Accessed: 24-Sep-2026].

[19] G. Jenks, "Sorted Containers," version 2.4.0. [Online]. Available: https://grantjenks.com/docs/sortedcontainers/ and https://pypi.org/project/sortedcontainers/. [Accessed: 24-Sep-2026].

[20] Profitina, "Profitina — AI Visibility (GEO) platform." [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

[21] WHATWG, "DOM Living Standard." [Online]. Available: https://dom.spec.whatwg.org/. [Accessed: 24-Sep-2026].

Bab 12

Evaluasi dan Pembahasan III: Rekursi Bekerja pada List dan Pohon

Bab ini tidak menambahkan struktur baru. Bab ini mempekerjakan *double linked list* dan list sirkular dari Bab 9, rekursi dari Bab 10, serta pohon dan *binary search tree* dari Bab 11 pada enam belas soal, masing-masing diselesaikan dari halaman kosong, selangkah kecil demi selangkah, dalam C++, Java, dan Python. Sepuluh di antaranya memakai satu pohon kecil yang sama: menelusurinya secara *depth first*, mencetak satu tingkatnya, menjumlahkan dan melipattigakan nilainya, mencerminkannya, memeriksa keseimbangannya, menghapus sebuah nilai, mencari induk dan anak sebuah simpul, dan meratakan seluruh pohon menjadi *double linked list* tanpa mengalokasikan satu simpul pun. Enam soal lainnya baru: membalik *double linked list*, lingkaran Josephus, memilih buku dengan *backtracking*, memeriksa apakah sebuah pohon benar-benar *binary search tree*, leluhur bersama terendah, dan mencetak pohon tingkat demi tingkat. Untuk setiap soal Anda akan menggambar pohonnya, menelusuri rekursinya dengan tangan, membaca tiga program lengkap beserta keluarannya yang sesungguhnya, menyaksikan sebuah penyelesaian keliru yang menggoda gagal (untuk *bug* memori, dalam laporan pemeriksa memori itu sendiri), dan menutupnya dengan varian *uji diri* beserta jawabannya.

12.1 Titik Tolak Kita

Tiga bab telah kita lalui sejak bab sejenis yang terakhir, dan masing-masing memberi kita satu alat.

Bab 9 memberi setiap simpul tautan kedua. Dalam *double linked list*, setiap simpul menunjuk ke simpul sesudahnya (`next`) dan ke simpul sebelumnya (`prev`), sehingga list dapat ditelusuri ke dua arah dan sebuah simpul dapat dibuang walaupun yang kita pegang hanyalah pointer ke simpul itu (Subbab 9.2). Kita menyimpan `head` dan `tail` (Subbab 9.4), menulis penyisipan dan penghapusan di kedua ujung, dan berkenalan dengan list *sirkular*, yang simpul terakhirnya menunjuk kembali ke simpul pertama, sehingga list itu sama sekali tidak berujung; yang ada hanyalah tempat yang kita sepakati sebagai awal (Subbab 9.6). Setiap operasi datang bersama aturan dua panahnya: setiap kali sebuah panah maju berubah, panah mundur pasangannya harus ikut berubah.

Bab 10 mengajarkan *rekursi*: fungsi yang menyelesaikan suatu soal dengan memanggil dirinya sendiri pada contoh yang lebih kecil dari soal yang sama. Kita mengenal dua bagian yang dimiliki setiap fungsi rekursif, yaitu *base case* (kasus dasar), yang dijawab langsung, dan *recursive case* (kasus rekursif), yang memperkecil soal dan memercayai bahwa pemanggilan yang lebih kecil memberi jawaban yang benar (Subbab 10.2). Kita menelusuri faktorial, Fibonacci, faktor persekutuan terbesar, dan Menara Hanoi, menyaksikan *call stack* membesar dan mengecil (Subbab 10.3), melihat bagaimana Fibonacci naif mengulang pekerjaan yang sama secara eksponensial, dan menulis relasi rekurens seperti $T(n) = 2T(n-1) + 1$ untuk menghitung banyaknya langkah Hanoi (Subbab 10.8).

Bab 11 menyatukan kedua gagasan itu. *Pohon* (*tree*) adalah struktur simpul yang setiap simpulnya, kecuali satu, yaitu *akar* (*root*), memiliki tepat satu *induk* (*parent*), dan *pohon biner* adalah pohon yang setiap simpulnya memiliki paling banyak dua anak, *kiri* dan *kanan*. *Binary search tree* (BST) menambahkan satu aturan tentang nilai: semua nilai di subpohon kiri sebuah simpul lebih kecil daripada simpul itu, dan semua nilai di subpohon kanannya lebih besar. Kita menulis `insert`, tiga penelusuran `preOrder`, `inOrder`, dan `postOrder` (Subbab 11.6.1), penelusuran *level order* dengan queue (Subbab 11.6.3), `search`, `count`, `height`, `min`, `max`, dan `leaf`, hampir semuanya rekursif (Subbab 11.7), karena pohon itu sendiri adalah benda rekursif: sebuah akar dengan pohon kiri dan pohon kanan yang tergantung padanya.

Bab ini memakai ketiga alat itu, sering kali di dalam fungsi yang sama. **Tabel 12.1** mendaftar keenam belas soal, subbab yang menyelesaikan masing-masing, dan bab yang menyumbangkan gagasan utamanya.

Tabel 12.1 Enam belas soal bab ini, subbab yang menyelesaikan masing-masing, dan bab yang menyumbangkan gagasan utamanya.

Soal	Subbab	Yang diminta	Alat utama dari
12.1	12.4	Mencetak penelusuran *depth first*, cabang kiri lebih dulu	Bab 10, 11
12.2	12.5	Mencetak setiap simpul pada tingkat *k*	Bab 10, 11
12.3	12.6	Menjumlahkan semua nilai di pohon	Bab 10, 11
12.4	12.7	Mengalikan setiap nilai dengan tiga	Bab 10, 11
12.5	12.8	Mengubah pohon menjadi bayangan cerminnya	Bab 10, 11
12.6	12.9	Memutuskan apakah setiap simpul seimbang tingginya	Bab 10, 11
12.7	12.10	Menghapus sebuah nilai dari *binary search tree*	Bab 11
12.8	12.11	Mencari induk simpul yang memuat sebuah nilai	Bab 11
12.9	12.12	Melaporkan anak-anak simpul yang memuat sebuah nilai	Bab 11
12.10	12.13	Meratakan pohon menjadi *double linked list*, di tempat	Bab 9, 10, 11
12.11	12.14	Membalik *double linked list* di tempat	Bab 9
12.12	12.14	Lingkaran Josephus: siapa yang tersisa?	Bab 9, 10
12.13	12.15	Semua cara memilih *k* dari *n* buku	Bab 10
12.14	12.16	Apakah pohon biner ini *binary search tree*?	Bab 10, 11
12.15	12.16	Leluhur bersama terendah dua nilai	Bab 11
12.16	12.16	Mencetak pohon tingkat demi tingkat	Bab 5, 11

Sepanjang bab ini kita mempertahankan contoh berjalan buku ini, katalog perpustakaan kecil *Pustaka Ceria*, yang bukunya memiliki pengenal seperti `B1001`. Sepuluh soal pertama memakai satu pohon bilangan bulat yang kecil, yaitu pohon yang diperoleh dengan menyisipkan 6, 3, 10, 1, 5, 7, 8, dan 9 ke dalam *binary search tree* kosong, karena inti soal-soal itu adalah menyaksikan banyak fungsi yang berbeda menelusuri gambar yang sama.

> **WAWASAN KUNCI: SATU GAGASAN DI BALIK HAMPIR SETIAP SOAL.** *Sebuah pohon itu kosong, atau berupa akar dengan pohon kiri dan pohon kanan; maka jawablah pohon kosong secara langsung, ajukan pertanyaan yang sama kepada kedua pohon yang lebih kecil, lalu gabungkan kedua jawabannya dengan akar.* Jumlah sebuah pohon adalah akar ditambah jumlah kedua pohonnya; cerminnya adalah akar dengan kedua pohonnya ditukar dan dicerminkan; tingginya satu lebih besar daripada yang lebih tinggi di antara kedua pohonnya. Ketika sebuah program dalam bab ini salah, salah satu dari ketiga bagian itu dilanggar: pohon kosong dijawab keliru (Soal 12.1 dan 12.6), pertanyaannya berubah dalam perjalanan turun

(Soal 12.2), atau jawaban pohon yang lebih kecil dibuang alih-alih digabungkan (Soal 12.3 dan 12.7).

12.2 Cara Menghadapi Soal Rekursif pada Struktur Bertaut

Penyelesaian rekursif pada pohon atau list adalah sebuah janji: "berikan aku jawaban untuk potongan-potongan yang lebih kecil, dan aku akan membangun jawaban untuk keseluruhannya". Kodenya sering hanya lima baris; kesulitannya terletak pada memercayai janji itu, dan pada menjaga agar pointer tetap benar ketika fungsi mengubah strukturnya. Metodenya terdiri atas tujuh langkah.

Langkah 1 — Nyatakan kembali soalnya dengan kata-kata sendiri, disertai satu contoh. Tuliskan apa yang masuk, apa yang keluar, dan satu pohon atau list kecil beserta jawabannya yang dikerjakan dengan tangan.

Langkah 2 — Tetapkan apa yang dikembalikan fungsi untuk struktur kosong. Inilah *base case*, tempat kebanyakan fungsi rekursif salah. Jumlah dari tidak ada nilai adalah 0; tinggi dari tidak ada pohon adalah -1 (supaya satu simpul bertinggi 0, Subbab 11.2.3); pohon kosong *adalah* pohon seimbang dan *adalah binary search tree*; mencetaknya tidak mencetak apa pun. Periksalah bahwa jawaban itu membuat langkah penggabungan pada Langkah 4 benar untuk satu simpul.

Langkah 3 — Gambarlah pohonnya, dan tandai apa yang diterima setiap pemanggilan. Di samping setiap simpul, tuliskan argumen yang diterima pemanggilannya: tingkat yang masih harus ditempuh, batas-batasnya, simpul sebelumnya dalam list. Anak yang menerima pertanyaan yang salah lalu tampak sekilas pandang.

Langkah 4 — Tuliskan kasus rekursif sebagai "akar, jawaban kiri, jawaban kanan". Andaikan bahwa pemanggilan pada kedua subpohon mengembalikan jawaban yang benar, yaitu *recursive leap of faith* (lompatan keyakinan rekursif) pada Subbab 10.2, yang dibenarkan oleh induksi [1], lalu tuliskan hanya cara simpul itu menggabungkannya. Tentukan *kapan* simpul itu mengerjakan bagiannya sendiri: sebelum pemanggilan (*preorder*, seperti ketika mencetak urutan *depth first*), di antaranya (*inorder*, seperti ketika meratakan BST menjadi urutan terurut), atau sesudahnya (*postorder*, seperti ketika membebaskan pohon).

Langkah 5 — Jika fungsi mengubah tautan, buatlah fungsi itu mengembalikan akar yang baru. Fungsi yang mungkin menghapus atau mengganti simpul yang diterimanya mengembalikan apa yang kini berdiri di situ, dan induknya memasangnya kembali: `root->left = f(root->left)`. Soal 12.7 dan 12.10 bergantung pada aturan ini, dan penyelesaian kelirunya melanggarnya.

Langkah 6 — Daftarlah kasus-kasus tepi. Untuk pohon: pohon kosong, satu simpul, simpul yang hanya punya anak kiri atau hanya anak kanan, nilai yang tidak ada, dan rantai panjang. Untuk list: kosong, satu simpul, simpul pertama, simpul terakhir.

Langkah 7 — Ujilah, termasuk di bawah pemeriksa memori. Jalankan contohnya, lalu setiap kasus tepi; dalam C++, jalankan sekali lagi dengan *sanitizer*, karena fungsi yang membaca simpul yang sudah dibebaskan bisa saja mencetak jawaban yang benar secara kebetulan.

Pseudokode 12.1 menuliskan Langkah 2, 4, dan 5 sebagai sebuah templat. Hampir setiap fungsi pohon dalam bab ini adalah templat ini dengan ketiga bagian kosongnya diisi.

Pseudokode 12.1 Templat fungsi pohon rekursif. Mengisi bagian kosongnya dengan 0 dan $T.key + a + b$ menghasilkan jumlah pada Soal 12.3; dengan -1 dan $1 + \max(a, b)$ menghasilkan tinggi.

```
TREE-FUNCTION(T)
1  if T = NIL
2      return ANSWER-FOR-THE-EMPTY-TREE          // Langkah 2: base case
3  a = TREE-FUNCTION(T.left)                     // percayai pemanggilan kecil
4  b = TREE-FUNCTION(T.right)                    // percayai pemanggilan kecil
5  return COMBINE(T.key, a, b)                   // Langkah 4: akar, kiri, kanan
```

Berapa biaya fungsi seperti itu? Setiap pemanggilan pada pohon tak kosong mengerjakan sejumlah tetap pekerjaannya sendiri, katakanlah c langkah, dan melakukan dua pemanggilan pada subpohon; setiap pemanggilan pada pohon kosong mengerjakan sejumlah tetap c_0. Jika pohon memiliki n simpul, terdapat tepat n pemanggilan pada pohon tak kosong, satu per simpul, dan tepat $n + 1$ pemanggilan pada pohon kosong, karena setiap simpul memiliki dua pointer anak, sehingga ada $2n$ pointer, dan $n - 1$ di antaranya menunjuk ke simpul sungguhan (setiap simpul kecuali akar ditunjuk oleh satu induk), sehingga $2n - (n - 1) = n + 1$ di antaranya bernilai `nullptr`. Jadi biaya totalnya

$$T(n) = c\,n + c_0\,(n + 1) = \Theta(n). \tag{12.1}$$

Persamaan (12.1) adalah alasan hampir setiap fungsi dalam bab ini linear: fungsi itu menyentuh setiap simpul sekali dan setiap anak kosong sekali. Pengecualiannya patut diperhatikan ketika kita menjumpainya: `printKthLevel` berhenti lebih awal, `search` dan fungsi-fungsi BST hanya menempuh satu lintasan, dan pemeriksaan keseimbangan pada Soal 12.6 memanggil `height` berulang-ulang.

Tabel 12.2 meringkas metode itu dalam satu halaman. Setiap soal di bawah menerapkannya, dan label tebal di dalam setiap soal mengikuti urutannya.

Tabel 12.2 Metode tujuh langkah untuk soal rekursif pada list atau pohon. Langkah 1 dan 2 memastikan jawabannya, Langkah 3 sampai 5 merancang rekursinya, dan Langkah 6 dan 7 memeriksanya.

Langkah	Tulis di kertas	Pertanyaan yang dijawabnya
1	Soal dengan kata-kata sendiri, dan satu contoh dengan tangan	Apakah saya tahu keluaran yang harus dihasilkan?
2	Jawaban untuk pohon atau list kosong	Apa *base case*-nya, dan benarkah untuk satu simpul?
3	Pohon, dengan argumen yang diterima setiap pemanggilan	Apakah setiap anak menerima pertanyaan yang benar?
4	Akar, jawaban kiri, jawaban kanan, dan urutan ketiganya	Bagaimana jawaban digabungkan, dan kapan simpul dikunjungi?
5	`root->left = f(root->left)` di mana pun tautan berubah	Apakah setiap induk memasang kembali apa yang dikembalikan anaknya?
6	Kasus-kasus tepi	Kosong, satu simpul, satu anak, nilai yang tidak ada, rantai panjang?
7	Pengujian, lalu pengujian yang sama di bawah pemeriksa memori	Apakah program bekerja, atau hanya *tampak* bekerja?

Bagaimana program-program bab ini diperiksa. Setiap program di bawah lengkap dan sudah dijalankan. Program C++ dikompilasi dengan GCC dalam mode C++17 dengan semua peringatan yang lazim dinyalakan, tanpa satu peringatan pun; masing-masing dikompilasi lagi dengan AddressSanitizer dan UndefinedBehaviorSanitizer [2], [3], yang mengawasi setiap akses memori selama program berjalan, dan dijalankan di bawah Valgrind [4]; setiap jalannya bersih, tanpa akses tidak sah dan tanpa satu *byte* pun yang bocor. Perintahnya sebagai berikut (di Windows, jalankan `p1201`, bukan `./p1201`):

```
g++ -std=c++17 -O2 -Wall -Wextra -o p1201 p1201_depth_first.cpp      (build biasa)
./p1201
g++ -std=c++17 -g -Wall -Wextra -fsanitize=address,undefined \
    -o p1201_asan p1201_depth_first.cpp
./p1201_asan         (keluaran sama, ditambah laporan jika memori disalahgunakan)
valgrind --leak-check=full ./p1201    (Linux: pemeriksaan sama, tanpa kompilasi)
```

Program Java dikompilasi dengan `javac *.java` di foldernya dan program Python dijalankan dengan `python3`; program-program itu ditujukan untuk JDK 25 dan Python 3.14 tetapi hanya memerlukan JDK 21 dan Python 3.10. Untuk setiap soal, ketiga program mencetak baris-baris yang persis sama, yang diperiksa *byte* demi *byte*. Folder kode bab ini beserta `README.md`-nya memuat setiap program dan setiap penyelesaian keliru yang diperlihatkan di bawah.

Satu pohon, tiga bahasa. **Tabel 12.3** adalah kamus kecil untuk membaca ketiga versi setiap program secara berdampingan. Tabel ini memperluas tabel sejenis di Bab 8 dengan baris-baris yang diperlukan pohon.

Tabel 12.3 Gagasan-gagasan pohon bab ini dalam tiga bahasa buku ini. Hanya C++ yang memungkinkan fungsi menerima alamat pointer milik pemanggil; Java dan Python mengembalikan akar yang baru, atau menyimpan variabel yang harus diubah di dalam sebuah objek.

Gagasan	C++	Java	Python
Simpul pohon	`struct Node { int data; Node *left; Node *right; };`	`class Node { int data; Node left, right; }`	`class Node:` dengan `data`, `left`, `right`
Pohon kosong	`nullptr`	`null`	`None`
Menyisipkan, mengubah akar milik pemanggil	`insert(&root, 6)` dengan `Node **root`	`root = insert(root, 6)`	`root = insert(root, 6)`
Membebaskan seluruh pohon	`clear(&root)`: anak-anak dulu, lalu simpulnya	`root = null` (*garbage collector*)	`root = None` (otomatis)
Fungsi bantu yang harus mengubah dua variabel pemanggil	kirim `Node **head, Node **prev`	letakkan keduanya dalam satu objek kecil	letakkan keduanya dalam satu objek kecil
Berbagi *toolkit*	`#include "tree_kit.hpp"`	`class P1201... extends TreeKit`	`from tree_kit import *`
Stack / queue pustaka	`std::stack`, `std::queue`	`ArrayDeque`	`list`, `collections.deque`

Dua baris layak mendapat satu kalimat masing-masing. Hanya C++ yang mewajibkan *Anda* membebaskan simpul, sehingga hanya program C++ yang dapat membocorkan subpohon atau membaca simpul yang sudah dibebaskan, dan hanya program C++ yang dijalankan di bawah pemeriksa memori; setiap program itu diakhiri dengan memanggil `clear`. Dan hanya C++ yang dapat mengirim alamat pointer milik pemanggil, sebuah `Node **` (Bab 6); Java dan Python *mengembalikan* akar yang baru, sebagaimana versi-versinya di Bab 8 [5], [6], [7].

12.3 Pohon yang Kita Kerjakan

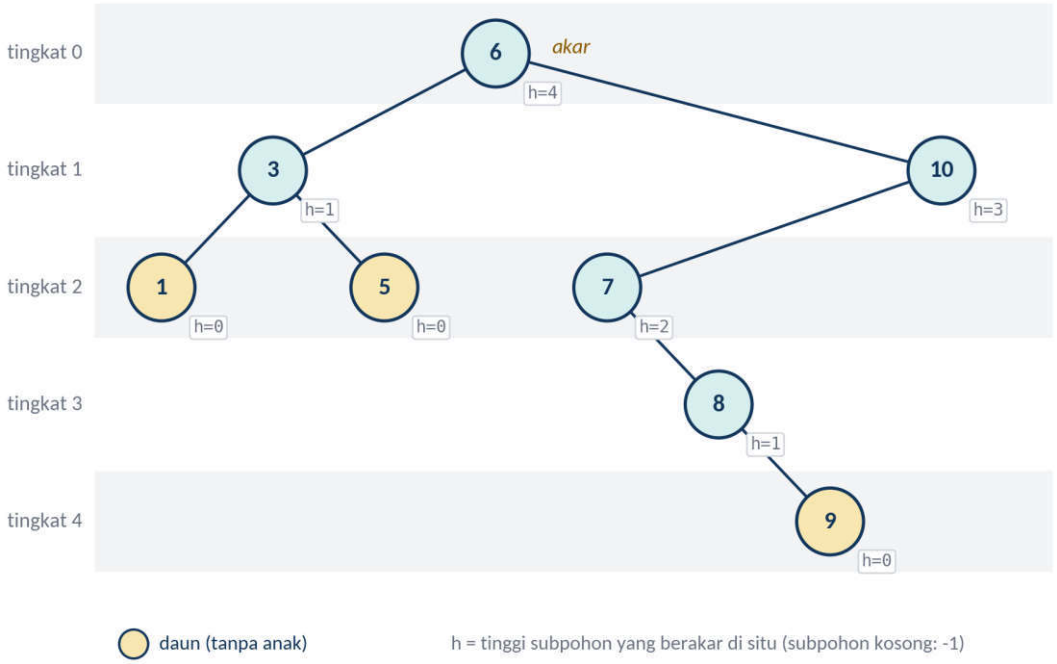
12.3.1 Titik awalnya

Sepuluh soal pertama berangkat dari satu titik awal yang sama, yang kita nyatakan lebih dulu dalam bentuk aslinya.

> **SOAL.** Starting from a binary search tree program that already has `insert`, `clear`, the traversals `preOrder`, `inOrder` and `postOrder`, `search`, `count` and `height`, write the functions of

> Problems 12.1 to 12.10. As a case study, use the tree built by this driver: `clear(&root); insert(&root, 6); insert(&root, 3); insert(&root, 10); insert(&root, 1); insert(&root, 5); insert(&root, 7); insert(&root, 8); insert(&root, 9);`
>
> *Dengan kata lain:* berangkat dari program *binary search tree* yang sudah memiliki `insert`, `clear`, penelusuran `preOrder`, `inOrder`, dan `postOrder`, `search`, `count`, serta `height`, tulislah fungsi-fungsi Soal 12.1 sampai 12.10, dengan pohon yang dibangun oleh *driver* di atas sebagai studi kasus.

Driver itu membangun pohon pada **Gambar 12.1**. Bangunlah sekali dengan tangan, karena setiap soal di bawah menelusurinya. Nilai pertama, 6, masuk ke pohon kosong dan menjadi akar. Lalu 3 lebih kecil daripada 6, jadi ia ke kiri 6. 10 lebih besar daripada 6, jadi ia ke kanan. 1 lebih kecil daripada 6 (ke kiri, ke 3) dan lebih kecil daripada 3 (ke kiri lagi), dan kiri 3 kosong, jadi 1 menetap di situ. 5 lebih kecil daripada 6 (ke 3) tetapi lebih besar daripada 3, jadi ia menjadi anak kanan 3. 7 lebih besar daripada 6 (ke 10) dan lebih kecil daripada 10, jadi ia menjadi anak kiri 10. 8 ke kanan di 6, ke kiri di 10, ke kanan di 7. Dan 9 ke kanan di 6, ke kiri di 10, ke kanan di 7, dan ke kanan di 8. Delapan nilai, delapan simpul.



Gambar 12.1 Pohon bab ini, dibangun dengan menyisipkan 6, 3, 10, 1, 5, 7, 8, 9 ke dalam *binary search tree* kosong. Tingkat dihitung dari 0 di akar, daun diberi arsiran, dan h adalah tinggi subpohon yang berakar di setiap simpul.

Bacalah tiga istilah dari gambar itu, karena soal-soal memakainya terus-menerus. **Tingkat** (atau *kedalaman*) sebuah simpul adalah banyaknya sisi dari akar turun ke simpul itu: 6 berada di tingkat 0, 3 dan 10 di tingkat 1, 1, 5, dan 7 di tingkat 2, 8 di tingkat 3, dan 9 di tingkat 4. **Tinggi** sebuah simpul adalah banyaknya sisi pada lintasan terpanjang dari simpul itu turun ke sebuah daun; tinggi pohon adalah tinggi akarnya, di sini 4. Menurut konvensi Subbab 11.2.3, pohon kosong bertinggi -1, yang membuat aturannya sama untuk setiap simpul:

$$h(\varnothing) = -1, \qquad h(T) = 1 + \max(h(T.left),\, h(T.right)). \tag{12.2}$$

Periksalah Persamaan (12.2) pada sebuah daun: kedua anaknya kosong, jadi tingginya $1 + \max(-1, -1) = 0$, sebagaimana mestinya. Pada simpul 8: kirinya kosong (-1) dan kanannya daun

9 (0), jadi tingginya $1 + 0 = 1$. **Daun** adalah simpul yang sama sekali tidak memiliki anak: 1, 5, dan 9.

Pohon ini adalah *binary search tree*, dan aturan yang menjadikannya demikian layak dituliskan dengan cermat, karena Soal 12.4, 12.5, dan 12.14 bertumpu padanya:

$$\text{untuk setiap simpul } x\text{:}\quad y < x.key \text{ untuk setiap } y \text{ di } x.left,\quad y > x.key \text{ untuk setiap } y \text{ di } x.right. \tag{12.3}$$

Perhatikan kata *setiap* pada Persamaan (12.3). Aturan itu tidak hanya menyangkut sebuah simpul dan kedua anaknya; aturan itu menyangkut sebuah simpul dan *semua* keturunannya. Simpul 5 berada di subpohon kiri 6, jadi ia harus lebih kecil daripada 6, walaupun ia adalah anak *kanan* dari induknya sendiri, 3. Akibat yang akan kita pakai berulang kali adalah bahwa penelusuran *inorder* (kiri, simpul, kanan) pada *binary search tree* mencetak nilai-nilainya dalam urutan menaik [1], [8].

Seberapa tinggi, dan seberapa rendah, pohon biner dengan n simpul? Subbab 11.5.3 sudah menjawabnya, dan kita ulangi argumennya. Tingkat k dapat memuat paling banyak 2^k simpul, karena setiap simpul di tingkat $k-1$ memiliki paling banyak dua anak. Jadi pohon bertinggi h memuat paling banyak $1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1$ simpul, dan jika dibalik,

$$n \le 2^{h+1} - 1 \quad \Rightarrow \quad h \ge \lceil \lg(n+1) \rceil - 1, \qquad \text{dan selalu} \quad h \le n - 1. \tag{12.4}$$

Di sini lg adalah logaritma berbasis 2 dan $\lceil x \rceil$ membulatkan x ke atas menjadi bilangan bulat. Untuk $n = 8$ kita, Persamaan (12.4) menyatakan bahwa tingginya paling sedikit $\lceil \lg 9 \rceil - 1 = 4 - 1 = 3$ dan paling banyak 7. Pohon kita bertinggi 4, dekat ujung yang rendah. Pohon yang dibangun dengan menyisipkan 1, 2, 3, ..., 8 dalam urutan menaik akan menjadi satu rantai yang condong ke kanan, bertinggi 7, ujung yang tinggi. Rantai itu adalah kasus terburuk bagi setiap fungsi dalam bab ini yang menempuh satu lintasan, dan kita akan memakainya sebagai kasus tepi.

12.3.2 *Toolkit*, dalam tiga bahasa

Alih-alih mengulang fungsi penyisipan, penelusuran, pencarian, penghitungan, dan tinggi di setiap program, bab ini menyimpannya dalam satu berkas kecil per bahasa, yaitu *toolkit*, dan setiap program pohon memakainya. **Kode 12.1** adalah *toolkit* C++. Fungsi-fungsinya adalah fungsi-fungsi Subbab 11.7, dengan tiga perubahan: `clear` benar-benar membebaskan simpul (versi yang hanya menyetel akar menjadi `nullptr` akan membocorkan setiap simpul), `search` mengembalikan `nullptr` tanpa mencetak apa pun, dan dua fungsi bantu membangun pohon dengan cepat: `insertAll` menyisipkan sederet nilai, dan `buildSample` membangun pohon pada **Gambar 12.1**. Fungsi terakhir, `showSideways`, mencetak pohon sebagai teks, diputar seperempat putaran ke kiri, sehingga program dapat memperlihatkan bentuk pohon yang telah diubahnya.

Kode 12.1 `tree_kit.hpp`: *toolkit* C++ yang dipakai bersama oleh program-program pohon bab ini.

```cpp
// tree_kit.hpp - the binary search tree toolkit shared by Problems 12.1-12.10.
// Every tree program of Chapter 12 starts with #include "tree_kit.hpp".
#ifndef TREE_KIT_HPP
#define TREE_KIT_HPP

#include <cstdio>
#include <initializer_list>

struct Node {
    int data;
    Node *left;     // the subtree of smaller values
    Node *right;    // the subtree of larger values
};

// Insert newData into the BST whose root pointer lives at *root.
// A value that is already in the tree is ignored.
inline void insert(Node **root, int newData) {
    if (*root == nullptr) {
        *root = new Node{newData, nullptr, nullptr};
    } else if (newData < (*root)->data) {
        insert(&(*root)->left, newData);
    } else if (newData > (*root)->data) {
        insert(&(*root)->right, newData);
    }
}

// Free every node (both subtrees first, then the node) and empty the tree.
inline void clear(Node **root) {
    if (*root == nullptr) return;
    clear(&(*root)->left);
    clear(&(*root)->right);
    delete *root;
    *root = nullptr;
}

inline void preOrder(Node *root) {      // node, left, right
    if (root == nullptr) return;
    printf("%d ", root->data);
    preOrder(root->left);
    preOrder(root->right);
}
```

Kode 12.1 (Lanjutan) *tree_kit.hpp: toolkit C++ yang dipakai bersama oleh program-program pohon bab ini.*

```
inline void inOrder(Node *root) {       // left, node, right
    if (root == nullptr) return;
    inOrder(root->left);
    printf("%d ", root->data);
    inOrder(root->right);
}

inline void postOrder(Node *root) {     // left, right, node
    if (root == nullptr) return;
    postOrder(root->left);
    postOrder(root->right);
    printf("%d ", root->data);
}

// The node that holds data, or nullptr if there is none.
inline Node *search(Node *root, int data) {
    if (root == nullptr) return nullptr;
    if (data < root->data) return search(root->left, data);
    if (data > root->data) return search(root->right, data);
    return root;
}

inline int count(Node *root) {
    if (root == nullptr) return 0;
    return count(root->left) + count(root->right) + 1;
}

// Height in edges: -1 for the empty tree, 0 for a single node.
inline int height(Node *root) {
    if (root == nullptr) return -1;
    int u = height(root->left);
    int v = height(root->right);
    return (u > v ? u : v) + 1;
}

// Insert several values, from left to right.
inline void insertAll(Node **root, std::initializer_list<int> values) {
    for (int v : values) insert(root, v);
}

// The tree used throughout the chapter: 6, 3, 10, 1, 5, 7, 8, 9 in this order.
inline void buildSample(Node **root) {
    insertAll(root, {6, 3, 10, 1, 5, 7, 8, 9});
}
```

Kode 12.1 (Lanjutan) *tree_kit.hpp: toolkit C++ yang dipakai bersama oleh program-program pohon bab ini.*

```
// Print the tree turned a quarter turn to the left: the root at the margin,
// its right subtree above it and its left subtree below it.
inline void showSideways(Node *root, int depth = 0) {
    if (root == nullptr) return;
    showSideways(root->right, depth + 1);
    printf("%*s%d\n", 4 * depth, "", root->data);
    showSideways(root->left, depth + 1);
}

#endif
```

Tiga rincian *toolkit* ini layak dibahas sebelum kita mengandalkannya. `insert` **menerima** `Node **root`, alamat pointer milik pemanggil, karena menyisipkan ke pohon kosong harus mengubah `root` milik pemanggil dari `nullptr` menjadi simpul baru; pemanggilan `insert(&(*root)->left, newData)` mengirim alamat medan `left` di dalam simpul saat ini, sehingga pemanggilan rekursif

dapat mengisi medan itu ketika sampai di dasar. Bab 6 menjelaskan pointer ganda ini, dan Soal 12.10 akan memerlukannya lagi. `clear` **adalah fungsi *postorder***: fungsi ini membebaskan kedua subpohon sebelum simpulnya sendiri, karena begitu sebuah simpul dibebaskan, medan `left` dan `right`-nya tidak boleh dibaca lagi. `showSideways` **adalah penelusuran "kanan, simpul, kiri" dari Subbab 11.7.6** yang mengindentasi setiap nilai sebanyak empat spasi per tingkat. Mencetak subpohon kanan lebih dulu menempatkannya di atas, sehingga jika halaman diputar seperempat putaran searah jarum jam, cetakannya tampak seperti **Gambar 12.1**.

Kode 12.2 dan **Kode 12.3** adalah *toolkit* yang sama dalam Java dan Python. Keduanya mengembalikan akar yang baru dari `insert`, sebagaimana dijelaskan **Tabel 12.3**. Dalam Java setiap program pohon bab ini dideklarasikan `extends TreeKit`, yang membuatnya dapat memakai `Node`, `insert`, dan lainnya dengan nama pendeknya; dalam Python setiap program diawali dengan `from tree_kit import *`.

Kode 12.2 `TreeKit.java`: *toolkit* Java.

```java
// TreeKit.java - the binary search tree toolkit shared by Problems 12.1-12.10.
// Every tree program of Chapter 12 is declared as "extends TreeKit" to use it.
public class TreeKit {

    static class Node {
        int data;
        Node left;      // the subtree of smaller values
        Node right;     // the subtree of larger values

        Node(int data, Node left, Node right) {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }

    // Insert newData and return the root (a new root if the tree was empty).
    // A value that is already in the tree is ignored.
    static Node insert(Node root, int newData) {
        if (root == null) {
            return new Node(newData, null, null);
        }
        if (newData < root.data) {
            root.left = insert(root.left, newData);
        } else if (newData > root.data) {
            root.right = insert(root.right, newData);
        }
        return root;
    }

    // Insert several values, from left to right.
    static Node insertAll(Node root, int... values) {
        for (int v : values) root = insert(root, v);
        return root;
    }

    // The tree used throughout the chapter: 6, 3, 10, 1, 5, 7, 8, 9 in this order.
    static Node buildSample() {
        return insertAll(null, 6, 3, 10, 1, 5, 7, 8, 9);
    }
```

Kode 12.2 (Lanjutan) *TreeKit.java: toolkit Java.*

```
    static void preOrder(Node root) {      // node, left, right
        if (root == null) return;
        System.out.print(root.data + " ");
        preOrder(root.left);
        preOrder(root.right);
    }

    static void inOrder(Node root) {       // left, node, right
        if (root == null) return;
        inOrder(root.left);
        System.out.print(root.data + " ");
        inOrder(root.right);
    }

    static void postOrder(Node root) {     // left, right, node
        if (root == null) return;
        postOrder(root.left);
        postOrder(root.right);
        System.out.print(root.data + " ");
    }

    // The node that holds data, or null if there is none.
    static Node search(Node root, int data) {
        if (root == null) return null;
        if (data < root.data) return search(root.left, data);
        if (data > root.data) return search(root.right, data);
        return root;
    }

    static int count(Node root) {
        if (root == null) return 0;
        return count(root.left) + count(root.right) + 1;
    }

    // Height in edges: -1 for the empty tree, 0 for a single node.
    static int height(Node root) {
        if (root == null) return -1;
        return Math.max(height(root.left), height(root.right)) + 1;
    }
```

Kode 12.2 (Lanjutan) *TreeKit.java: toolkit Java.*

```
    // Print the tree turned a quarter turn to the left: the root at the margin,
    // its right subtree above it and its left subtree below it.
    static void showSideways(Node root, int depth) {
        if (root == null) return;
        showSideways(root.right, depth + 1);
        System.out.println(" ".repeat(4 * depth) + root.data);
        showSideways(root.left, depth + 1);
    }

    static void showSideways(Node root) {
        showSideways(root, 0);
    }
}
```

Kode 12.3 `tree_kit.py`: *toolkit* Python.

```
# tree_kit.py - the binary search tree toolkit shared by Problems 12.1-12.10.
# Every tree program of Chapter 12 starts with "from tree_kit import *".


class Node:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left        # the subtree of smaller values
        self.right = right      # the subtree of larger values


def insert(root, new_data):
    """Insert new_data and return the root (a new root if the tree was empty).
    A value that is already in the tree is ignored."""
    if root is None:
        return Node(new_data)
    if new_data < root.data:
        root.left = insert(root.left, new_data)
    elif new_data > root.data:
        root.right = insert(root.right, new_data)
    return root


def insert_all(root, values):
    """Insert several values, from left to right."""
    for v in values:
        root = insert(root, v)
    return root


def build_sample():
    """The tree used throughout the chapter: 6, 3, 10, 1, 5, 7, 8, 9, in order."""
    return insert_all(None, [6, 3, 10, 1, 5, 7, 8, 9])


def pre_order(root):            # node, left, right
    if root is None:
        return
    print(root.data, end=" ")
    pre_order(root.left)
    pre_order(root.right)
```

Kode 12.3 (Lanjutan) *tree_kit.py: toolkit Python.*

```
def in_order(root):             # left, node, right
    if root is None:
        return
    in_order(root.left)
    print(root.data, end=" ")
    in_order(root.right)


def post_order(root):           # left, right, node
    if root is None:
        return
    post_order(root.left)
    post_order(root.right)
    print(root.data, end=" ")


def search(root, data):
    """The node that holds data, or None if there is none."""
    if root is None:
        return None
    if data < root.data:
        return search(root.left, data)
    if data > root.data:
        return search(root.right, data)
    return root


def count(root):
    if root is None:
        return 0
    return count(root.left) + count(root.right) + 1


def height(root):
    """Height in edges: -1 for the empty tree, 0 for a single node."""
    if root is None:
        return -1
    return max(height(root.left), height(root.right)) + 1
```

Kode 12.3 (Lanjutan) *tree_kit.py: toolkit Python.*

```
def show_sideways(root, depth=0):
    """Print the tree turned a quarter turn to the left: the root at the margin,
    its right subtree above it and its left subtree below it."""
    if root is None:
        return
    show_sideways(root.right, depth + 1)
    print("    " * (4 * depth) + str(root.data))
    show_sideways(root.left, depth + 1)
```

12.3.3 Mengamati pohonnya

Kode 12.4 membangun pohon itu dan mengamatinya dengan segala cara yang dimungkinkan *toolkit*. Keluarannya menjadi acuan kita untuk seluruh bab.

Kode 12.4 `p1200_sample_tree.cpp`: pohon bab ini, dibangun dan diamati.

```cpp
// p1200_sample_tree.cpp - the tree of Chapter 12, built and inspected.
#include "tree_kit.hpp"

int main() {
    Node *root = nullptr;
    buildSample(&root);                      // 6, 3, 10, 1, 5, 7, 8, 9
    printf("The tree, turned on its side (root at the left):\n");
    showSideways(root);
    printf("preorder : ");
    preOrder(root);
    printf("\ninorder  : ");
    inOrder(root);
    printf("\npostorder: ");
    postOrder(root);
    printf("\nnodes %d, height %d\n", count(root), height(root));
    printf("search 8 : %s\n", search(root, 8) != nullptr ? "found" : "not found");
    printf("search 18: %s\n", search(root, 18) != nullptr ? "found" : "not found");
    clear(&root);
    printf("after clear: nodes %d, height %d\n", count(root), height(root));
    return 0;
}
```

```
The tree, turned on its side (root at the left):
    10
                9
            8
        7
6
        5
    3
        1
preorder : 6 3 1 5 10 7 8 9
inorder  : 1 3 5 6 7 8 9 10
postorder: 1 5 3 9 8 7 10 6
nodes 8, height 4
search 8 : found
search 18: not found
after clear: nodes 0, height -1
```

Putarlah halaman seperempat putaran searah jarum jam, dan delapan baris pertama menjadi **Gambar 12.1**: 6 di akar, 10 di atasnya di sebelah kanan, 7, 8, dan 9 tergantung di bawah 10, dan 3 dengan anak-anaknya 1 dan 5 di sebelah kiri. Baris *preorder* mengunjungi setiap simpul sebelum subpohonnya; baris *inorder* terurut, sebagaimana dijanjikan Persamaan (12.3); baris *postorder* mengunjungi setiap simpul sesudah subpohonnya, sehingga akar 6 muncul terakhir. Banyaknya simpul 8 dan tingginya 4, keduanya sesuai perhitungan tangan. Sesudah `clear`, kedua fungsi melihat pohon kosong dan memberikan 0 dan -1, dan Valgrind memastikan bahwa `clear` membebaskan kedelapan simpul. **Kode 12.5** dan **Kode 12.6** mencetak baris-baris yang persis sama.

Kode 12.5 `P1200SampleTree.java`: pohon diamati dalam Java.

```java
// P1200SampleTree.java - the tree of Chapter 12, built and inspected.
public class P1200SampleTree extends TreeKit {

    static String found(Node root, int x) {
        return search(root, x) != null ? "found" : "not found";
    }

    public static void main(String[] args) {
        Node root = buildSample();               // 6, 3, 10, 1, 5, 7, 8, 9
        System.out.println("The tree, turned on its side (root at the left):");
        showSideways(root);
        System.out.print("preorder : ");
        preOrder(root);
        System.out.print("\ninorder  : ");
        inOrder(root);
        System.out.print("\npostorder: ");
        postOrder(root);
        System.out.printf("%nnodes %d, height %d%n", count(root), height(root));
        System.out.println("search 8 : " + found(root, 8));
        System.out.println("search 18: " + found(root, 18));
        root = null;  // the garbage collector frees the nodes
        System.out.printf("after clear: nodes %d, height %d%n",
                count(root), height(root));
    }
}
```

Kode 12.6 `p1200_sample_tree.py`: pohon diamati dalam Python.

```python
# p1200_sample_tree.py - the tree of Chapter 12, built and inspected.
from tree_kit import *


def main():
    root = build_sample()                    # 6, 3, 10, 1, 5, 7, 8, 9
    print("The tree, turned on its side (root at the left):")
    show_sideways(root)
    print("preorder : ", end="")
    pre_order(root)
    print("\ninorder  : ", end="")
    in_order(root)
    print("\npostorder: ", end="")
    post_order(root)
    print(f"\nnodes {count(root)}, height {height(root)}")
    print("search 8 :", "found" if search(root, 8) is not None else "not found")
    print("search 18:", "found" if search(root, 18) is not None else "not found")
    root = None                              # Python frees the nodes by itself
    print(f"after clear: nodes {count(root)}, height {height(root)}")


if __name__ == "__main__":
    main()
```

12.4 Soal 12.1 — Penelusuran *Depth First*, Cabang Kiri Lebih Dulu

12.4.1 Soalnya

> **SOAL.** Write a recursive function to print the depth-first traversal of a tree, i.e., choosing left branches first. `void printDepthFirst(Node *root)`
>
> *Dengan kata lain:* tulislah fungsi rekursif yang mencetak penelusuran *depth first* (mendalam lebih dulu) sebuah pohon, dengan selalu memilih cabang kiri lebih dulu.

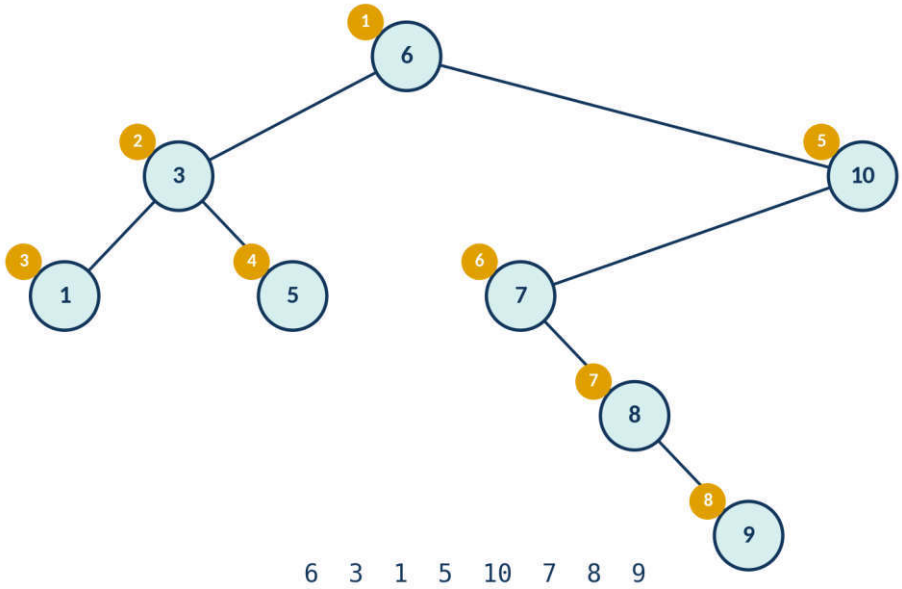
12.4.2 Dengan kata-kata kita sendiri

Depth first berarti: begitu Anda turun ke sebuah cabang, selesaikan seluruh cabang itu sebelum melihat ke tempat lain. *Cabang kiri lebih dulu* berarti: di setiap simpul, cabang kirilah yang pertama dimasuki. Dan *mencetak* penelusuran berarti mencetak setiap simpul pada saat Anda pertama kali tiba di sana. Jika digabungkan: cetak simpulnya, lalu cetak semua yang ada di subpohon kirinya, lalu semua yang ada di subpohon kanannya. Itulah tepat *preorder* dari Subbab 11.6.1, dan untuk pohon kita keluaran yang diharapkan adalah `6 3 1 5 10 7 8 9`.

Base case-nya (Langkah 2 metode kita) adalah pohon kosong, dan mencetak pohon kosong tidak mencetak apa pun. Kasus rekursifnya (Langkah 4) adalah "simpul, kiri, kanan", dengan pekerjaan simpul itu sendiri, yaitu `printf`, dikerjakan *sebelum* kedua pemanggilan.

12.4.3 Dengan tangan, selangkah demi selangkah

Gambar 12.2 memberi nomor pada simpul-simpul pohon menurut urutan pencetakannya. Ikutilah rekursinya, dengan mengingat bahwa setiap pemanggilan menuntaskan subpohon kirinya sebelum memulai subpohon kanannya.



Gambar 12.2 Urutan *depth first* pada Soal 12.1: setiap lencana menunjukkan saat simpul itu dicetak. Seluruh subpohon kiri 6 (lencana 2 sampai 4) dituntaskan sebelum subpohon kanan (lencana 5 sampai 8) dimasuki.

1. Pemanggilan pada 6 mencetak **6**, lalu memanggil dirinya pada anak kiri 3.
2. Pemanggilan pada 3 mencetak **3**, lalu memanggil dirinya pada 1.
3. Pemanggilan pada 1 mencetak **1**. Anak kirinya `nullptr`, sehingga pemanggilan itu langsung kembali; demikian pula pemanggilan pada anak kanannya. Pemanggilan pada 1 selesai dan kembali ke pemanggilan pada 3.
4. Pemanggilan pada 3 kini memanggil dirinya pada anak kanan 5, yang mencetak **5** dan, karena kedua anaknya kosong, kembali. Pemanggilan pada 3 selesai; seluruh subpohon kiri 6 sudah tercetak.
5. Pemanggilan pada 6 memanggil dirinya pada 10, yang mencetak **10** dan turun ke kiri ke 7, yang mencetak **7**. Kiri 7 kosong; kanannya 8, yang mencetak **8**, dan kanan 8 adalah 9, yang mencetak **9**.

Barisan yang tercetak adalah `6 3 1 5 10 7 8 9`. Seberapa dalam *call stack*-nya? Pada saat 9 dicetak, pemanggilan pada 6, 10, 7, 8, dan 9 semuanya sedang menunggu di stack, lima buah: satu lebih banyak daripada tingginya. Secara umum sebuah penelusuran memerlukan ruang stack sebanding dengan $h + 1$, yang kecil untuk pohon rendah dan sebesar n untuk sebuah rantai.

12.4.4 Penyelesaian C++

Kode 12.7 memberikan fungsi rekursifnya dan, di sebelahnya, versi kedua tanpa rekursi, untuk memperlihatkan ke mana pekerjaan tersembunyi rekursi itu pergi. Setiap pemanggilan rekursif, di baliknya, adalah sebuah *push* ke *call stack*; versi iteratif mem-*push* simpul-simpul yang masih harus dikunjungi ke `std::stack` miliknya sendiri (Bab 5), teknik Subbab 10.13 dan 11.6.4. Versi itu mem-*push* anak kanan *sebelum* anak kiri, sehingga anak kiri keluar dari stack lebih dulu, sesuai tuntutan "cabang kiri lebih dulu".

Kode 12.7 `p1201_depth_first.cpp`: Soal 12.1, secara rekursif dan dengan stack eksplisit.

```
// p1201_depth_first.cpp - Problem 12.1: depth-first traversal, left branch first.
#include <stack>
#include "tree_kit.hpp"

// Visit the node, then everything on its left, then everything on its right.
void printDepthFirst(Node *root) {
    if (root == nullptr) return;         // base case: an empty tree prints nothing
    printf("%d ", root->data);           // 1. the node itself
    printDepthFirst(root->left);         // 2. the whole left subtree
    printDepthFirst(root->right);        // 3. the whole right subtree
}

// The same order without recursion: a stack holds the nodes still to visit.
void printDepthFirstStack(Node *root) {
    std::stack<Node *> todo;
    if (root != nullptr) todo.push(root);
    while (!todo.empty()) {
        Node *cur = todo.top();
        todo.pop();
        printf("%d ", cur->data);
        if (cur->right != nullptr) todo.push(cur->right);   // pushed first,
        if (cur->left != nullptr) todo.push(cur->left);     // so visited second
    }
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    printf("depth-first, recursive : ");
    printDepthFirst(root);
    printf("\ndepth-first, with stack: ");
    printDepthFirstStack(root);
    Node *empty = nullptr;
    printf("\nempty tree             : [");
    printDepthFirst(empty);
    printf("]\n");
    clear(&root);
    return 0;
}
```

```
depth-first, recursive : 6 3 1 5 10 7 8 9
depth-first, with stack: 6 3 1 5 10 7 8 9
empty tree             : []
```

Baris demi baris. Kedua versi mencetak `6 3 1 5 10 7 8 9` yang diharapkan. Baris ketiga memanggil fungsi itu pada pohon kosong; tanda kurung siku di sekeliling keluarannya

memperlihatkan bahwa fungsi itu tidak mencetak apa pun dan, yang lebih penting, tidak *crash*. Versi rekursifnya memiliki tiga baris kerja nyata: *base case*, pencetakan, dan dua pemanggilan. **Biaya.** Menurut Persamaan (12.1), waktunya $\Theta(n)$: satu pencetakan per simpul dan sejumlah tetap kerja per anak kosong. Memori tambahannya adalah stack: $O(h)$ *frame* untuk versi rekursif, dan paling banyak $O(h)$ simpul yang menunggu di stack eksplisit pada versi iteratif (setiap tingkat menyumbang paling banyak satu anak kanan yang menunggu).

12.4.5 Penyelesaian Java dan Python

Kode 12.8 dan **Kode 12.9** mencetak ketiga baris yang persis sama. Java memakai `ArrayDeque` sebagai stack, yang `push` dan `pop`-nya bekerja di depan [9]; Python memakai list biasa, yang `append` dan `pop`-nya bekerja di ujung belakang [10].

Kode 12.8 `P1201DepthFirst.java`: Soal 12.1 dalam Java.

```java
// P1201DepthFirst.java - Problem 12.1: depth-first traversal, left branch first.
import java.util.ArrayDeque;

public class P1201DepthFirst extends TreeKit {

    // Visit the node, then everything on its left, then everything on its right.
    static void printDepthFirst(Node root) {
        if (root == null) return;  // base case: an empty tree prints nothing
        System.out.print(root.data + " ");    // 1. the node itself
        printDepthFirst(root.left);           // 2. the whole left subtree
        printDepthFirst(root.right);          // 3. the whole right subtree
    }

    // The same order without recursion: a stack holds the nodes still to visit.
    static void printDepthFirstStack(Node root) {
        ArrayDeque<Node> todo = new ArrayDeque<>();
        if (root != null) todo.push(root);
        while (!todo.isEmpty()) {
            Node cur = todo.pop();
            System.out.print(cur.data + " ");
            if (cur.right != null) todo.push(cur.right);   // pushed first,
            if (cur.left != null) todo.push(cur.left);     // so visited second
        }
    }

    public static void main(String[] args) {
        Node root = buildSample();
        System.out.print("depth-first, recursive : ");
        printDepthFirst(root);
        System.out.print("\ndepth-first, with stack: ");
        printDepthFirstStack(root);
        System.out.print("\nempty tree             : [");
        printDepthFirst(null);
        System.out.println("]");
    }
}
```

Kode 12.9 `p1201_depth_first.py`: Soal 12.1 dalam Python.

```python
# p1201_depth_first.py - Problem 12.1: depth-first traversal, left branch first.
from tree_kit import *


def print_depth_first(root):
    """Visit the node, then everything on its left, then on its right."""
    if root is None:                    # base case: an empty tree prints nothing
        return
    print(root.data, end=" ")           # 1. the node itself
    print_depth_first(root.left)        # 2. the whole left subtree
    print_depth_first(root.right)       # 3. the whole right subtree


def print_depth_first_stack(root):
    """The same order without recursion: a stack holds the nodes still to visit."""
    todo = [root] if root is not None else []
    while todo:
        cur = todo.pop()
        print(cur.data, end=" ")
        if cur.right is not None:
            todo.append(cur.right)      # pushed first,
        if cur.left is not None:
            todo.append(cur.left)       # so visited second


def main():
    root = build_sample()
    print("depth-first, recursive : ", end="")
    print_depth_first(root)
    print("\ndepth-first, with stack: ", end="")
    print_depth_first_stack(root)
    print("\nempty tree             : [", end="")
    print_depth_first(None)
    print("]")


if __name__ == "__main__":
    main()
```

12.4.6 Penyelesaian keliru yang menggoda

Jalan pintas yang menggoda adalah membuang *base case*, dengan anggapan bahwa "akarnya toh tidak pernah kosong":

```
void printDepthFirst(Node *root) {
    printf("%d ", root->data);          // WRONG: no base case at all
    printDepthFirst(root->left);
    printDepthFirst(root->right);
}
```

Compiler sudah mencium masalah dan memperingatkan `infinite recursion detected [-Winfinite-recursion]`: fungsi yang memanggil dirinya tanpa syarat tidak akan pernah berhenti dengan sendirinya. Yang sebenarnya menghentikannya lebih buruk lagi. Menjalankan program `wrong/p1201_wrong.cpp` di bawah *sanitizer* mencetak hal berikut (alamat heksadesimal yang panjang berbeda dari satu jalannya ke jalannya yang lain):

```
depth-first, recursive : 6 3 1 p1201_wrong.cpp:7:11: runtime error: member access within
null pointer of type 'struct Node'
==27788==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x55b82ce5f5d4
bp 0x7ffe08c512b0 sp 0x7ffe08c512a0 T0)
==27788==The signal is caused by a READ memory access.
==27788==Hint: address points to the zero page.
    #0 0x55b82ce5f5d4 in printDepthFirst(Node*) p1201_wrong.cpp:7
    #1 0x55b82ce5f66c in printDepthFirst(Node*) p1201_wrong.cpp:8
SUMMARY: AddressSanitizer: SEGV p1201_wrong.cpp:7 in printDepthFirst(Node*)
```

Penelusuran mencetak 6, 3, dan 1, lalu pemanggilan pada 1 memanggil dirinya pada anak kiri 1, yang bernilai `nullptr`, dan baris 7 membaca `root->data` melaluinya: program membaca alamat 0, yang tidak boleh dibaca oleh program mana pun, dan mati. Bacalah laporan itu dari atas: *apa* yang terjadi (pembacaan melalui pointer *null*), *di mana* (baris 7 dari `printDepthFirst`), dan *bagaimana kita sampai ke sana* (baris 8, pemanggilan pada anak kiri). Subbab 10.3.4 memperlihatkan kegagalan yang sama untuk fungsi yang *base case*-nya hilang. Pelajarannya adalah Langkah 2 metode kita: setiap fungsi rekursif pada pohon harus menyatakan apa yang terjadi pada `nullptr`, karena setiap pohon, sebesar apa pun, berakhir pada anak-anak `nullptr`, sebanyak $n + 1$ buah.

12.4.7 Uji diri

> **UJI DIRI.** Ubahlah satu baris `printDepthFirst` sehingga fungsi itu mencetak cabang *kanan* lebih dulu, dan berikan keluarannya pada pohon kita. Lalu sebutkan apa yang harus diubah pada versi iteratif.
>
> *Jawaban.* Tukarlah kedua pemanggilan rekursif: `printDepthFirst(root->right);` `printDepthFirst(root->left);`. Keluarannya menjadi `6 10 7 8 9 3 5 1`. Pada versi iteratif, tukarlah kedua baris `push`, sehingga anak kiri di-*push* lebih dulu dan anak kanan keluar dari stack lebih dulu. (Keluaran ini juga merupakan *preorder* dari bayangan cermin pohon itu, yang akan dihasilkan Soal 12.5.)

12.5 Soal 12.2 — Mencetak Satu Tingkat

12.5.1 Soalnya

> **SOAL.** Write a recursive function to print all nodes at the k-th level in the binary tree. `void printKthLevel(Node *root, int k)`
>
> *Dengan kata lain:* tulislah fungsi rekursif yang mencetak semua simpul pada tingkat ke-k pohon biner.

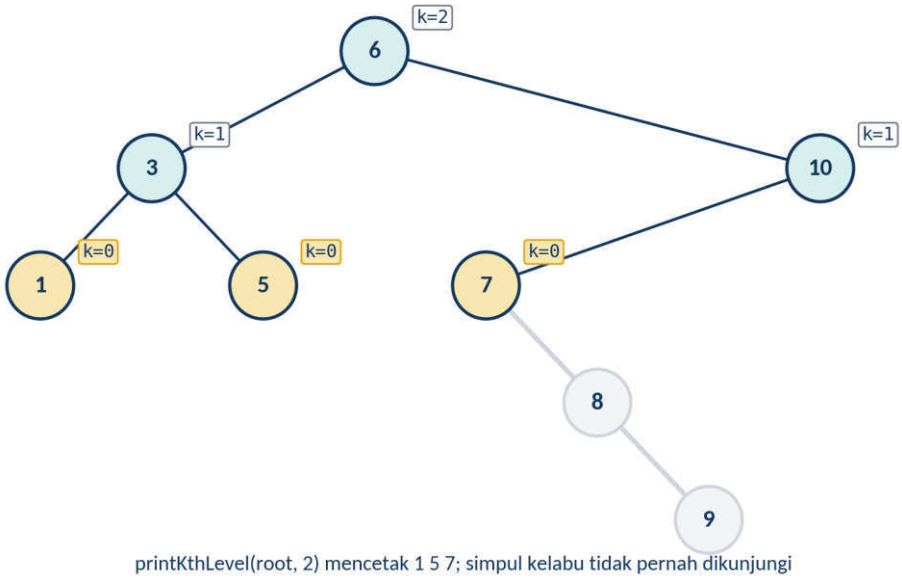
12.5.2 Dengan kata-kata kita sendiri

Akar berada di tingkat 0, anak-anaknya di tingkat 1, anak-anak mereka di tingkat 2, dan seterusnya (Subbab 12.3.1). Kita harus mencetak, dari kiri ke kanan, setiap simpul yang tingkatnya tepat k. Melihat **Gambar 12.1**: tingkat 0 adalah `6`, tingkat 1 adalah `3 10`, tingkat 2 adalah `1 5 7`, tingkat 3 adalah `8`, tingkat 4 adalah `9`, dan tingkat 5 serta yang lebih dalam kosong.

Kiat yang membuat soal ini rekursif adalah mengubah pertanyaannya dalam perjalanan turun. "Cetaklah simpul-simpul yang k tingkat di bawah*ku*" berarti, bagi setiap anakku, "cetaklah simpul-simpul yang $k - 1$ tingkat di bawah*mu*". Ketika bilangan itu mencapai 0, simpul yang menerimanya berada tepat di tingkat yang diinginkan dan mencetak dirinya sendiri. Ketika sebuah pemanggilan jatuh keluar dari pohon (menerima `nullptr`) sebelum bilangan itu mencapai 0, lintasan itu terlalu pendek, dan ia tidak mencetak apa pun.

12.5.3 Dengan tangan, selangkah demi selangkah

Gambar 12.3 menuliskan, di samping setiap simpul, nilai k yang diterima pemanggilannya ketika kita meminta tingkat 2.



Gambar 12.3 Pemanggilan `printKthLevel(root, 2)`: label setiap simpul adalah k yang diterima pemanggilannya. Simpul-simpul yang menerima $k = 0$ mencetak dirinya lalu berhenti, sehingga 8 dan 9, di bawahnya, tidak pernah dikunjungi.

1. Pemanggilan pada 6 menerima $k = 2$. Nilai itu bukan 0, jadi ia memanggil dirinya pada 3 dan pada 10 dengan $k = 1$.
2. Pemanggilan pada 3 menerima $k = 1$ dan memanggil dirinya pada 1 dan pada 5 dengan $k = 0$. Keduanya mencetak dirinya: **1**, **5**.
3. Pemanggilan pada 10 menerima $k = 1$ dan memanggil dirinya pada 7 dengan $k = 0$, yang mencetak **7**, dan pada anak kanannya, `nullptr`, yang langsung kembali.
4. Pemanggilan dengan $k = 0$ tidak turun lebih jauh, sehingga 8 dan 9 tidak pernah disentuh.

Keluarannya `1 5 7`, dalam urutan dari kiri ke kanan, karena setiap pemanggilan menangani anak kirinya sebelum anak kanannya.

12.5.4 Penyelesaian C++

Kode 12.10 `p1202_kth_level.cpp`: Soal 12.2, mencetak setiap tingkat dari 0 sampai satu tingkat melewati tingginya.

```cpp
// p1202_kth_level.cpp - Problem 12.2: print every node on level k (root: level 0).
#include "tree_kit.hpp"

void printKthLevel(Node *root, int k) {
    if (root == nullptr) return;         // fell off the tree: nothing on this path
    if (k == 0) {                        // this node is exactly k levels down
        printf("%d ", root->data);
        return;
    }
    printKthLevel(root->left, k - 1);    // one level lower, one step fewer to go
    printKthLevel(root->right, k - 1);
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    for (int k = 0; k <= height(root) + 1; k++) {
        printf("level %d: ", k);
        printKthLevel(root, k);
        printf("\n");
    }
    clear(&root);
    return 0;
}
```

```
level 0: 6
level 1: 3 10
level 2: 1 5 7
level 3: 8
level 4: 9
level 5:
```

Baris demi baris. Perulangan di `main` meminta setiap tingkat dari 0 sampai `height(root) + 1`, satu lebih banyak daripada yang ada, untuk menguji kasus tingkat di bawah daun terdalam: tercetak baris kosong, sebagaimana mestinya. Setiap baris lainnya sesuai dengan perhitungan tangan. Perhatikan bahwa fungsi ini menguji `root == nullptr` *sebelum* menguji `k == 0`: urutan sebaliknya akan membaca `root->data` dari pointer *null* setiap kali sebuah lintasan lebih pendek daripada k.

Biaya. Pemanggilan dengan suatu k hanya mengunjungi simpul-simpul pada tingkat 0 sampai k, sehingga biayanya $O(\text{banyaknya simpul pada tingkat } 0..k)$, paling banyak $\Theta(n)$. Mencetak *semua* tingkat dengan cara ini, seperti dilakukan `main`, berbiaya jumlah atas semua k, yang untuk rantai berisi n simpul adalah $1 + 2 + \cdots + n = \Theta(n^2)$. Soal 12.16 mencetak semua tingkat dalam $\Theta(n)$ dengan sebuah queue.

12.5.5 Penyelesaian Java dan Python

Kode 12.11 dan **Kode 12.12** adalah versi Java dan Python-nya; keduanya mencetak baris-baris yang persis sama.

Kode 12.11 `P1202KthLevel.java`: Soal 12.2 dalam Java.

```java
// P1202KthLevel.java - Problem 12.2: print every node on level k (root: level 0).
public class P1202KthLevel extends TreeKit {

    static void printKthLevel(Node root, int k) {
        if (root == null) return;  // fell off the tree: nothing on this path
        if (k == 0) {                          // this node is exactly k levels down
            System.out.print(root.data + " ");
            return;
        }
        printKthLevel(root.left, k - 1);  // one level lower, one step fewer to go
        printKthLevel(root.right, k - 1);
    }

    public static void main(String[] args) {
        Node root = buildSample();
        for (int k = 0; k <= height(root) + 1; k++) {
            System.out.print("level " + k + ": ");
            printKthLevel(root, k);
            System.out.println();
        }
    }
}
```

Kode 12.12 `p1202_kth_level.py`: Soal 12.2 dalam Python.

```python
# p1202_kth_level.py - Problem 12.2: print every node on level k (root: level 0).
from tree_kit import *


def print_kth_level(root, k):
    if root is None:                    # fell off the tree: nothing on this path
        return
    if k == 0:                          # this node is exactly k levels down
        print(root.data, end=" ")
        return
    print_kth_level(root.left, k - 1)    # one level lower, one step fewer to go
    print_kth_level(root.right, k - 1)


def main():
    root = build_sample()
    for k in range(height(root) + 2):
        print(f"level {k}: ", end="")
        print_kth_level(root, k)
        print()


if __name__ == "__main__":
    main()
```

Keduanya mencetak keenam baris yang sama dengan **Kode 12.10**. Dalam Python, `range(height(root) + 2)` memberikan $0, 1, \ldots, h + 1$, tingkat-tingkat yang sama dengan perulangan C++.

12.5.6 Penyelesaian keliru yang menggoda

"k harus berkurang satu di setiap tingkat", pikir si pemrogram, lalu menuliskan pengurangan itu ke dalam pemanggilan:

```
    printKthLevel(root->left, --k);      // WRONG: --k changes k itself,
    printKthLevel(root->right, --k);     // so the right side gets k - 2
```

Kode ini terkompilasi tanpa peringatan, dan `wrong/p1202_wrong.cpp` mencetak:

```
level 0: 6
level 1: 3
level 2: 1 10
level 3: 5 7
level 4:
level 5: 8
```

`--k` bukanlah "k dikurangi satu"; artinya "*jadikan* k lebih kecil satu, lalu pakai". Pemanggilan pertama menerima $k-1$, sesuai maksudnya, tetapi ia juga telah mengubah variabel lokal k, sehingga pemanggilan kedua menerima $k-2$. Karena itu setiap anak kanan ditanyai tingkat yang satu lebih dangkal, dan sisi kanan pohon muncul satu baris terlalu awal: 10, yang berada di tingkat 1, tercetak sebagai bagian dari tingkat 2. Ekspresi yang benar, `k - 1`, menghitung nilai baru dan membiarkan `k` tak tersentuh, sehingga kedua anak menerima pertanyaan yang sama. Inilah Langkah 3 metode kita: tuliskan argumen di samping setiap simpul, maka kedua anak 6 tampak jelas menerima bilangan yang berbeda.

12.5.7 Uji diri

> **UJI DIRI.** Tulislah `int countKthLevel(Node *root, int k)` yang *mengembalikan* banyaknya simpul pada tingkat k alih-alih mencetaknya. Apa yang dikembalikannya untuk $k = 0,1,2,3,4,5$ pada pohon kita, dan berapa jumlah keenam bilangan itu?
>
> *Jawaban.* `if (root == nullptr) return 0; if (k == 0) return 1; return countKthLevel(root->left, k - 1) + countKthLevel(root->right, k - 1);`. Fungsi itu mengembalikan 1, 2, 3, 1, 1, 0, yang jumlahnya 8, yaitu banyaknya simpul: setiap simpul berada di tepat satu tingkat.

12.6 Soal 12.3 — Jumlah Semua Simpul

12.6.1 Soalnya

> **SOAL.** Write a recursive function to calculate the sum of all nodes in the binary tree. `int sumOfNodes(Node *root)`
>
> *Dengan kata lain:* tulislah fungsi rekursif yang menghitung jumlah nilai semua simpul pohon biner.

12.6.2 Dengan kata-kata kita sendiri

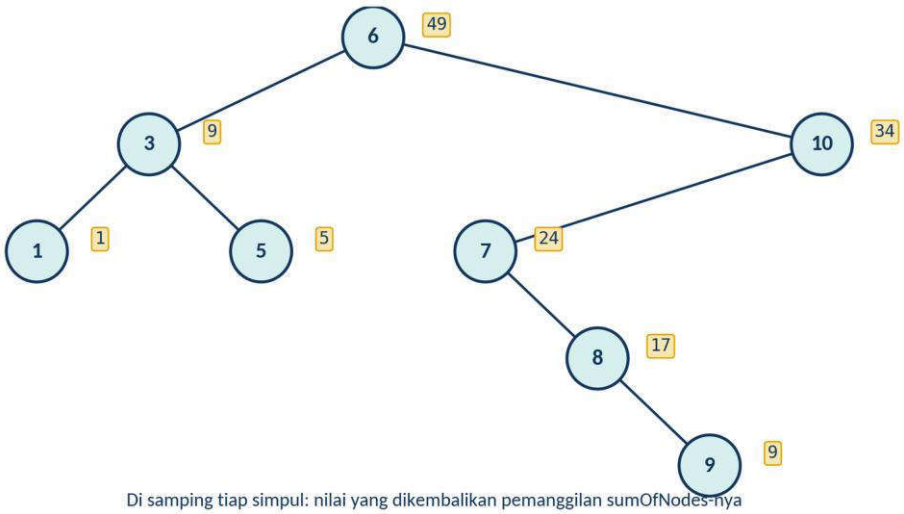
Jumlahkan nilai semua simpul. Dengan tangan, $6+3+10+1+5+7+8+9=49$. Definisi rekursifnya mengikuti kotak wawasan pada Subbab 12.1 kata demi kata:

$$S(\varnothing)=0, \qquad S(T)=T.key+S(T.left)+S(T.right). \tag{12.5}$$

Base case-nya 0 karena 0 adalah bilangan yang tidak mengubah apa pun bila ditambahkan: jumlah sebuah daun harus sama dengan nilai daun itu sendiri, dan menurut Persamaan (12.5) memang $key+0+0$.

12.6.3 Dengan tangan, selangkah demi selangkah

Gambar 12.4 menuliskan, di samping setiap simpul, nilai yang dikembalikan pemanggilannya. Bacalah dari bawah ke atas, karena itulah urutan pemanggilan-pemanggilan itu *selesai*: sebuah pemanggilan hanya dapat kembali setelah pemanggilan kedua anaknya kembali.



Gambar 12.4 Nilai yang dikembalikan `sumOfNodes` pada setiap subpohon. Masing-masing adalah nilai simpul itu sendiri ditambah dua nilai yang dikembalikan di bawahnya; akar mengembalikan 49.

Daun-daun mengembalikan nilainya sendiri: 1, 5, dan 9. Simpul 8 mengembalikan $8 + 0 + 9 = 17$, simpul 7 mengembalikan $7 + 0 + 17 = 24$, simpul 10 mengembalikan $10 + 24 + 0 = 34$, simpul 3 mengembalikan $3 + 1 + 5 = 9$, dan akar mengembalikan $6 + 9 + 34 = 49$. Sebagai jejak pemanggilan, diindentasi menurut kedalaman:

```
sumOfNodes(6)
    sumOfNodes(3)
        sumOfNodes(1) = 1 + 0 + 0 = 1
        sumOfNodes(5) = 5 + 0 + 0 = 5
    = 3 + 1 + 5 = 9
    sumOfNodes(10)
        sumOfNodes(7)
            sumOfNodes(8)
                sumOfNodes(9) = 9 + 0 + 0 = 9
            = 8 + 0 + 9 = 17
        = 7 + 0 + 17 = 24
    = 10 + 24 + 0 = 34
= 6 + 9 + 34 = 49
```

12.6.4 Penyelesaian C++

Kode 12.13 adalah program C++ untuk Soal 12.3.

Kode 12.13 `p1203_sum.cpp`: Soal 12.3, satu baris rekursi.

```cpp
// p1203_sum.cpp - Problem 12.3: the sum of all the values in a binary tree.
#include "tree_kit.hpp"

int sumOfNodes(Node *root) {
    if (root == nullptr) return 0;       // the empty tree adds nothing
    return root->data + sumOfNodes(root->left) + sumOfNodes(root->right);
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    printf("sum of the whole tree  : %d\n", sumOfNodes(root));
    printf("sum again (same answer): %d\n", sumOfNodes(root));
    printf("left subtree of 6      : %d\n", sumOfNodes(root->left));
    printf("right subtree of 6     : %d\n", sumOfNodes(root->right));
    printf("empty tree             : %d\n", sumOfNodes(nullptr));
    clear(&root);
    return 0;
}
```

```
sum of the whole tree  : 49
sum again (same answer): 49
left subtree of 6      : 9
right subtree of 6     : 34
empty tree             : 0
```

Baris demi baris. Seluruh pohon berjumlah 49. Baris kedua memanggil fungsi itu lagi dan memperoleh 49 lagi; pengujian itu tampak terlalu jelas untuk dilakukan, dan penyelesaian keliru di bawah memperlihatkan mengapa sebenarnya tidak. Subpohon kiri dan kanan 6 memberikan 9 dan 34, dan $6 + 9 + 34 = 49$ memeriksa jawabannya dengan cara kedua. Pohon kosong memberikan 0. **Biaya.** $\Theta(n)$ menurut Persamaan (12.1), dan stack $O(h)$. (Untuk pohon yang sangat besar dengan nilai-nilai besar, sebuah `int` dapat meluap; jumlah bertipe `long long` akan menjadi pilihan yang hati-hati. Nilai-nilai kita kecil.)

12.6.5 Penyelesaian Java dan Python

Kode 12.14 dan **Kode 12.15** adalah versi Java dan Python-nya; keduanya mencetak baris-baris yang persis sama.

Kode 12.14 `P1203Sum.java`: Soal 12.3 dalam Java.

```java
// P1203Sum.java - Problem 12.3: the sum of all the values in a binary tree.
public class P1203Sum extends TreeKit {

    static int sumOfNodes(Node root) {
        if (root == null) return 0;          // the empty tree adds nothing
        return root.data + sumOfNodes(root.left) + sumOfNodes(root.right);
    }

    public static void main(String[] args) {
        Node root = buildSample();
        System.out.println("sum of the whole tree  : " + sumOfNodes(root));
        System.out.println("sum again (same answer): " + sumOfNodes(root));
        System.out.println("left subtree of 6      : " + sumOfNodes(root.left));
        System.out.println("right subtree of 6     : " + sumOfNodes(root.right));
        System.out.println("empty tree             : " + sumOfNodes(null));
    }
}
```

Kode 12.15 `p1203_sum.py`: Soal 12.3 dalam Python.

```python
# p1203_sum.py - Problem 12.3: the sum of all the values in a binary tree.
from tree_kit import *


def sum_of_nodes(root):
    if root is None:                    # the empty tree adds nothing
        return 0
    return root.data + sum_of_nodes(root.left) + sum_of_nodes(root.right)


def main():
    root = build_sample()
    print("sum of the whole tree  :", sum_of_nodes(root))
    print("sum again (same answer):", sum_of_nodes(root))
    print("left subtree of 6      :", sum_of_nodes(root.left))
    print("right subtree of 6     :", sum_of_nodes(root.right))
    print("empty tree             :", sum_of_nodes(None))


if __name__ == "__main__":
    main()
```

12.6.6 Penyelesaian keliru yang menggoda

Pemrogram yang belajar menjumlahkan bilangan dengan perulangan sering membawa kebiasaan perulangan itu ke dalam rekursi: menyimpan jumlah berjalan dalam variabel di luar fungsi, dan membiarkan setiap pemanggilan menambahkan ke dalamnya.

```
int total = 0;                  // WRONG: a running total outside the function

int sumOfNodes(Node *root) {
    if (root == nullptr) return total;
    total += root->data;
    sumOfNodes(root->left);
    sumOfNodes(root->right);
    return total;
}
```

Pemanggilan pertama bahkan memberikan jawaban yang benar, dan itulah yang membuat kesalahan ini begitu menggoda. Program `wrong/p1203_wrong.cpp` mencetak:

```
sum of the whole tree  : 49
sum again (same answer): 98
left subtree of 6      : 107
right subtree of 6     : 141
empty tree             : 141
```

Variabel `total` hidup *di luar* setiap pemanggilan, sehingga ia mengingat semua yang pernah ditambahkan oleh pemanggilan sebelumnya. Pemanggilan kedua pada seluruh pohon berangkat dari 49 dan berakhir di 98; pemanggilan pada subpohon kiri menambahkan 9 ke 98 dan melaporkan 107; bahkan pohon kosong melaporkan 141. Fungsi yang jawabannya bergantung pada apa yang dipanggil sebelumnya tidak lagi menghitung sifat pohon. Versi yang benar tidak menyimpan apa pun di antara pemanggilan: setiap pemanggilan *mengembalikan* jumlahnya, dan induknya menjumlahkan nilai-nilai yang dikembalikan. Aturan umumnya adalah Langkah 4 metode kita: fungsi rekursif menggabungkan apa yang *dikembalikan* pemanggilan-pemanggilannya; fungsi itu tidak berkomunikasi melalui variabel yang mereka pakai bersama.

12.6.7 Uji diri

> **UJI DIRI.** Tulislah `int maxValue(Node *root)` untuk pohon biner *apa pun* (tidak harus pohon pencarian), yang mengembalikan `INT_MIN` untuk pohon kosong. Mengapa `INT_MIN` adalah *base case* yang tepat di sini, padahal untuk jumlah yang tepat adalah 0?
>
> *Jawaban.* `if (root == nullptr) return INT_MIN; return max(root->data, max(maxValue(root->left), maxValue(root->right)));`. *Base case* harus berupa nilai yang tidak mengubah apa pun di bawah operasi penggabungannya. Untuk $+$ nilai itu 0; untuk max nilai itu adalah nilai terkecil yang mungkin, karena $\max(x,\text{INT\_MIN}) = x$ untuk setiap `int` x. *Base case* 0 akan memberikan maksimum yang salah untuk pohon yang semua nilainya negatif. (Dalam pohon *pencarian*, maksimumnya cukup simpul paling kanan, yang ditemukan dalam $O(h)$ oleh fungsi `max` pada Subbab 11.7.4.)

12.7 Soal 12.4 — Melipattigakan Setiap Simpul

12.7.1 Soalnya

> **SOAL.** Write a recursive function which multiplies by three every node of the tree. `void triple(Node *root)`
>
> *Dengan kata lain:* tulislah fungsi rekursif yang mengalikan nilai setiap simpul pohon dengan tiga.

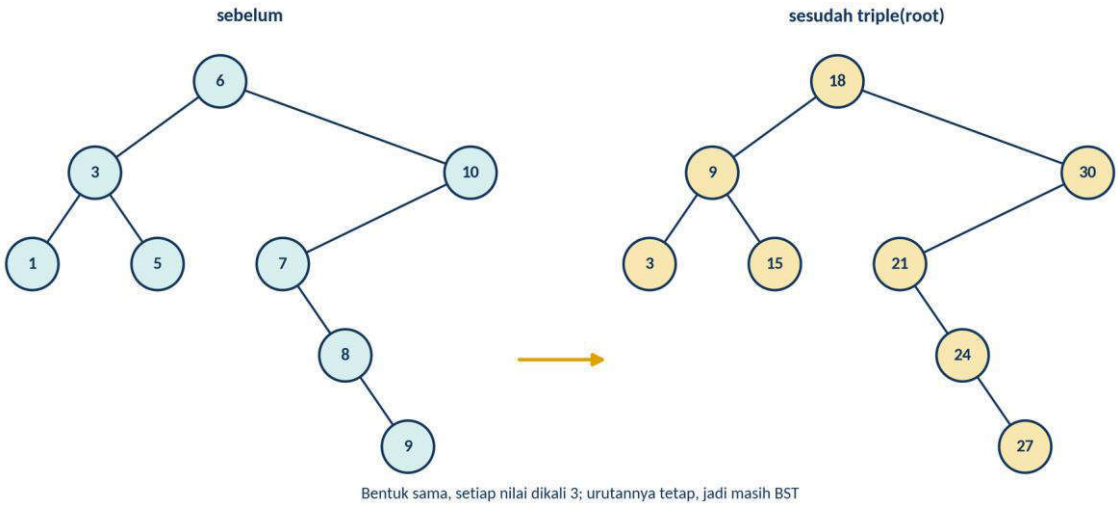
12.7.2 Dengan kata-kata kita sendiri

Kunjungi setiap simpul sekali dan ganti nilainya dengan tiga kali nilai itu. Fungsi ini tidak mengembalikan apa pun; seluruh pengaruhnya ada pada simpul-simpulnya. Karena setiap simpul diubah secara terpisah dari yang lain, urutan kunjungannya tidak penting; kita memakai *preorder* (ubah simpulnya, lalu kedua subpohonnya) karena terbaca wajar. Hasil yang diharapkan: *preorder* `6 3 1 5 10 7 8 9` menjadi `18 9 3 15 30 21 24 27`.

Satu pertanyaan layak diajukan sebelum kita menulis apa pun: **apakah pohon yang dilipattigakan itu masih *binary search tree*?** Masih, dan alasannya satu baris aljabar. Mengalikan dengan bilangan positif mempertahankan urutan dua bilangan mana pun:

$$a < b \quad \Rightarrow \quad 3a < 3b. \tag{12.6}$$

Setiap perbandingan yang dituntut Persamaan (12.3) antara sebuah simpul dan keturunannya sebelumnya benar, jadi menurut Persamaan (12.6) perbandingan itu tetap benar sesudahnya. **Gambar 12.5** memperlihatkan pohon sebelum dan sesudahnya: bentuk sama, nilai baru, tetap berurutan. (Mengalikan dengan -3 akan membalik setiap perbandingan dan merusak pohon pencarian; Soal 12.5 merusaknya dengan cara lain.)



Gambar 12.5 Soal 12.4: pohon sebelum dan sesudah `triple(root)`. Bentuknya tidak berubah, dan karena $a < b$ mengakibatkan $3a < 3b$, pohon yang dilipattigakan tetap *binary search tree*.

12.7.3 Penyelesaian C++

Kode 12.16 adalah program C++ untuk Soal 12.4.

Kode 12.16 `p1204_triple.cpp`: Soal 12.4, mengubah simpul melalui pointer.

```cpp
// p1204_triple.cpp - Problem 12.4: multiply every value in the tree by three.
#include "tree_kit.hpp"

void triple(Node *root) {
    if (root == nullptr) return;
    root->data *= 3;  // change the node itself, through the pointer
    triple(root->left);
    triple(root->right);
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    printf("before, preorder: ");
    preOrder(root);
    triple(root);
    printf("\nafter,  preorder: ");
    preOrder(root);
    printf("\nafter,  inorder : ");
    inOrder(root);
    printf("\nsearch 24: %s, search 8: %s\n",
           search(root, 24) != nullptr ? "found" : "not found",
           search(root, 8) != nullptr ? "found" : "not found");
    clear(&root);
    return 0;
}
```

```
before, preorder: 6 3 1 5 10 7 8 9
after,  preorder: 18 9 3 15 30 21 24 27
after,  inorder : 3 9 15 18 21 24 27 30
search 24: found, search 8: not found
```

Baris demi baris. *Preorder* sesudah pelipattigaan tepat tiga kali *preorder* sebelumnya. Baris *inorder* tetap menaik, yang merupakan cara program memastikan Persamaan (12.6); dan karena pohonnya tetap pohon pencarian, `search` menemukan 24 (ia mengikuti $24 > 18$, $24 < 30$, $24 > 21$, ditemukan) dan dengan benar gagal menemukan 8, yang tidak lagi ada di pohon. Baris

kuncinya adalah `root->data *= 3`: fungsi menerima *pointer* ke simpul, dan menulis melalui pointer itu mengubah simpulnya sendiri, simpul yang hidup di dalam pohon. **Biaya.** $\Theta(n)$.

12.7.4 Penyelesaian Java dan Python

Kode 12.17 dan **Kode 12.18** adalah versi Java dan Python-nya; keduanya mencetak baris-baris yang persis sama.

Kode 12.17 `P1204Triple.java`: Soal 12.4 dalam Java.

```java
// P1204Triple.java - Problem 12.4: multiply every value in the tree by three.
public class P1204Triple extends TreeKit {

    static void triple(Node root) {
        if (root == null) return;
        root.data *= 3;  // change the node itself, through the reference
        triple(root.left);
        triple(root.right);
    }

    public static void main(String[] args) {
        Node root = buildSample();
        System.out.print("before, preorder: ");
        preOrder(root);
        triple(root);
        System.out.print("\nafter,  preorder: ");
        preOrder(root);
        System.out.print("\nafter,  inorder : ");
        inOrder(root);
        System.out.printf("%nsearch 24: %s, search 8: %s%n",
                search(root, 24) != null ? "found" : "not found",
                search(root, 8) != null ? "found" : "not found");
    }
}
```

Kode 12.18 `p1204_triple.py`: Soal 12.4 dalam Python.

```python
# p1204_triple.py - Problem 12.4: multiply every value in the tree by three.
from tree_kit import *


def triple(root):
    if root is None:
        return
    root.data *= 3                      # change the node object itself
    triple(root.left)
    triple(root.right)


def main():
    root = build_sample()
    print("before, preorder: ", end="")
    pre_order(root)
    triple(root)
    print("\nafter,  preorder: ", end="")
    pre_order(root)
    print("\nafter,  inorder : ", end="")
    in_order(root)
    found = lambda x: "found" if search(root, x) is not None else "not found"
    print(f"\nsearch 24: {found(24)}, search 8: {found(8)}")


if __name__ == "__main__":
    main()
```

Dalam Java dan Python, variabel bertipe `Node` memuat *referensi* ke simpul, tidak pernah salinannya, sehingga `root.data *= 3` mengubah simpul di dalam pohon, persis seperti pointer C++. Karena itulah fungsi ini tidak memerlukan kehati-hatian tambahan dalam kedua bahasa itu.

12.7.5 Penyelesaian keliru yang menggoda

Namun dalam C++, perubahan kecil pada tipe parameternya membuat fungsi itu sama sekali tidak berbuat apa-apa:

```
void triple(Node node) {                // WRONG: receives a copy of the node
    node.data *= 3;                     // triples the copy only
    if (node.left != nullptr) triple(*node.left);
    if (node.right != nullptr) triple(*node.right);
}
```

Dipanggil sebagai `triple(*root)`, kode ini terkompilasi bersih dan `wrong/p1204_wrong.cpp` mencetak:

```
before, preorder: 6 3 1 5 10 7 8 9
after,  preorder: 6 3 1 5 10 7 8 9
after,  inorder : 1 3 5 6 7 8 9 10
search 24: not found, search 8: found
```

Tidak ada yang berubah. Parameter bertipe `Node` diterima *by value* (Bab 6, dan Soal 8.9): fungsi mendapat salinan baru dari ketiga medan simpul, melipattigakan `data` salinan itu, dan membuang salinannya ketika kembali. Medan `left` dan `right` salinan itu tetap menunjuk ke anak-anak yang sesungguhnya, sehingga rekursinya mencapai setiap simpul; hanya saja ia tidak pernah mengubah satu pun. Obatnya adalah menerima simpul melalui pointer, seperti tanda tangan `triple(Node *root)` pada soal, atau melalui referensi, `triple(Node &node)`.

12.7.6 Uji diri

> **UJI DIRI.** Tulislah `void addToAll(Node *root, int d)` yang menambahkan `d` ke setiap nilai. Untuk nilai `d` berapa saja hasilnya tetap *binary search tree*? Dan untuk `void negate(Node *root)`, yang mengganti setiap nilai x dengan $-x$, langkah tambahan apa yang mengembalikan hasilnya menjadi pohon pencarian?
>
> *Jawaban.* Menambahkan `d` yang sama mempertahankan setiap perbandingan ($a < b \Rightarrow a + d < b + d$), sehingga hasilnya pohon pencarian untuk *setiap* `d`. Penegatifan membalik setiap perbandingan, sehingga sesudah `negate` setiap subpohon kiri memuat nilai-nilai yang lebih besar; menukar kedua anak setiap simpul, yang tepat merupakan cermin pada Soal 12.5, mengembalikannya menjadi pohon pencarian.

12.8 Soal 12.5 — Bayangan Cermin

12.8.1 Soalnya

> **SOAL.** Write a recursive function to create a mirror image of the binary tree. `void mirror(Node *root)`
>
> *Dengan kata lain:* tulislah fungsi rekursif yang mengubah pohon biner menjadi bayangan cerminnya.

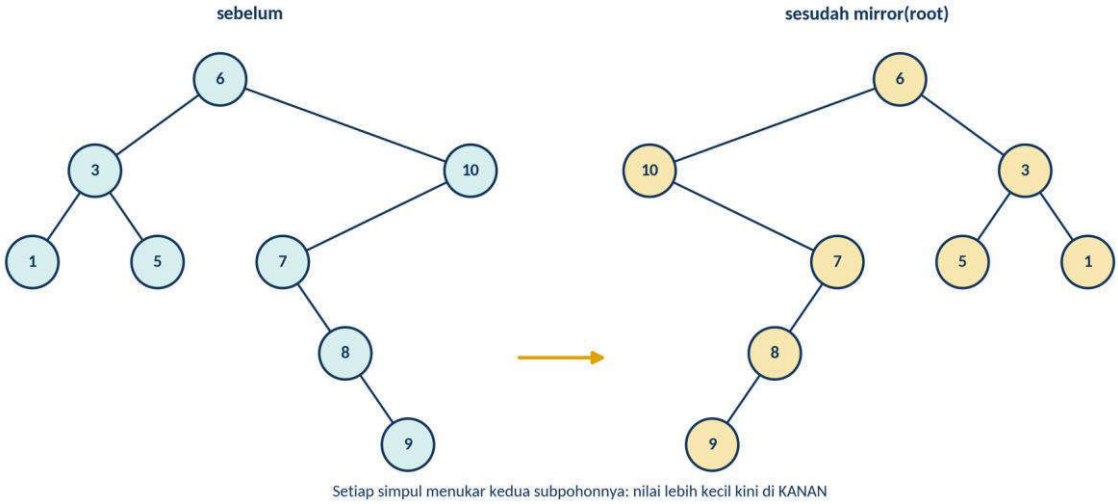
12.8.2 Dengan kata-kata kita sendiri

Hadapkan pohon ke cermin: yang tadinya di kiri kini di kanan, di setiap tingkat. Jadi di setiap simpul, subpohon kiri dan subpohon kanan bertukar tempat, dan masing-masing juga

dicerminkan. Menurut kotak wawasan, itulah seluruh kasus rekursifnya: *tukar kedua anak akar, lalu cerminkan masing-masing. Base case*-nya: cermin dari pohon kosong adalah pohon kosong, jadi tidak ada yang perlu dikerjakan. Fungsi ini mengubah pohon di tempat dan tidak mengembalikan apa pun.

12.8.3 Dengan tangan, selangkah demi selangkah

Gambar 12.6 memperlihatkan pohon sebelum dan sesudahnya.



Gambar 12.6 Soal 12.5: pohon sebelum dan sesudah `mirror(root)`. Setiap simpul telah menukar kedua subpohonnya, sehingga nilai-nilai yang lebih kecil kini tergantung di kanan; hasilnya bukan lagi *binary search tree*.

1. Di 6: tukar. Kini kiri 6 adalah subpohon 10 dan kanannya subpohon 3.
2. Cerminkan subpohon kiri yang baru, yang berakar di 10. Di 10: kirinya tadinya 7 dan kanannya kosong; sesudah ditukar, kirinya kosong dan kanannya 7. Cerminkan 7: kirinya tadinya kosong dan kanannya 8; sesudah ditukar, 8 menjadi anak *kiri*-nya. Cerminkan 8: 9 pindah dari kanannya ke kirinya. Cerminkan 9: sebuah daun, tidak ada yang ditukar.
3. Cerminkan subpohon kanan yang baru, berakar di 3: 1 dan 5 bertukar tempat, dan keduanya daun.

Preorder cerminnya adalah `6 10 7 8 9 3 5 1` (jawaban uji diri Soal 12.1), dan *inorder*-nya adalah *inorder* semula yang dibalik, `10 9 8 7 6 5 3 1`, karena *inorder* mengunjungi "kiri, simpul, kanan" dan cermin telah menukar kiri dan kanan di mana-mana.

12.8.4 Penyelesaian C++

Kode 12.19 adalah program C++ untuk Soal 12.5.

Kode 12.19 `p1205_mirror.cpp`: Soal 12.5, penukaran dengan pointer yang disimpan di setiap simpul.

```cpp
// p1205_mirror.cpp - Problem 12.5: turn a binary tree into its mirror image.
#include "tree_kit.hpp"

void mirror(Node *root) {
    if (root == nullptr) return;
    Node *saved = root->left;           // keep the left subtree reachable
    root->left = root->right;
    root->right = saved;                // the two subtrees have changed places
    mirror(root->left);                 // now mirror each of them
    mirror(root->right);
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    mirror(root);
    printf("The mirrored tree, turned on its side:\n");
    showSideways(root);
    printf("preorder: ");
    preOrder(root);
    printf("\ninorder : ");
    inOrder(root);
    printf("\nsearch 3 in the mirror: %s\n",
           search(root, 3) != nullptr ? "found" : "not found");
    mirror(root);                       // mirroring twice gives the original back
    printf("mirrored twice, inorder: ");
    inOrder(root);
    printf("\n");
    clear(&root);
    return 0;
}
```

```
The mirrored tree, turned on its side:
        1
    3
        5
6
        7
            8
                9
    10
preorder: 6 10 7 8 9 3 5 1
inorder : 10 9 8 7 6 5 3 1
search 3 in the mirror: not found
mirrored twice, inorder: 1 3 5 6 7 8 9 10
```

Baris demi baris. Gambar menyampingnya, bila diputar seperempat putaran searah jarum jam, adalah bagian kanan **Gambar 12.6**. *Preorder* dan *inorder*-nya sesuai dengan perhitungan tangan. Lalu datang sebuah baris yang perlu direnungkan: **mencari 3 di dalam cermin gagal**, walaupun 3 jelas ada di pohon. `search` bergantung pada Persamaan (12.3): ia membandingkan $3 < 6$ lalu ke *kiri*, tempat cermin kini menyimpan nilai-nilai yang lebih besar 10, 7, 8, 9, dan ia jatuh keluar dari pohon tanpa pernah melihat 3. Cermin sebuah pohon pencarian adalah pohon biner yang sah, tetapi bukan pohon pencarian, sehingga fungsi-fungsi BST dari Bab 11 tidak boleh dipakai padanya. Baris terakhir mencerminkan pohon itu untuk kedua kalinya dan mendapatkan kembali *inorder* semula: pencerminan adalah invers dirinya sendiri. **Biaya.** $\Theta(n)$, tiga penugasan per simpul.

12.8.5 Penyelesaian Java dan Python

Kode 12.20 dan **Kode 12.21** adalah versi Java dan Python-nya; keduanya mencetak baris-baris yang persis sama.

Kode 12.20 `P1205Mirror.java`: Soal 12.5 dalam Java.

```java
// P1205Mirror.java - Problem 12.5: turn a binary tree into its mirror image.
public class P1205Mirror extends TreeKit {

    static void mirror(Node root) {
        if (root == null) return;
        Node saved = root.left;              // keep the left subtree reachable
        root.left = root.right;
        root.right = saved;  // the two subtrees have changed places
        mirror(root.left);                   // now mirror each of them
        mirror(root.right);
    }

    public static void main(String[] args) {
        Node root = buildSample();
        mirror(root);
        System.out.println("The mirrored tree, turned on its side:");
        showSideways(root);
        System.out.print("preorder: ");
        preOrder(root);
        System.out.print("\ninorder : ");
        inOrder(root);
        System.out.println("\nsearch 3 in the mirror: "
                + (search(root, 3) != null ? "found" : "not found"));
        mirror(root);  // mirroring twice gives the original back
        System.out.print("mirrored twice, inorder: ");
        inOrder(root);
        System.out.println();
    }
}
```

Kode 12.21 `p1205_mirror.py`: Soal 12.5 dalam Python, dengan penukaran ditulis sebagai satu penugasan *tuple*.

```python
# p1205_mirror.py - Problem 12.5: turn a binary tree into its mirror image.
from tree_kit import *


def mirror(root):
    if root is None:
        return
    root.left, root.right = root.right, root.left  # both old values read first
    mirror(root.left)                    # now mirror each of them
    mirror(root.right)


def main():
    root = build_sample()
    mirror(root)
    print("The mirrored tree, turned on its side:")
    show_sideways(root)
    print("preorder: ", end="")
    pre_order(root)
    print("\ninorder : ", end="")
    in_order(root)
    found = "found" if search(root, 3) is not None else "not found"
    print("\nsearch 3 in the mirror:", found)
    mirror(root)                           # mirroring twice gives the original back
    print("mirrored twice, inorder: ", end="")
    in_order(root)
    print()


if __name__ == "__main__":
    main()
```

Versi Python menukar dengan `root.left, root.right = root.right, root.left`. Cara ini aman karena Python mengevaluasi seluruh ruas kanan, kedua nilai lamanya, *sebelum* menugaskan apa pun [10]; itulah pointer simpanan yang sama seperti di C++, dituliskan untuk kita oleh bahasanya.

12.8.6 Penyelesaian keliru yang menggoda

Kesalahan klasiknya adalah menukar dua variabel dengan dua penugasan tanpa salinan yang disimpan:

```
root->left = root->right;            // WRONG: the old left subtree is lost
root->right = root->left;            // ...and this copies the NEW left pointer
```

Sesudah baris pertama, kedua pointer menunjuk ke subpohon kanan; baris kedua menyalin pointer yang sama itu kembali. Subpohon kiri kini tidak terjangkau, dan subpohon kanan tergantung di *kedua* sisi. Menjalankan `wrong/p1205_wrong.cpp` di bawah *sanitizer* menghasilkan hal berikut (laporannya dipersingkat menjadi baris-baris yang paling berguna):

```
The mirrored tree, turned on its side:
    10
6
    10
preorder: 6 10 10
inorder : 10 6 10
search 3 in the mirror: not found
mirrored twice, inorder: 10 6 10
==27839==ERROR: AddressSanitizer: heap-use-after-free on address 0x5030000000a8 at pc
0x55e840816e4f bp 0x7ffd7e794e60 sp 0x7ffd7e794e50
READ of size 8 at 0x5030000000a8 thread T0
    #0 0x55e840816e4e in clear(Node**) ../tree_kit.hpp:29
    #3 0x55e8408168d4 in main p1205_wrong.cpp:28
freed by thread T0 here:
    #1 0x55e840817006 in clear(Node**) ../tree_kit.hpp:32
SUMMARY: AddressSanitizer: heap-use-after-free ../tree_kit.hpp:29 in clear(Node**)
```

Tiga hal salah, dan keluarannya memperlihatkan ketiganya. Kesalahan yang sama di simpul 10 membuat 10 kehilangan satu-satunya anaknya, 7, sehingga 7, 8, dan 9 ikut lenyap, dan "pohon" yang tercetak adalah 6 dengan simpul 10 tergantung di kedua sisinya. Enam simpul, 3, 1, dan 5 lalu 7, 8, dan 9, tidak terjangkau: sebuah kebocoran. Dan ketika `clear` membebaskan pohon, ia membebaskan simpul 10 melalui pointer kiri 6 lalu, mengikuti pointer kanan 6, membaca simpul yang baru saja dibebaskannya: *use after free*, persis seperti yang dikatakan laporannya, dengan baris yang membacanya (`clear`, baris 29) dan baris yang telah membebaskannya (baris 32). Obatnya adalah aturan Bab 8 di tempat yang baru: sebelum menimpa sebuah pointer, simpanlah apa yang ditunjuknya.

12.8.7 Uji diri

> **UJI DIRI.** Apakah urutannya penting? Tulis ulang `mirror` sehingga fungsi itu lebih dulu mencerminkan kedua subpohon dan *baru kemudian* menukarnya (versi *postorder*). Apakah hasilnya sama? Lalu tulislah `bool isMirror(Node *a, Node *b)`, yang menyatakan apakah pohon `b` adalah bayangan cermin pohon `a`.
>
> *Jawaban.* Versi *postorder* memberikan pohon yang persis sama: entah kedua subpohon dicerminkan sebelum atau sesudah ditukar, masing-masing berakhir tercermin dan di sisi yang lain. `isMirror`: `if (a == nullptr || b == nullptr) return a == b; return a->data == b->data && isMirror(a->left, b->right) && isMirror(a->right, b->left);`. Perhatikan bagaimana fungsi itu membandingkan kiri dengan kanan: persilangan itulah seluruh definisi sebuah cermin.

12.9 Soal 12.6 — Apakah Pohonnya Seimbang Tingginya?

12.9.1 Soalnya

> **SOAL.** Write a recursive function to check if the binary tree is height-balanced (the difference between the heights of the left and right subtrees of any node is not more than 1). `int isBalanced(Node *root)` You may use the helper function `int isBalancedHelper(Node *root)`, where you use the function `int height(Node *root)` that is already defined.
>
> *Dengan kata lain:* tulislah fungsi rekursif yang memeriksa apakah pohon biner seimbang tingginya (selisih tinggi subpohon kiri dan kanan di simpul mana pun tidak lebih dari 1); Anda boleh memakai fungsi bantu `isBalancedHelper` yang memanfaatkan `height` yang sudah ada.

12.9.2 Dengan kata-kata kita sendiri

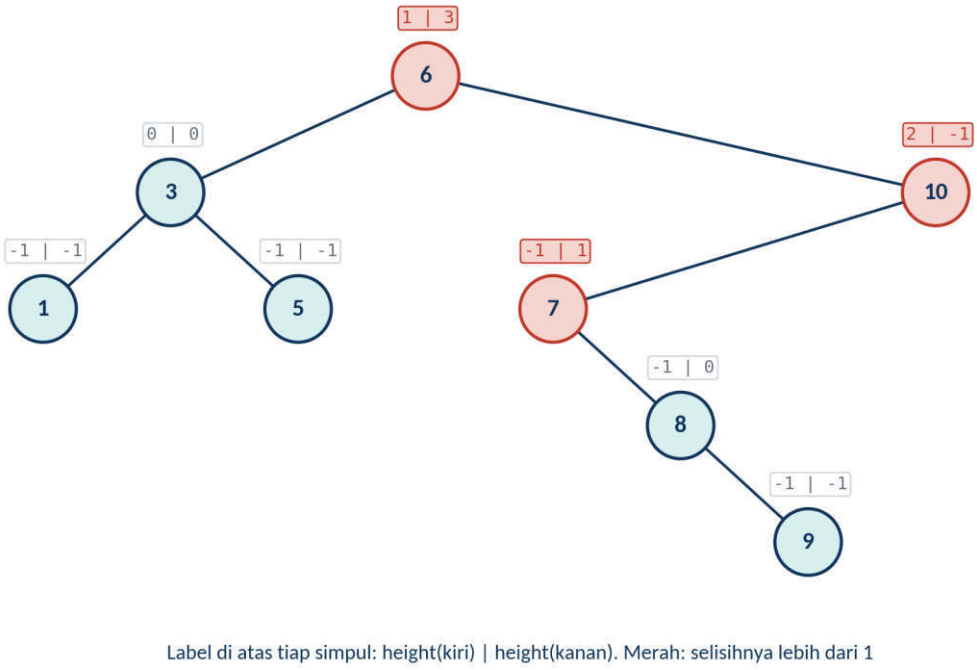
Sebuah simpul *seimbang* bila tinggi kedua subpohonnya berselisih paling banyak 1. *Pohon*-nya seimbang tingginya bila **setiap** simpul seimbang, bukan hanya akarnya. Fungsinya menjawab 1 (ya) atau 0 (tidak), sebagaimana disiratkan tipe kembalian `int` pada soal.

> **DEFINISI.** Pohon biner T *seimbang tingginya* (*height-balanced*) jika T kosong, atau jika $|h(T.left) - h(T.right)| \leq 1$ dan $T.left$ serta $T.right$ keduanya seimbang tingginya. Pohon dengan sifat ini adalah bentuk yang dijaga pohon AVL pada Subbab 11.9.2, karya Adelson-Velsky dan Landis [11].

Definisi itu sudah merupakan fungsi rekursif, dengan *base case* pada kata-kata pertamanya: **pohon kosong itu seimbang**. Ingatlah kalimat itu; penyelesaian keliru di bawah salah justru di situ.

12.9.3 Dengan tangan, selangkah demi selangkah

Gambar 12.7 menuliskan di atas setiap simpul tinggi kedua subpohonnya, yang dihitung dengan Persamaan (12.2).



Gambar 12.7 Soal 12.6: di atas setiap simpul, tinggi subpohon kirinya dan subpohon kanannya. Ketiga simpul merah melanggar aturan, sehingga pohonnya tidak seimbang tingginya.

- Daun 1, 5, dan 9 memiliki dua subpohon kosong: $|-1-(-1)| = 0$. Seimbang.
- Simpul 3 memiliki dua daun: $|0-0| = 0$. Seimbang.
- Simpul 8 memiliki kiri kosong dan daun 9 di kanan: $|-1-0| = 1$. Seimbang, pas-pasan.
- Simpul 7 memiliki kiri kosong dan subpohon 8–9, bertinggi 1, di kanan: $|-1-1| = 2$. **Tidak seimbang.**
- Simpul 10 memiliki subpohon 7–8–9, bertinggi 2, di kiri dan tidak ada apa-apa di kanan: $|2-(-1)| = 3$. **Tidak seimbang.**
- Akar 6 memiliki tinggi 1 dan 3: $|1-3| = 2$. **Tidak seimbang.**

Jadi jawaban untuk pohon kita adalah 0. Sebaliknya, pohon `6 3 9 1 5 8 10` lengkap (setiap tingkatnya penuh), kedua tinggi di setiap simpulnya sama, dan jawabannya 1. Pengujian yang paling mendidik adalah pohon yang akarnya seimbang tetapi menyembunyikan simpul tak seimbang di bawah, seperti pohon yang dibangun dari 5, 3, 8, 1, 4, 7, 9, 0, 10, 11: akarnya memiliki subpohon bertinggi 2 dan 3, yang tidak masalah, tetapi simpul 8 memiliki tinggi 0 dan 2. Fungsi yang hanya memeriksa akar akan keliru menjawab ya.

12.9.4 Penyelesaian C++

Kode 12.22 memuat dua penyelesaian. Yang pertama adalah yang diminta soal: `isBalancedHelper` memeriksa simpulnya dengan dua pemanggilan `height`, lalu memeriksa kedua subpohonnya. Yang kedua, `checkedHeight`, menghitung tinggi dan putusannya dalam penelusuran yang sama: ia mengembalikan tinggi subpohon yang seimbang, atau sebuah nilai khusus, `UNBALANCED`, begitu menemukan simpul yang melanggar aturan.

Kode 12.22 `p1206_balanced.cpp`: Soal 12.6, fungsi bantu yang memakai `height`, dan versi satu-lintasan.

```
// p1206_balanced.cpp - Problem 12.6: is every node height-balanced?
#include <cstdlib>
#include "tree_kit.hpp"

// 1 if every node of this subtree has |height(left) - height(right)| <= 1.
int isBalancedHelper(Node *root) {
    if (root == nullptr) return 1;       // the empty tree is balanced
    int diff = abs(height(root->left) - height(root->right));
    if (diff > 1) return 0;              // this node breaks the rule
    return isBalancedHelper(root->left) && isBalancedHelper(root->right);
}

int isBalanced(Node *root) {
    return isBalancedHelper(root);
}

// One pass: return the height, or UNBALANCED as soon as a node breaks the rule.
const int UNBALANCED = -2;               // not -1: -1 already means "empty tree"

int checkedHeight(Node *root) {
    if (root == nullptr) return -1;
    int u = checkedHeight(root->left);
    if (u == UNBALANCED) return UNBALANCED;
    int v = checkedHeight(root->right);
    if (v == UNBALANCED) return UNBALANCED;
    if (abs(u - v) > 1) return UNBALANCED;
    return (u > v ? u : v) + 1;
}

void report(const char *name, Node *root) {
    printf("%-22s isBalanced = %d, one-pass = %d\n", name, isBalanced(root),
           checkedHeight(root) != UNBALANCED ? 1 : 0);
}

int main() {
    Node *sample = nullptr, *full = nullptr, *chain = nullptr, *empty = nullptr;
    Node *deep = nullptr;
    buildSample(&sample);                            // 6 3 10 1 5 7 8 9
    insertAll(&full, {6, 3, 9, 1, 5, 8, 10});
    insertAll(&chain, {1, 2, 3});
    insertAll(&deep, {5, 3, 8, 1, 4, 7, 9, 0, 10, 11});
    report("sample tree", sample);
    report("6 3 9 1 5 8 10", full);
    report("chain 1 2 3", chain);
    report("empty tree", empty);
    report("5 3 8 1 4 7 9 0 10 11", deep);  // the root is fine, node 8 is not
```

Kode 12.22 (Lanjutan) *p1206_balanced.cpp: Soal 12.6, fungsi bantu yang memakai height, dan versi satu-lintasan.*

```
    clear(&sample);
    clear(&full);
    clear(&chain);
    clear(&deep);
    return 0;
}
```

```
sample tree            isBalanced = 0, one-pass = 0
6 3 9 1 5 8 10         isBalanced = 1, one-pass = 1
chain 1 2 3            isBalanced = 0, one-pass = 0
empty tree             isBalanced = 1, one-pass = 1
5 3 8 1 4 7 9 0 10 11  isBalanced = 0, one-pass = 0
```

Baris demi baris. Kedua versi sepakat pada kelima pohon. Pohon contoh kita tidak seimbang; pohon lengkap seimbang; rantai `1 2 3` tidak (akarnya bertinggi −1 dan 1); pohon kosong

seimbang, menurut definisinya; dan pohon dengan simpul 8 yang tersembunyi tak seimbang ditolak dengan benar.

Mengapa `UNBALANCED` bernilai -2 dan bukan -1. Sebagian buku menandai "tidak seimbang" dengan -1, tetapi di sini -1 adalah tinggi pohon kosong, sehingga setiap subpohon kosong akan tampak tidak seimbang. Tinggi tidak pernah di bawah -1, jadi -2 aman.

Biaya. Versi satu-lintasan mengunjungi setiap simpul sekali: $\Theta(n)$ menurut Persamaan (12.1). Versi dengan fungsi bantu lebih lambat, karena di setiap simpul yang diperiksanya ia memanggil `height` pada kedua subpohon, dan `height` menelusuri seluruh subpohon itu. Dengan $size(x)$ sebagai banyaknya simpul di subpohon x, biayanya sebanding dengan

$$C = \sum_{x \text{ diperiksa}} size\,(x) = \sum_{x \text{ diperiksa}} (d(x) + 1) \le n\,(h + 1), \tag{12.7}$$

dengan $d(x)$ kedalaman x. Langkah tengahnya menghitung hal yang sama dengan cara kedua: sebuah simpul y berada di dalam subpohon setiap leluhurnya dan subpohonnya sendiri, seluruhnya $d(y) + 1$ subpohon. Persamaan (12.7) menyatakan bahwa versi fungsi bantu berbiaya $O(nh)$. Itu tidak seburuk kelihatannya, karena `&&` menghentikan pemeriksaan pada simpul tak seimbang yang pertama, dan pohon yang semua simpul terperiksanya seimbang memiliki tinggi paling banyak sekitar $1.44 \lg n$ [11]; jadi versi fungsi bantu berbiaya $O(n \log n)$, dibandingkan $\Theta(n)$ untuk versi satu-lintasan. Pada pohon sejuta simpul, itulah perbedaan antara sekitar dua puluh juta langkah dan sejuta langkah.

12.9.5 Penyelesaian Java dan Python

Kode 12.23 dan **Kode 12.24** adalah versi Java dan Python-nya; keduanya mencetak baris-baris yang persis sama.

Kode 12.23 `P1206Balanced.java`: Soal 12.6 dalam Java.

```java
// P1206Balanced.java - Problem 12.6: is every node height-balanced?
public class P1206Balanced extends TreeKit {

    // true if every node of this subtree has |height(left) - height(right)| <= 1.
    static boolean isBalancedHelper(Node root) {
        if (root == null) return true;        // the empty tree is balanced
        int diff = Math.abs(height(root.left) - height(root.right));
        if (diff > 1) return false;           // this node breaks the rule
        return isBalancedHelper(root.left) && isBalancedHelper(root.right);
    }

    static int isBalanced(Node root) {
        return isBalancedHelper(root) ? 1 : 0;
    }

    // One pass: the height, or UNBALANCED as soon as a node breaks the rule.
    // UNBALANCED is -2, not -1: -1 already means "empty tree".
    static final int UNBALANCED = -2;

    static int checkedHeight(Node root) {
        if (root == null) return -1;
        int u = checkedHeight(root.left);
        if (u == UNBALANCED) return UNBALANCED;
        int v = checkedHeight(root.right);
        if (v == UNBALANCED) return UNBALANCED;
        if (Math.abs(u - v) > 1) return UNBALANCED;
        return Math.max(u, v) + 1;
    }

    static void report(String name, Node root) {
        System.out.printf("%-22s isBalanced = %d, one-pass = %d%n", name,
                isBalanced(root), checkedHeight(root) != UNBALANCED ? 1 : 0);
    }

    public static void main(String[] args) {
        report("sample tree", buildSample());
        report("6 3 9 1 5 8 10", insertAll(null, 6, 3, 9, 1, 5, 8, 10));
        report("chain 1 2 3", insertAll(null, 1, 2, 3));
        report("empty tree", null);
        report("5 3 8 1 4 7 9 0 10 11",
                insertAll(null, 5, 3, 8, 1, 4, 7, 9, 0, 10, 11));
    }
}
```

Kode 12.24 `p1206_balanced.py`: Soal 12.6 dalam Python.

```python
# p1206_balanced.py - Problem 12.6: is every node height-balanced?
from tree_kit import *


def is_balanced_helper(root):
    """True if every node here has |height(left) - height(right)| <= 1."""
    if root is None:                    # the empty tree is balanced
        return True
    if abs(height(root.left) - height(root.right)) > 1:
        return False                    # this node breaks the rule
    return is_balanced_helper(root.left) and is_balanced_helper(root.right)


def is_balanced(root):
    return 1 if is_balanced_helper(root) else 0


UNBALANCED = -2                         # not -1: -1 already means "empty tree"


def checked_height(root):
    """One pass: the height, or UNBALANCED as soon as a node breaks the rule."""
    if root is None:
        return -1
    u = checked_height(root.left)
    if u == UNBALANCED:
        return UNBALANCED
    v = checked_height(root.right)
    if v == UNBALANCED:
        return UNBALANCED
    if abs(u - v) > 1:
        return UNBALANCED
    return max(u, v) + 1


def report(name, root):
    one_pass = 1 if checked_height(root) != UNBALANCED else 0
    print(f"{name:<22} isBalanced = {is_balanced(root)}, one-pass = {one_pass}")
```

Kode 12.24 (Lanjutan) *p1206_balanced.py: Soal 12.6 dalam Python.*

```python
def main():
    report("sample tree", build_sample())
    report("6 3 9 1 5 8 10", insert_all(None, [6, 3, 9, 1, 5, 8, 10]))
    report("chain 1 2 3", insert_all(None, [1, 2, 3]))
    report("empty tree", None)
    report("5 3 8 1 4 7 9 0 10 11",
           insert_all(None, [5, 3, 8, 1, 4, 7, 9, 0, 10, 11]))


if __name__ == "__main__":
    main()
```

Keduanya mengembalikan `true`/`True` dari fungsi bantu dan mengubahnya menjadi 1 atau 0 di `isBalanced`, supaya baris-baris yang tercetak sama dengan program C++, yang fungsi bantunya mengembalikan `int`.

12.9.6 Penyelesaian keliru yang menggoda

Kesalahan yang paling mudah terjadi justru di baris pertama. Fungsi bantu mengembalikan `int` yang dipakai sebagai nilai kebenaran, dan “tidak ada apa-apa” terasa seperti 0:

```
    if (root == nullptr) return 0;       // WRONG: "empty tree: not balanced"
```

Selebihnya tidak berubah, dan `wrong/p1206_wrong.cpp` mencetak:

```
sample tree            isBalanced = 0, one-pass = 0
6 3 9 1 5 8 10         isBalanced = 0, one-pass = 1
chain 1 2 3            isBalanced = 0, one-pass = 0
empty tree             isBalanced = 0, one-pass = 1
5 3 8 1 4 7 9 0 10 11  isBalanced = 0, one-pass = 0
```

Setiap pohon kini "tidak seimbang", bahkan pohon yang sempurna. Sebabnya, setiap pohon berakhir pada subpohon kosong: di sebuah daun, fungsi bantu memeriksa kedua tinggi daun itu (tidak masalah) lalu meminta `isBalancedHelper(leaf->left)`, yang mengembalikan 0, dan `&&` membawa 0 itu sampai ke akar. Sifat terburuk *bug* ini adalah bahwa ia lolos dari contoh soal itu sendiri: pohon contoh memang tidak seimbang, sehingga jawaban 0 tampak benar. Hanya kasus-kasus tepi pada Langkah 6, pohon kosong dan pohon yang *memang* seimbang, yang membongkarnya. Aturan di balik *base case* yang benar sama dengan aturan untuk jumlah: *base case* adalah nilai yang tidak mengubah apa pun di bawah operasi penggabungan, dan untuk `&&` nilai itu adalah *benar*.

12.9.7 Uji diri

> **UJI DIRI.** (a) Apakah pohon `2 1 3`, yaitu 2 dengan anak 1 dan 3, seimbang? Apakah pohon `4 2 6 1 3 5 7 0` seimbang? (b) Berapa banyak nilai paling sedikit yang harus disisipkan, dalam urutan tertentu, untuk mendapatkan pohon bertinggi 3 yang tetap seimbang tingginya?
>
> *Jawaban.* (a) `2 1 3` lengkap, jadi ya. Pada `4 2 6 1 3 5 7 0`, simpul 1 bertinggi 0 dan -1, simpul 2 bertinggi 1 dan 0, akarnya 2 dan 1: setiap selisih paling banyak 1, jadi ya. (b) Tujuh. Pohon seimbang bertinggi 3 memerlukan subpohon bertinggi 2 dan satu lagi bertinggi paling sedikit 1; pohon terkecil semacam itu memiliki 4 dan 2 simpul, jadi yang terkecil memiliki $1 + 4 + 2 = 7$ simpul, misalnya dengan menyisipkan 5, 3, 7, 2, 4, 6, 1. (Pohon seimbang terkecil untuk setiap tinggi adalah *pohon Fibonacci*, dan ukurannya, 1, 2, 4, 7, 12, 20, ..., tumbuh seperti bilangan Fibonacci pada Subbab 10.5 [8], [11].)

12.10 Soal 12.7 — Menghapus Sebuah Nilai

12.10.1 Soalnya

> **SOAL.** Write a recursive function to delete a particular data. `Node *delNode(Node *root, int delData)` You may use the helper function `Node *minValueNode(Node *node)`, where you find the minimum value of the node from the tree.
>
> *Dengan kata lain:* tulislah fungsi rekursif yang menghapus sebuah data tertentu dari pohon; Anda boleh memakai fungsi bantu `minValueNode` yang mencari simpul bernilai terkecil.

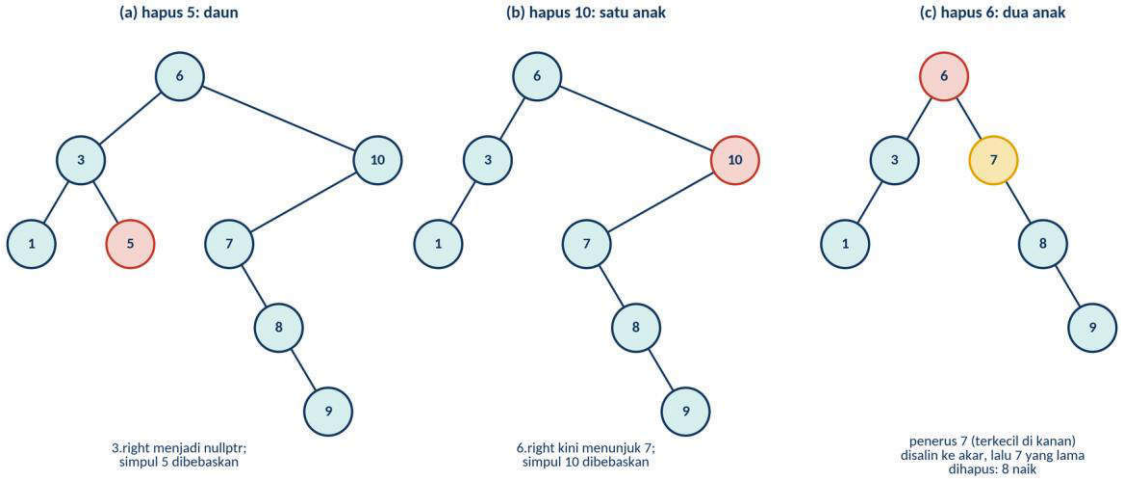
12.10.2 Dengan kata-kata kita sendiri

Buang simpul yang memuat `delData` dari *binary search tree*, bebaskan memorinya, dan sisakan pohon yang tetap *binary search tree* dan tetap memuat setiap nilai lainnya. Jika `delData` tidak ada di pohon, jangan ubah apa pun. Fungsinya mengembalikan `Node *`: akar pohon sesudah penghapusan. Nilai kembalian itulah kunci seluruh soal, karena akar itu sendiri mungkin simpul yang lenyap.

Menemukan simpulnya adalah `search` lagi: lebih kecil ke kiri, lebih besar ke kanan. Pekerjaannya ada pada membuangnya, dan ada tiga kasus, tergantung pada banyaknya anak simpul itu. **Tabel 12.4** mendaftarnya dan **Gambar 12.8** menggambar satu dari masing-masing.

Tabel 12.4 Tiga kasus penghapusan simpul dari *binary search tree*. Kasus 3 direduksi menjadi kasus 1 atau kasus 2, karena penerus tidak pernah memiliki anak kiri.

Kasus	Simpul yang dihapus memiliki	Yang menggantikan tempatnya	Mengapa pohon tetap BST
1	tanpa anak (daun)	tidak ada: pointer induknya menjadi `nullptr`	tidak ada simpul lain yang berpindah
2	tepat satu anak	anak itu, beserta seluruh subpohonnya	subpohon itu sudah berada di sisi yang benar dari setiap leluhur
3	dua anak	*penerus*-nya (*successor*), nilai terkecil di subpohon kanannya, disalin ke atas; lalu penerus itu dihapus dari subpohon kanan	penerus lebih besar daripada semua yang di kiri dan lebih kecil daripada semua yang lain di kanan



Gambar 12.8 Tiga kasus Soal 12.7 pada pohon kita: (a) daun 5 cukup dibuang; (b) 10 memiliki satu anak, dan 7 menggantikan tempatnya; (c) akar 6 memiliki dua anak, sehingga penerusnya, 7, disalin ke akar dan 7 yang lama dihapus di bawah.

Kasus 3 memerlukan penjelasan. Kita tidak dapat sekadar membuang simpul yang beranak dua, karena induknya hanya punya tempat untuk satu pointer di tempat simpul itu, sedangkan ada dua subpohon yang harus digantungkan. Maka kita biarkan simpul itu di tempatnya dan mengubah *nilai*-nya. Nilai mana yang dapat berdiri di tempat itu tanpa melanggar Persamaan (12.3)? Nilai itu harus lebih besar daripada semua yang ada di subpohon kiri dan lebih kecil daripada semua yang ada di subpohon kanan. Nilai terkecil subpohon kanan memiliki tepat sifat itu: ia lebih besar daripada nilai lama simpul itu (ia berada di subpohon kanan), sehingga lebih besar daripada semua yang di kiri, dan ia lebih kecil daripada setiap nilai lain di kanan. Nilai itu disebut *penerus*, karena ia nilai berikutnya dalam urutan terurut, dan ia tinggal di simpul paling kiri subpohon kanan, yang ditemukan oleh `minValueNode`. Simpul paling kiri tidak memiliki anak kiri (jika punya, ia bukan yang paling kiri), sehingga menghapusnya dari subpohon kanan sesudahnya adalah kasus 1 atau kasus 2. Skema ini berasal dari Hibbard [12], sama dengan skema Subbab 11.7.10, dan merupakan skema baku di buku-buku teks [1], [5].

12.10.3 Dengan tangan, selangkah demi selangkah

Kita menghapus 5, lalu 10, lalu 6, lalu 42, dari pohon yang sama, satu demi satu.

1. **Hapus 5.** $5 < 6$: ke kiri, dan apa pun yang dikembalikan pemanggilan kiri menjadi `6->left` yang baru. $5 > 3$: ke kanan; apa pun yang kembali menjadi `3->right` yang baru. Di 5: ditemukan, dan ia tidak memiliki anak kiri, sehingga pemanggilan itu menyimpan anak kanannya (`nullptr`), membebaskan simpul 5, dan mengembalikan `nullptr`. Kembali di 3: `3-`

`>right = nullptr`. Kembali di 6: `6->left = 3`, tidak berubah. *Preorder*-nya kini `6 3 1 10 7 8 9`.

2. **Hapus 10.** 10 > 6: ke kanan. Di 10: ditemukan; anak kirinya 7 tetapi anak kanannya `nullptr`, sehingga pemanggilan itu menyimpan 7, membebaskan 10, dan mengembalikan 7. Kembali di 6: `6->right = 7`. Subpohon 7–8–9 naik satu tingkat sebagai satu blok, dan tetap di sisi kanan 6, tempat ia seharusnya. *Preorder*: `6 3 1 7 8 9`.
3. **Hapus 6**, akarnya, yang memiliki dua anak, 3 dan 7. `minValueNode(6->right)` berangkat dari 7 dan ke kiri sejauh mungkin: 7 tidak memiliki anak kiri, sehingga penerusnya 7. Salin 7 ke akar. Kini ada dua 7, jadi hapuslah 7 dari subpohon *kanan*: di 7 yang lama, ditemukan, tanpa anak kiri, sehingga ia digantikan anak kanannya, 8. Kanan akar menjadi 8. *Preorder*: `7 3 1 8 9`. Nilai 6 lenyap, dan simpul yang tadinya memuatnya tetap menjadi akar.
4. **Hapus 42.** 42 > 7, 42 > 8, 42 > 9, dan kanan 9 adalah `nullptr`: *base case* mengembalikan `nullptr`, yang dipasang kembali oleh 9 sebagai anak kanannya, sebagaimana sebelumnya. Tidak ada yang berubah.

12.10.4 Penyelesaian C++

Kode 12.25 adalah program C++ untuk Soal 12.7.

Kode 12.25 `p1207_delete.cpp`: Soal 12.7, penghapusan pada ketiga kasus, dengan setiap hasil dipasang kembali oleh induknya.

```
// p1207_delete.cpp - Problem 12.7: delete a value from a binary search tree.
#include "tree_kit.hpp"

// The node with the smallest value in a non-empty subtree: its leftmost node.
Node *minValueNode(Node *node) {
    if (node->left == nullptr) return node;
    return minValueNode(node->left);
}

// Delete delData from the subtree and return the subtree's (possibly new) root.
Node *delNode(Node *root, int delData) {
    if (root == nullptr) return nullptr;              // not found: nothing changes
    if (delData < root->data) {
        root->left = delNode(root->left, delData);    // re-attach the left side
    } else if (delData > root->data) {
        root->right = delNode(root->right, delData);  // re-attach the right side
    } else if (root->left == nullptr) {               // no left child:
        Node *child = root->right;                    //   the right child (maybe
        delete root;                                  //   nullptr) takes its place
        return child;
    } else if (root->right == nullptr) {              // no right child:
        Node *child = root->left;                     //   the left child takes
        delete root;                                  //   its place
        return child;
    } else {                                          // two children:
        Node *succ = minValueNode(root->right);       //   the next larger value
        root->data = succ->data;                      //   is copied up here,
        root->right = delNode(root->right, succ->data);  // then deleted below
    }
    return root;
}

void show(const char *what, Node *root) {
    printf("%-10s preorder: ", what);
    preOrder(root);
    printf("| inorder: ");
    inOrder(root);
    printf("\n");
}
```

Kode 12.25 (Lanjutan) *p1207_delete.cpp: Soal 12.7, penghapusan pada ketiga kasus, dengan setiap hasil dipasang kembali oleh induknya.*

```
int main() {
    Node *root = nullptr;
    buildSample(&root);
    show("start", root);
    root = delNode(root, 5);          // a leaf
    show("delete 5", root);
    root = delNode(root, 10);         // one child (7)
    show("delete 10", root);
    root = delNode(root, 6);          // the root, two children
    show("delete 6", root);
    root = delNode(root, 42);         // not in the tree
    show("delete 42", root);
    printf("The tree now, turned on its side:\n");
    showSideways(root);
    for (int v : {7, 3, 1, 8, 9}) root = delNode(root, v);
    printf("after deleting the rest: %s\n",
           root == nullptr ? "empty" : "not empty");
    return 0;
}
```

```
start      preorder: 6 3 1 5 10 7 8 9 | inorder: 1 3 5 6 7 8 9 10
delete 5   preorder: 6 3 1 10 7 8 9 | inorder: 1 3 6 7 8 9 10
delete 10  preorder: 6 3 1 7 8 9 | inorder: 1 3 6 7 8 9
delete 6   preorder: 7 3 1 8 9 | inorder: 1 3 7 8 9
delete 42  preorder: 7 3 1 8 9 | inorder: 1 3 7 8 9
The tree now, turned on its side:
        9
    8
7
    3
        1
after deleting the rest: empty
```

Baris demi baris. Setiap baris keluaran sesuai dengan perhitungan tangan, dan setiap baris *inorder* tetap menaik: pohonnya tetap pohon pencarian sesudah setiap penghapusan. Gambar menyamping memperlihatkan pohon lima simpul yang dihasilkan, dengan 7 di akar. Baris terakhir menghapus kelima nilai yang tersisa, termasuk akarnya beberapa kali, dan berakhir dengan pohon kosong; di bawah Valgrind program ini membebaskan kedelapan simpulnya.

Perhatikan bentuk `delNode`. Fungsi itu tidak pernah melihat induknya dan tidak pernah mengubahnya. Ia hanya *mengembalikan* apa yang seharusnya berdiri di tempatnya, dan setiap pemanggilnya menulis `root->left = delNode(root->left, delData)` atau `root->right = ...`, dan `main` menulis `root = delNode(root, v)`. Inilah Langkah 5 metode kita, dan inilah yang membuat ketiga kasusnya begitu pendek: kasus 1 dan kasus 2 adalah tiga baris yang sama, “simpan satu-satunya anak, bebaskan simpulnya, kembalikan anak itu”, karena pada kasus 1 “satu-satunya anak” itu adalah `nullptr`.

Biaya. Fungsi ini menempuh satu lintasan dari akar turun ke simpul itu, dan pada kasus 3 berlanjut turun ke penerusnya, sehingga berbiaya $O(h)$, dengan h tingginya: antara $O(\log n)$ untuk pohon yang rendah dan $O(n)$ untuk rantai, menurut Persamaan (12.4).

12.10.5 Penyelesaian Java dan Python

Kode 12.26 dan **Kode 12.27** adalah versi Java dan Python-nya; keduanya mencetak baris-baris yang persis sama.

Kode 12.26 `P1207Delete.java`: Soal 12.7 dalam Java.

```java
// P1207Delete.java - Problem 12.7: delete a value from a binary search tree.
public class P1207Delete extends TreeKit {

    // The node with the smallest value in a non-empty subtree: its leftmost node.
    static Node minValueNode(Node node) {
        if (node.left == null) return node;
        return minValueNode(node.left);
    }

    // Delete delData from the subtree; return the subtree's (maybe new) root.
    static Node delNode(Node root, int delData) {
        if (root == null) return null;              // not found: nothing changes
        if (delData < root.data) {
            root.left = delNode(root.left, delData);     // re-attach the left side
        } else if (delData > root.data) {
            root.right = delNode(root.right, delData);  // re-attach the right side
        } else if (root.left == null) {             // no left child:
            return root.right;                      //   the right child replaces it
        } else if (root.right == null) {            // no right child:
            return root.left;                       //   the left child replaces it
        } else {                                    // two children:
            Node succ = minValueNode(root.right);   //   the next larger value
            root.data = succ.data;                  //   is copied up here,
            root.right = delNode(root.right, succ.data);   // then deleted below
        }
        return root;
    }

    static void show(String what, Node root) {
        System.out.printf("%-10s preorder: ", what);
        preOrder(root);
        System.out.print("| inorder: ");
        inOrder(root);
        System.out.println();
    }
```

Kode 12.26 (Lanjutan) *P1207Delete.java: Soal 12.7 dalam Java.*

```java
    public static void main(String[] args) {
        Node root = buildSample();
        show("start", root);
        root = delNode(root, 5);          // a leaf
        show("delete 5", root);
        root = delNode(root, 10);         // one child (7)
        show("delete 10", root);
        root = delNode(root, 6);          // the root, two children
        show("delete 6", root);
        root = delNode(root, 42);         // not in the tree
        show("delete 42", root);
        System.out.println("The tree now, turned on its side:");
        showSideways(root);
        for (int v : new int[] {7, 3, 1, 8, 9}) root = delNode(root, v);
        System.out.println("after deleting the rest: "
                + (root == null ? "empty" : "not empty"));
    }
}
```

Kode 12.27 `p1207_delete.py`: Soal 12.7 dalam Python.

```
# p1207_delete.py - Problem 12.7: delete a value from a binary search tree.
from tree_kit import *


def min_value_node(node):
    """The node with the smallest value in a non-empty subtree: the leftmost."""
    if node.left is None:
        return node
    return min_value_node(node.left)


def del_node(root, del_data):
    """Delete del_data from the subtree; return the subtree's (maybe new) root."""
    if root is None:                                  # not found: nothing changes
        return None
    if del_data < root.data:
        root.left = del_node(root.left, del_data)     # re-attach the left side
    elif del_data > root.data:
        root.right = del_node(root.right, del_data)   # re-attach the right side
    elif root.left is None:                           # no left child:
        return root.right                             #   the right child replaces it
    elif root.right is None:                          # no right child:
        return root.left                              #   the left child replaces it
    else:                                             # two children:
        succ = min_value_node(root.right)             #   the next larger value
        root.data = succ.data                         #   is copied up here,
        root.right = del_node(root.right, succ.data)  # then deleted below
    return root


def show(what, root):
    print(f"{what:<10} preorder: ", end="")
    pre_order(root)
    print("| inorder: ", end="")
    in_order(root)
    print()
```

Kode 12.27 (Lanjutan) *p1207_delete.py: Soal 12.7 dalam Python.*

```
def main():
    root = build_sample()
    show("start", root)
    root = del_node(root, 5)         # a leaf
    show("delete 5", root)
    root = del_node(root, 10)        # one child (7)
    show("delete 10", root)
    root = del_node(root, 6)         # the root, two children
    show("delete 6", root)
    root = del_node(root, 42)        # not in the tree
    show("delete 42", root)
    print("The tree now, turned on its side:")
    show_sideways(root)
    for v in (7, 3, 1, 8, 9):
        root = del_node(root, v)
    print("after deleting the rest:", "empty" if root is None else "not empty")


if __name__ == "__main__":
    main()
```

Dalam Java dan Python tidak ada `delete`: simpul yang tidak lagi dirujuk apa pun diambil kembali oleh *runtime*, sehingga kasus 1 dan 2 menyusut menjadi "kembalikan satu-satunya anak". Selebihnya struktur fungsinya sama.

12.10.6 Penyelesaian keliru yang menggoda

Penghapusannya sendiri biasanya ditulis dengan benar. Yang hilang adalah pemasangan kembali dalam perjalanan pulang:

```
if (delData < root->data) {
    delNode(root->left, delData);         // WRONG: the result is thrown away
} else if (delData > root->data) {
    delNode(root->right, delData);        // WRONG: the result is thrown away
```

Kode ini terkompilasi tanpa peringatan. Di bawah *sanitizer*, `wrong/p1207_wrong.cpp` mencetak hal berikut (dipersingkat):

```
start      preorder: 6 3 1 5 10 7 8 9 | inorder: 1 3 5 6 7 8 9 10
delete 5   preorder: 6 3 1 ==27864==ERROR: AddressSanitizer: heap-use-after-free on address
0x503000000100 at pc 0x55d504641918 bp 0x7ffebfd9d330 sp 0x7ffebfd9d320
READ of size 4 at 0x503000000100 thread T0
    #0 0x55d504641917 in preOrder(Node*) ../tree_kit.hpp:38
freed by thread T0 here:
    #1 0x55d50464081f in delNode(Node*, int) p1207_wrong.cpp:19
previously allocated by thread T0 here:
    #1 0x55d50464149f in insert(Node**, int) ../tree_kit.hpp:19
SUMMARY: AddressSanitizer: heap-use-after-free ../tree_kit.hpp:38 in preOrder(Node*)
```

Pemanggilan pada simpul 5 membebaskan simpul itu dan mengembalikan `nullptr`, sebagaimana mestinya, tetapi simpul 3 mengabaikan nilai kembalian itu, sehingga `3->right` masih memuat alamat simpul yang sudah dibebaskan, sebuah *dangling pointer* (pointer menggantung). Penelusuran berikutnya mengikutinya dan membaca memori yang sudah dibebaskan. Tanpa *sanitizer*, program itu mungkin mencetak 5 yang basi, atau sampah, atau *crash*, atau tampak bekerja; ketidakpastian itulah arti "perilaku tak terdefinisi". Laporannya menyebut tiga tempat yang penting: tempat memori yang dibebaskan itu dibaca (`preOrder`), tempat ia dibebaskan (`delNode`, baris 19), dan tempat ia dialokasikan (`insert`). Perbaikannya satu penugasan di setiap cabang: pasang kembali apa yang dikembalikan anaknya.

Kesalahan kedua yang lebih senyap adalah membebaskan simpul dengan `free(root)` padahal simpul itu dibuat dengan `new`. C++ menuntut `delete` untuk memori dari `new` dan `free` untuk memori dari `malloc`; mencampurnya adalah perilaku tak terdefinisi, dan AddressSanitizer melaporkannya sebagai *alloc-dealloc-mismatch* [13], [2].

12.10.7 Uji diri

> **UJI DIRI.** Kasus 3 dapat juga memakai *pendahulu* (*predecessor*), nilai terbesar di subpohon kiri. Hapuslah 6 dari pohon semula dengan cara itu: nilai mana yang pindah ke akar, dan bagaimana *preorder*-nya sesudah itu? Mengapa pendahulu tidak pernah memiliki anak kanan?
>
> *Jawaban.* Nilai terbesar di kiri ditemukan dengan bergerak ke kanan dari 3 sejauh mungkin: 5. Salin 5 ke akar dan hapus 5 yang lama, sebuah daun. *Preorder*-nya `5 3 1 10 7 8 9`. Pendahulu adalah simpul paling kanan di subpohon kiri; jika ia memiliki anak kanan, anak itu lebih ke kanan dan lebih besar, sehingga ia bukan yang paling kanan.

12.11 Soal 12.8 — Mencari Induk

12.11.1 Soalnya

> **SOAL.** Write a recursive function to find the parent node of a given child data. `Node *findParent(Node *root, int *childData)`

> *Dengan kata lain:* tulislah fungsi rekursif yang mencari simpul induk dari sebuah data anak yang diberikan.

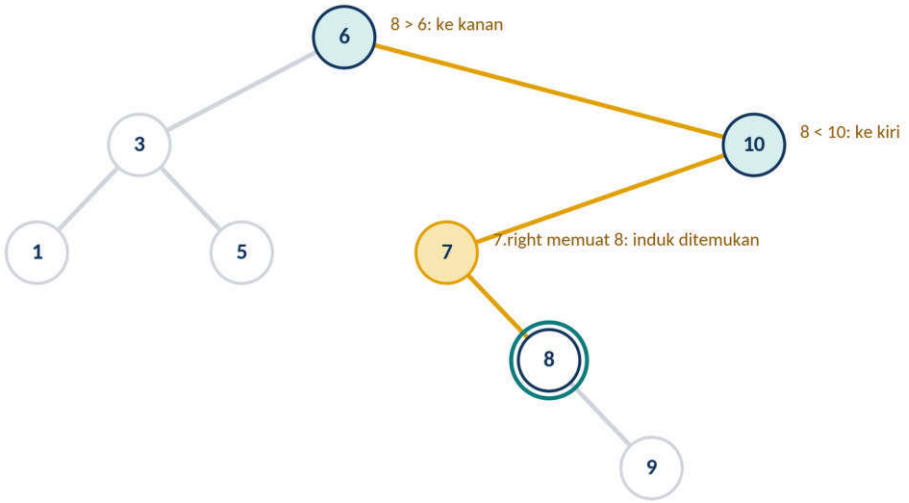
12.11.2 Dengan kata-kata kita sendiri

Diberikan sebuah nilai, kembalikan simpul yang anak kiri atau anak kanannya memuat nilai itu. Akar tidak memiliki induk, dan nilai yang tidak ada di pohon juga tidak; pada kedua kasus kita mengembalikan `nullptr`, dan pemanggil yang ingin membedakan keduanya dapat bertanya kepada `search` apakah nilai itu ada. Untuk pohon kita: induk 8 adalah 7, induk 10 adalah 6, induk 9 adalah 8, induk 1 adalah 3; 6 adalah akar; 4 tidak ada di pohon.

Soalnya mengirim nilai itu sebagai `int *childData`, pointer ke sebuah `int`. Bab 6 menjelaskan kapan fungsi memerlukan parameter pointer: ketika fungsi itu harus *mengubah* variabel milik pemanggil, atau ketika variabel itu terlalu besar untuk disalin. Tidak satu pun berlaku untuk satu `int` yang hanya dibaca fungsi, sehingga fungsi kita menerima `int childData` *by value*, yang juga memungkinkan pemanggil menulis `findParent(root, 8)` alih-alih mendeklarasikan variabel hanya untuk diambil alamatnya. Jika Anda harus mempertahankan tanda tangan yang diberikan, badan fungsinya sama dengan `*childData` di setiap tempat kita menulis `childData`, dan pemanggilannya menjadi `findParent(root, &x)`.

12.11.3 Dengan tangan, selangkah demi selangkah

Karena pohonnya *binary search tree*, simpul yang memuat nilai itu terletak pada satu-satunya lintasan yang akan ditempuh `search`, demikian pula induknya: induk itu tidak lain adalah *simpul terakhir sebelum* nilai itu pada lintasan tersebut. **Gambar 12.9** mengikuti lintasan untuk 8.



Gambar 12.9 `findParent(root, 8)` mengikuti lintasan pencarian 8. Di setiap simpul ia melihat satu langkah ke depan; di 7, simpul berikutnya pada lintasan memuat 8, sehingga 7 adalah induknya.

1. Di 6: $8 > 6$, jadi simpul berikutnya pada lintasan adalah anak kanan, 10. Simpul itu tidak memuat 8, jadi ajukan pertanyaan yang sama kepada 10.
2. Di 10: $8 < 10$, jadi simpul berikutnya adalah 7. Bukan 8; tanyakan kepada 7.
3. Di 7: $8 > 7$, jadi simpul berikutnya adalah anak kanan 7, yang memuat 8. **7 adalah induknya.**

Untuk 6, pengujian yang paling pertama sudah menemukan nilai itu di akar, dan jawabannya `nullptr`. Untuk 4, lintasannya 6, 3, 5, dan anak kiri 5 adalah `nullptr`: lintasan berakhir, 4 tidak ada, dan jawabannya `nullptr`.

12.11.4 Penyelesaian C++

Kode 12.28 memberikan dua versi. `findParent` memakai urutan pohon pencarian dan menempuh satu lintasan. `findParentAnyTree` bekerja pada pohon biner apa pun, terurut atau tidak, misalnya pada cermin Soal 12.5: ia memeriksa anak-anak simpul itu sendiri dan, jika tidak satu pun memuat nilainya, mencari induk di subpohon kiri lalu di subpohon kanan. Program ini menjalankan keduanya dan memeriksa bahwa keduanya sepakat.

Kode 12.28 `p1208_find_parent.cpp`: Soal 12.8, versi pohon pencarian dan versi pohon apa pun.

```
// p1208_find_parent.cpp - Problem 12.8: the parent of the node holding a value.
#include "tree_kit.hpp"

// BST version: walk the single path that childData would follow.
// Returns nullptr when childData is the root or is not in the tree.
Node *findParent(Node *root, int childData) {
    if (root == nullptr || root->data == childData) return nullptr;
    Node *next = (childData < root->data) ? root->left : root->right;
    if (next == nullptr) return nullptr;  // the path ends: not in the tree
    if (next->data == childData) return root;        // root is the parent
    return findParent(next, childData);              // ask the same question below
}

// Any binary tree, ordered or not: look on both sides.
Node *findParentAnyTree(Node *root, int childData) {
    if (root == nullptr) return nullptr;
    if ((root->left != nullptr && root->left->data == childData) ||
        (root->right != nullptr && root->right->data == childData)) {
        return root;
    }
    Node *found = findParentAnyTree(root->left, childData);
    if (found != nullptr) return found;
    return findParentAnyTree(root->right, childData);
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    for (int x : {8, 10, 9, 1, 6, 4}) {
        Node *p = findParent(root, x);
        Node *q = findParentAnyTree(root, x);
        printf("parent of %d: ", x);
        if (p != nullptr) {
            printf("%d", p->data);
        } else if (search(root, x) != nullptr) {
            printf("none (%d is the root)", x);
        } else {
            printf("none (%d is not in the tree)", x);
        }
        printf("  [both versions agree: %s]\n", p == q ? "yes" : "no");
    }
    clear(&root);
    return 0;
}
```

```
parent of 8: 7  [both versions agree: yes]
parent of 10: 6  [both versions agree: yes]
parent of 9: 8  [both versions agree: yes]
parent of 1: 3  [both versions agree: yes]
parent of 6: none (6 is the root)  [both versions agree: yes]
parent of 4: none (4 is not in the tree)  [both versions agree: yes]
```

Baris demi baris. Keenam jawabannya sesuai dengan perhitungan tangan, dan kedua versi sepakat pada setiap jawaban (dua `nullptr` juga dianggap sama). Versi BST memeriksa `next ==`

`nullptr` sebelum membaca `next->data`. **Biaya.** Versi BST menempuh satu lintasan, $O(h)$; versi pohon apa pun mungkin mencari di mana-mana, $\Theta(n)$ ketika nilainya tidak ada.

12.11.5 Penyelesaian Java dan Python

Kode 12.29 dan **Kode 12.30** adalah versi Java dan Python-nya; keduanya mencetak baris-baris yang persis sama.

Kode 12.29 `P1208FindParent.java`: Soal 12.8 dalam Java.

```java
// P1208FindParent.java - Problem 12.8: the parent of the node holding a value.
public class P1208FindParent extends TreeKit {

    // BST version: walk the single path that childData would follow.
    // Returns null when childData is the root or is not in the tree.
    static Node findParent(Node root, int childData) {
        if (root == null || root.data == childData) return null;
        Node next = (childData < root.data) ? root.left : root.right;
        if (next == null) return null;  // the path ends: not in the tree
        if (next.data == childData) return root;      // root is the parent
        return findParent(next, childData);          // ask the same question below
    }

    // Any binary tree, ordered or not: look on both sides.
    static Node findParentAnyTree(Node root, int childData) {
        if (root == null) return null;
        if ((root.left != null && root.left.data == childData)
                || (root.right != null && root.right.data == childData)) {
            return root;
        }
        Node found = findParentAnyTree(root.left, childData);
        if (found != null) return found;
        return findParentAnyTree(root.right, childData);
    }

    public static void main(String[] args) {
        Node root = buildSample();
        for (int x : new int[] {8, 10, 9, 1, 6, 4}) {
            Node p = findParent(root, x);
            Node q = findParentAnyTree(root, x);
            System.out.print("parent of " + x + ": ");
            if (p != null) {
                System.out.print(p.data);
            } else if (search(root, x) != null) {
                System.out.print("none (" + x + " is the root)");
            } else {
                System.out.print("none (" + x + " is not in the tree)");
            }
            System.out.println("  [both versions agree: "
                    + (p == q ? "yes" : "no") + "]");
        }
    }
}
```

Kode 12.30 `p1208_find_parent.py`: Soal 12.8 dalam Python.

```
# p1208_find_parent.py - Problem 12.8: find the parent of the node holding a value.
from tree_kit import *


def find_parent(root, child_data):
    """BST version: walk the single path that child_data would follow.
    Returns None when child_data is the root or is not in the tree."""
    if root is None or root.data == child_data:
        return None
    nxt = root.left if child_data < root.data else root.right
    if nxt is None:                              # the path ends: not in the tree
        return None
    if nxt.data == child_data:                   # root is the parent
        return root
    return find_parent(nxt, child_data)          # ask the same question below


def find_parent_any_tree(root, child_data):
    """Any binary tree, ordered or not: look on both sides."""
    if root is None:
        return None
    if (root.left is not None and root.left.data == child_data) or \
       (root.right is not None and root.right.data == child_data):
        return root
    found = find_parent_any_tree(root.left, child_data)
    if found is not None:
        return found
    return find_parent_any_tree(root.right, child_data)


def main():
    root = build_sample()
    for x in (8, 10, 9, 1, 6, 4):
        p = find_parent(root, x)
        q = find_parent_any_tree(root, x)
        if p is not None:
            answer = str(p.data)
        elif search(root, x) is not None:
            answer = f"none ({x} is the root)"
        else:
            answer = f"none ({x} is not in the tree)"
        agree = "yes" if p is q else "no"
        print(f"parent of {x}: {answer}  [both versions agree: {agree}]")
```

Kode 12.30 (Lanjutan) *p1208_find_parent.py: Soal 12.8 dalam Python.*

```
if __name__ == "__main__":
    main()
```

Program Java membandingkan kedua hasil dengan `p == q`, dan program Python dengan `p is q`: keduanya menanyakan apakah kedua variabel merujuk ke simpul yang *sama*, dan itulah pertanyaan yang tepat di sini.

12.11.6 Penyelesaian keliru yang menggoda

Percobaan pertama yang wajar menanyakan “apakah salah satu anakku memuat nilai itu?” dalam satu baris:

```
if (root->left->data == childData || root->right->data == childData) {
    return root;                          // WRONG: a child may be nullptr
}
```

Pada pohon kita, `wrong/p1208_wrong.cpp` mati pada pertanyaan pertamanya:

```
p1208_wrong.cpp:8:55: runtime error: member access within null pointer of type 'struct
Node'
==27876==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x558d9ab50593
bp 0x7ffc1a5e0460 sp 0x7ffc1a5e0430 T0)
    #0 0x558d9ab50593 in findParent(Node*, int) p1208_wrong.cpp:8
    #1 0x558d9ab5070d in findParent(Node*, int) p1208_wrong.cpp:12
SUMMARY: AddressSanitizer: SEGV p1208_wrong.cpp:8 in findParent(Node*, int)
```

Ketika mencari 8, pemanggilan pada 6 selamat (kedua anaknya ada), berpindah ke 10, dan di sana membaca `root->right->data`, padahal 10 tidak memiliki anak kanan. Kolom 55 baris 8 tepat ekspresi itu. Setengah dari simpul-simpul pohon yang lazim kehilangan salah satu anaknya, sehingga fungsi yang mengikuti pointer anak tanpa mengujinya akan *crash* pada hampir setiap masukan. Ujilah pointernya sebelum mengikutinya, seperti yang dilakukan kedua versi yang benar.

12.11.7 Uji diri

> **UJI DIRI.** Tulislah `Node *findGrandparent(Node *root, int x)` dengan memakai `findParent` dua kali. Apa yang dikembalikannya untuk 9, untuk 5, dan untuk 3?
>
> *Jawaban.* `Node *p = findParent(root, x); return p == nullptr ? nullptr : findParent(root, p->data);`. Untuk 9 induknya 8 dan induk 8 adalah 7. Untuk 5 induknya 3 dan induk 3 adalah 6. Untuk 3 induknya 6, akarnya, yang tidak memiliki induk, sehingga hasilnya `nullptr`.

12.12 Soal 12.9 — Melaporkan Anak-Anak

12.12.1 Soalnya

> **SOAL.** Write a recursive function to find the children node of a given parent data. `Node *findChild(Node *root, int parentData)` where you use the function `Node *search(Node *root, int data)` that is already defined.
>
> *Dengan kata lain:* tulislah fungsi rekursif yang mencari simpul-simpul anak dari sebuah data induk, dengan memanfaatkan `search` yang sudah ada.

12.12.2 Dengan kata-kata kita sendiri

Temukan simpul yang memuat `parentData` dan laporkan anak kiri dan anak kanannya, dengan menyatakan secara jelas kapan sebuah anak tidak ada dan kapan nilainya sama sekali tidak ada di pohon. Kembalikan simpul yang ditemukan, atau `nullptr`. Rekursinya ada di dalam `search`, yang disuruh dipakai oleh soal; `findChild` sendiri memanggilnya sekali lalu hanya melihat dua pointer.

Dengan tangan, dari **Gambar 12.1**: anak-anak 3 adalah 1 (kiri) dan 5 (kanan); 7 tidak memiliki anak kiri dan memiliki 8 di kanan; 9 adalah daun; 4 tidak ada di pohon.

12.12.3 Penyelesaian C++

Kode 12.31 adalah program C++ untuk Soal 12.9.

Kode 12.31 `p1209_find_child.cpp`: Soal 12.9, satu pemanggilan `search` dan dua pengamatan yang cermat.

```cpp
// p1209_find_child.cpp - Problem 12.9: the children of the node holding a value.
#include "tree_kit.hpp"

// Print the children of parentData and return its node (nullptr if absent).
Node *findChild(Node *root, int parentData) {
    Node *parent = search(root, parentData);   // the recursive walk
    if (parent == nullptr) {
        printf("%d is not in the tree\n", parentData);
        return nullptr;
    }
    if (parent->left != nullptr) {
        printf("left child of %d: %d\n", parentData, parent->left->data);
    } else {
        printf("%d has no left child\n", parentData);
    }
    if (parent->right != nullptr) {
        printf("right child of %d: %d\n", parentData, parent->right->data);
    } else {
        printf("%d has no right child\n", parentData);
    }
    return parent;
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    for (int x : {3, 7, 9, 4}) {
        Node *n = findChild(root, x);
        printf("  -> returned %s\n", n != nullptr ? "the node" : "no node");
    }
    clear(&root);
    return 0;
}
```

```
left child of 3: 1
right child of 3: 5
  -> returned the node
7 has no left child
right child of 7: 8
  -> returned the node
9 has no left child
9 has no right child
  -> returned the node
4 is not in the tree
  -> returned no node
```

Baris demi baris. Setiap nilai menghasilkan dua baris tentang anak-anaknya, atau satu baris yang menyatakan bahwa nilai itu tidak ada, dan baris yang diindentasi memperlihatkan apa yang dikembalikan fungsinya. Ketiga pengujian `parent == nullptr`, `parent->left != nullptr`, dan `parent->right != nullptr` adalah seluruh kesulitannya: masing-masing menjaga sebuah pointer sebelum diikuti. **Biaya.** Satu `search`, $O(h)$, ditambah kerja tetap.

12.12.4 Penyelesaian Java dan Python

Kode 12.32 dan **Kode 12.33** adalah versi Java dan Python-nya; keduanya mencetak baris-baris yang persis sama.

Kode 12.32 `P1209FindChild.java`: Soal 12.9 dalam Java.

```java
// P1209FindChild.java - Problem 12.9: the children of the node holding a value.
public class P1209FindChild extends TreeKit {

    // Print the children of parentData and return its node (null if absent).
    static Node findChild(Node root, int parentData) {
        Node parent = search(root, parentData);   // the recursive walk
        if (parent == null) {
            System.out.println(parentData + " is not in the tree");
            return null;
        }
        if (parent.left != null) {
            System.out.println("left child of " + parentData + ": "
                    + parent.left.data);
        } else {
            System.out.println(parentData + " has no left child");
        }
        if (parent.right != null) {
            System.out.println("right child of " + parentData + ": "
                    + parent.right.data);
        } else {
            System.out.println(parentData + " has no right child");
        }
        return parent;
    }

    public static void main(String[] args) {
        Node root = buildSample();
        for (int x : new int[] {3, 7, 9, 4}) {
            Node n = findChild(root, x);
            System.out.println("  -> returned "
                    + (n != null ? "the node" : "no node"));
        }
    }
}
```

Kode 12.33 `p1209_find_child.py`: Soal 12.9 dalam Python.

```python
# p1209_find_child.py - Problem 12.9: the children of the node holding a value.
from tree_kit import *


def find_child(root, parent_data):
    """Print the children of parent_data and return its node (None if absent)."""
    parent = search(root, parent_data)       # the recursive walk
    if parent is None:
        print(f"{parent_data} is not in the tree")
        return None
    if parent.left is not None:
        print(f"left child of {parent_data}: {parent.left.data}")
    else:
        print(f"{parent_data} has no left child")
    if parent.right is not None:
        print(f"right child of {parent_data}: {parent.right.data}")
    else:
        print(f"{parent_data} has no right child")
    return parent


def main():
    root = build_sample()
    for x in (3, 7, 9, 4):
        n = find_child(root, x)
        print("  -> returned", "the node" if n is not None else "no node")


if __name__ == "__main__":
    main()
```

12.12.5 Penyelesaian keliru yang menggoda

Versi yang menggoda memercayai bahwa `search` pasti berhasil dan langsung menuju anak-anaknya. Versi itu bekerja untuk 3, 7, dan 9, lalu `wrong/p1209_wrong.cpp` *crash* pada 4:

```
9 has no left child
9 has no right child
  -> returned the node
p1209_wrong.cpp:8:17: runtime error: member access within null pointer of type 'struct
Node'
==27888==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000008 (pc 0x55b2d5ad13e7
bp 0x7fffa4ac8290 sp 0x7fffa4ac8260 T0)
    #0 0x55b2d5ad13e7 in findChild(Node*, int) p1209_wrong.cpp:8
SUMMARY: AddressSanitizer: SEGV p1209_wrong.cpp:8 in findChild(Node*, int)
```

Alamat dalam laporan itu `0x8`, bukan 0. Program membaca medan `left` dari simpul yang tidak ada: `search` mengembalikan `nullptr`, alamat 0, dan medan `left` terletak 8 *byte* sesudah awal sebuah `Node` (sesudah `int` 4 *byte* dan 4 *byte* pengisi), sehingga program mencoba membaca alamat 8. Alamat beberapa *byte* di atas nol dalam laporan *crash* hampir selalu merupakan medan dari pointer *null*. Perbaikannya adalah menguji apa yang dikembalikan `search`.

12.12.6 Uji diri

> **UJI DIRI.** Tulislah `int degree(Node *root, int x)`, banyaknya anak simpul yang memuat `x` (0, 1, atau 2), yang mengembalikan -1 jika `x` tidak ada. Simpul mana saja di pohon kita yang berderajat 1?
>
> *Jawaban.* `Node *p = search(root, x); if (p == nullptr) return -1; return (p->left != nullptr) + (p->right != nullptr);` (dalam C++ sebuah `true` dihitung sebagai 1). Simpul berderajat 1 adalah 10, 7, dan 8, tiga simpul pada lintasan turun ke 9 yang hanya memiliki satu anak; 6 dan 3 berderajat 2 dan daun-daun 1, 5, dan 9 berderajat 0. Periksa:

> pohon biner dengan n_2 simpul berderajat 2 memiliki tepat $n_2 + 1$ daun, dan di sini $n_2 = 2$ dan daunnya ada 3.

12.13 Soal 12.10 — Dari Pohon Menjadi *Double Linked List*, di Tempat

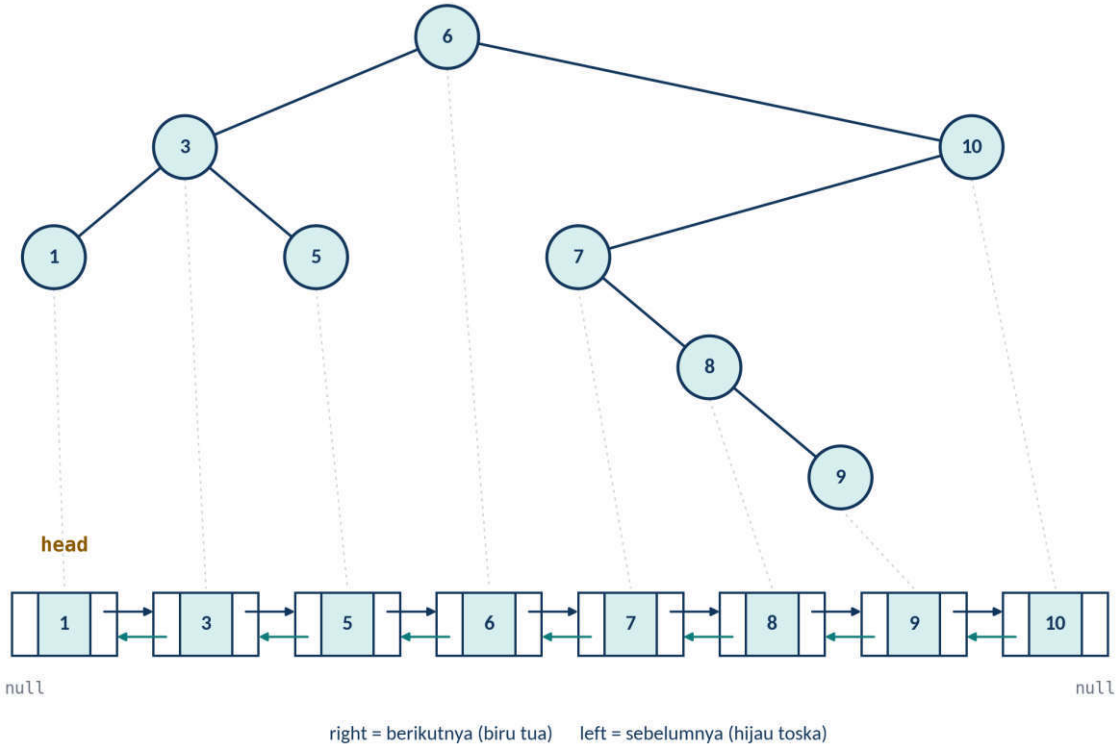
12.13.1 Soalnya

> **SOAL.** Write a recursive function to convert the binary tree to a doubly linked list in place. The left pointer of the tree should act as the previous pointer and the right pointer should act as the next pointer. `Node *convertToDLL(Node *root)` You may use the helper function `void convertToDLLHelper(Node *root, Node **head, Node **prev)`. You may also use the following function for the traversal of the list: `void preOrderDLL(Node *head) { Node *current = head; while (current != nullptr) { printf("%d ", current->data); current = current->right; } }`
>
> *Dengan kata lain:* tulislah fungsi rekursif yang mengubah pohon biner menjadi *double linked list* di tempat, dengan pointer kiri bertindak sebagai pointer "sebelumnya" dan pointer kanan sebagai pointer "berikutnya"; fungsi bantu dan fungsi penelusuran list di atas boleh dipakai.

12.13.2 Dengan kata-kata kita sendiri

Simpul pohon dan simpul *double linked list* (Subbab 9.2) memiliki bentuk yang sama: sebuah nilai dan dua pointer. Di pohon kedua pointer itu bernama `left` dan `right` dan berarti "subpohon yang lebih kecil" dan "subpohon yang lebih besar"; di list keduanya akan bernama `prev` dan `next`. Soal ini meminta kita *memakai ulang* simpul-simpul pohon, tanpa membuat satu pun yang baru (itulah arti *di tempat*), dan menyambung ulang kedua pointernya sehingga `left` berarti "simpul sebelumnya dalam list" dan `right` berarti "simpul berikutnya dalam list". Fungsinya mengembalikan simpul pertama list itu, yaitu *head*-nya.

Dalam urutan apa simpul-simpul itu seharusnya berdiri di dalam list? Soalnya tidak menyatakannya, tetapi ada satu jawaban wajar untuk *binary search tree*: urutan terurut, `1 3 5 6 7 8 9 10`, yaitu *inorder* pohon itu. **Gambar 12.10** memperlihatkan pohonnya dan list yang dihasilkannya. (Fungsi penelusuran yang diberikan bernama `preOrderDLL`, tetapi fungsi itu tidak melakukan *preorder* apa pun; ia menelusuri list yang sudah jadi dari *head* sepanjang pointer `right`, dan mencetak list dalam urutan list.)



Gambar 12.10 Soal 12.10: delapan simpul yang sama, disambung ulang. Di dalam list, `right` menunjuk ke nilai berikutnya yang lebih besar dan `left` ke nilai sebelumnya yang lebih kecil; garis pandu putus-putus memperlihatkan bahwa urutan list adalah *inorder* pohon itu.

12.13.3 Gagasannya, dengan tangan

Penelusuran *inorder* mengunjungi simpul-simpul tepat dalam urutan list. Maka kita menelusuri pohon secara *inorder* dan, di setiap simpul yang dikunjungi, *menambahkannya* ke ujung list yang sudah terbangun. Untuk menambahkan sebuah simpul kita perlu mengetahui simpul terakhir list sejauh ini; sebutlah `prev`. Menambahkan `root` sesudah `prev` berarti dua penugasan, aturan dua panah dari Bab 9:

- `prev->right = root` (panah maju: simpul terakhir yang lama kini menunjuk ke simpul yang baru), dan
- `root->left = prev` (panah mundur: simpul baru menunjuk balik ke simpul terakhir yang lama).

Lalu `root` menjadi simpul terakhir yang baru: `prev = root`. Simpul yang paling pertama dikunjungi tidak memiliki `prev` (nilainya `nullptr`); simpul itulah *head* list, sehingga kita mengingatnya di `head`.

Ada dua variabel, `head` dan `prev`, yang harus dilihat *dan diubah* oleh setiap pemanggilan fungsi bantu, dan perubahan yang dilakukan jauh di dalam rekursi harus tetap terlihat ketika pemanggilan-pemanggilan itu kembali. Karena itulah fungsi bantu menerima alamat keduanya, `Node **head` dan `Node **prev`: setiap pemanggilan membaca dan menulis dua variabel yang sama, yang hidup di dalam `convertToDLL`.

Inilah awal penelusurannya pada pohon kita, dengan `prev` sesudah setiap langkah:

1. *Inorder* turun menyusuri sisi kiri sampai ke 1. `prev` adalah `nullptr`, jadi `head = 1`; `1->left = nullptr`; `prev = 1`.
2. Kembali di 3: `1->right = 3`, `3->left = 1`, `prev = 3`.

3. Lalu subpohon kanan 3, yaitu 5: `3->right = 5`, `5->left = 3`, `prev = 5`.
4. Kembali di 6: `5->right = 6`, `6->left = 5`, `prev = 6`. Lalu subpohon kanan 6: 7, 8, 9, 10 ditambahkan dengan cara yang sama, dan yang terakhir, 10, mempertahankan `right = nullptr`, ujung list.

Amankah menimpa pointer yang masih kita telusuri? Inilah pertanyaan yang membuat soal ini sulit, dan layak dijawab dengan cermat. Kita menimpa `root->left` hanya sesudah pemanggilan pada subpohon kiri *selesai*, sehingga tidak ada lagi yang memerlukan pointer kiri yang lama. Dan `root->right` baru ditimpa belakangan, ketika simpul berikutnya ditambahkan dengan `prev->right = ...`; tetapi saat itu fungsi bantu sudah *membaca* `root->right` dan mengirimkannya ke pemanggilan pada subpohon kanan, karena argumen sebuah fungsi dievaluasi sebelum fungsi itu mulai. Pointer kanan yang lama dipakai tepat sekali, sebelum digantikan. Inilah aturan Bab 8 lagi, *simpan apa yang Anda perlukan sebelum mengubah sebuah tautan*, yang di sini dipatuhi oleh urutan ketiga langkah penelusuran *inorder*.

12.13.4 Penyelesaian C++

Kode 12.34 adalah program C++ untuk Soal 12.10.

Kode 12.34 `p1210_to_dll.cpp`: Soal 12.10, penelusuran *inorder* yang menambahkan setiap simpul ke list.

```c
// p1210_to_dll.cpp - Problem 12.10: turn a binary tree into a doubly linked list
// in place: left acts as "previous" and right acts as "next".
#include "tree_kit.hpp"

void convertToDLLHelper(Node *root, Node **head, Node **prev) {
    if (root == nullptr) return;
    convertToDLLHelper(root->left, head, prev);   // 1. every smaller value first
    if (*prev == nullptr) {
        *head = root;  // 2. nothing before: root is the head
    } else {
        (*prev)->right = root;                    //    link   prev -> root
    }
    root->left = *prev;                           //    and    prev <- root
    *prev = root;                                 //    root is now the last node
    convertToDLLHelper(root->right, head, prev);  // 3. every larger value after
}

Node *convertToDLL(Node *root) {
    Node *head = nullptr;
    Node *prev = nullptr;                         // the last node linked so far
    convertToDLLHelper(root, &head, &prev);
    return head;
}

// As given in the problem: follow the "next" links from the head.
void preOrderDLL(Node *head) {
    Node *current = head;
    while (current != nullptr) {
        printf("%d ", current->data);
        current = current->right;
    }
}

// Go to the last node, then follow the "prev" links back.
void printBackward(Node *head) {
    Node *tail = head;
    while (tail != nullptr && tail->right != nullptr) tail = tail->right;
    for (Node *cur = tail; cur != nullptr; cur = cur->left) {
        printf("%d ", cur->data);
    }
}
```

Kode 12.34 (Lanjutan) *p1210_to_dll.cpp: Soal 12.10, penelusuran inorder yang menambahkan setiap simpul ke list.*

```
void freeList(Node *head) {  // the tree is gone; free a list now
    while (head != nullptr) {
        Node *next = head->right;
        delete head;
        head = next;
    }
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    Node *head = convertToDLL(root);
    printf("forward : ");
    preOrderDLL(head);
    printf("\nbackward: ");
    printBackward(head);
    printf("\nhead %d, anything before the head? %s\n", head->data,
           head->left == nullptr ? "no" : "yes");
    freeList(head);

    Node *one = nullptr;
    insert(&one, 42);
    Node *h1 = convertToDLL(one);
    printf("one node: ");
    preOrderDLL(h1);
    printf("\nempty   : %s\n", convertToDLL(nullptr) == nullptr ? "no list" : "?");
    freeList(h1);
    return 0;
}
```

```
forward : 1 3 5 6 7 8 9 10
backward: 10 9 8 7 6 5 3 1
head 1, anything before the head? no
one node: 42
empty   : no list
```

Baris demi baris. Menelusuri maju sepanjang `right` memberikan nilai-nilai dalam urutan menaik; menelusuri mundur sepanjang `left` dari simpul terakhir memberikannya dalam urutan menurun, yang membuktikan bahwa *kedua* panah setiap simpul benar. *Head*-nya 1, dan tidak ada apa pun sebelumnya. Pohon satu simpul menjadi list satu simpul, dan pohon kosong tidak menjadi list apa pun. Begitu dikonversi, simpul-simpul itu membentuk list dan bukan lagi pohon, sehingga `clear` milik pohon tidak boleh dipakai padanya: fungsi itu akan mengikuti pointer `left` ke belakang dan membebaskan simpul dua kali. Program membebaskannya dengan `freeList`, penelusuran sepanjang `right`, dan Valgrind memastikan bahwa kesembilan simpul (delapan ditambah satu simpul 42) dibebaskan tepat sekali.

Biaya. Satu penelusuran *inorder*, waktu $\Theta(n)$, dan sama sekali tanpa simpul baru: satu-satunya memori tambahan adalah *call stack* $O(h)$.

12.13.5 Penyelesaian Java dan Python

Java dan Python tidak memiliki `Node **`. Fungsi bantu tetap harus mengubah dua variabel milik pemanggilnya, sehingga **Kode 12.35** dan **Kode 12.36** meletakkan keduanya di dalam satu objek kecil, `Ends`, dan mengirim objek itu ke setiap pemanggilan. Semua pemanggilan lalu memakai `head` dan `prev` yang sama, persis seperti pemanggilan-pemanggilan C++ memakai bersama dua variabel yang alamatnya mereka terima.

Kode 12.35 `P1210ToDll.java`: Soal 12.10 dalam Java, dengan `head` dan `prev` disimpan di satu objek bersama.

```
// P1210ToDll.java - Problem 12.10: turn a binary tree into a doubly linked list
// in place: left acts as "previous" and right acts as "next".
public class P1210ToDll extends TreeKit {

    // Java has no Node**: the two references to change live in one object.
    static class Ends {
        Node head;                          // the first node of the list
        Node prev;                          // the last node linked so far
    }

    static void convertToDLLHelper(Node root, Ends e) {
        if (root == null) return;
        convertToDLLHelper(root.left, e);     // 1. every smaller value first
        if (e.prev == null) {
            e.head = root;                   // 2. nothing before: root is the head
        } else {
            e.prev.right = root;              //    link   prev -> root
        }
        root.left = e.prev;                   //    and    prev <- root
        e.prev = root;                        //    root is now the last node
        convertToDLLHelper(root.right, e);    // 3. every larger value after
    }

    static Node convertToDLL(Node root) {
        Ends e = new Ends();                  // head and prev both start as null
        convertToDLLHelper(root, e);
        return e.head;
    }

    static void preOrderDLL(Node head) {      // as given: follow the "next" links
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.right;
        }
    }

    static void printBackward(Node head) {    // go to the end, then follow "prev"
        Node tail = head;
        while (tail != null && tail.right != null) tail = tail.right;
        for (Node cur = tail; cur != null; cur = cur.left) {
            System.out.print(cur.data + " ");
        }
    }
```

Kode 12.35 (Lanjutan) *P1210ToDll.java: Soal 12.10 dalam Java, dengan head dan prev disimpan di satu objek bersama.*

```
    public static void main(String[] args) {
        Node head = convertToDLL(buildSample());
        System.out.print("forward : ");
        preOrderDLL(head);
        System.out.print("\nbackward: ");
        printBackward(head);
        System.out.println("\nhead " + head.data + ", anything before the head? "
                + (head.left == null ? "no" : "yes"));
        Node h1 = convertToDLL(insert(null, 42));
        System.out.print("one node: ");
        preOrderDLL(h1);
        System.out.println("\nempty   : "
                + (convertToDLL(null) == null ? "no list" : "?"));
    }
}
```

Kode 12.36 `p1210_to_dll.py`: Soal 12.10 dalam Python.

```
# p1210_to_dll.py - Problem 12.10: turn a binary tree into a doubly linked list
# in place: left acts as "previous" and right acts as "next".
from tree_kit import *


class Ends:
    """The two references the helper must change: Python has no Node**."""
    def __init__(self):
        self.head = None                    # the first node of the list
        self.prev = None                    # the last node linked so far


def convert_to_dll_helper(root, e):
    if root is None:
        return
    convert_to_dll_helper(root.left, e)     # 1. every smaller value first
    if e.prev is None:
        e.head = root                       # 2. nothing before: root is the head
    else:
        e.prev.right = root                 #    link   prev -> root
    root.left = e.prev                      #    and    prev <- root
    e.prev = root                           #    root is now the last node
    convert_to_dll_helper(root.right, e)    # 3. every larger value after


def convert_to_dll(root):
    e = Ends()
    convert_to_dll_helper(root, e)
    return e.head


def pre_order_dll(head):                    # as given: follow the "next" links
    current = head
    while current is not None:
        print(current.data, end=" ")
        current = current.right
```

Kode 12.36 (Lanjutan) *p1210_to_dll.py: Soal 12.10 dalam Python.*

```
def print_backward(head):                    # go to the end, then follow "prev"
    tail = head
    while tail is not None and tail.right is not None:
        tail = tail.right
    cur = tail
    while cur is not None:
        print(cur.data, end=" ")
        cur = cur.left


def main():
    head = convert_to_dll(build_sample())
    print("forward : ", end="")
    pre_order_dll(head)
    print("\nbackward: ", end="")
    print_backward(head)
    before = "no" if head.left is None else "yes"
    print(f"\nhead {head.data}, anything before the head? {before}")
    h1 = convert_to_dll(insert(None, 42))
    print("one node: ", end="")
    pre_order_dll(h1)
    print("\nempty   :", "no list" if convert_to_dll(None) is None else "?")


if __name__ == "__main__":
    main()
```

12.13.6 Penyelesaian keliru yang menggoda

Fungsi bantu yang diberikan menerima `Node **prev`, dan pointer ganda itu tampak seperti kerepotan yang tidak perlu. Kirimkan `prev` *by value* saja, dan setiap pemanggilan mendapat salinannya sendiri:

```
void convertToDLLHelper(Node *root, Node **head, Node *prev) {  // WRONG: a copy
    ...
    prev = root;                                  // changes the copy only
    convertToDLLHelper(root->right, head, prev);
}
```

`wrong/p1210_wrong.cpp` mencetak ini lalu sebuah laporan kebocoran:

```
forward : 6 10
backward: 10 6
head 6, anything before the head? no
one node: 42
empty   : no list

==27900==ERROR: LeakSanitizer: detected memory leaks
SUMMARY: AddressSanitizer: 144 byte(s) leaked in 6 allocation(s).
```

Ketika pemanggilan pada 1 menyetel `prev`-nya menjadi 1, perubahan itu mati bersama pemanggilan itu; pemanggilan pada 3, yang memanggilnya, masih memiliki `prev == nullptr`, sehingga ia mengira *dirinya* simpul pertama, menyetel `head = 3` dan `3->left = nullptr`, memotong 1. Hal yang sama terjadi di 6, yang menjadi *head* dan memotong seluruh sisi kiri. Di sisi kanan, 7, 8, dan 9 tersambung satu sama lain, tetapi salinan `prev` milik 10 sendiri masih 6, sehingga 10 disambung langsung sesudah 6. List-nya `6 10`, dan enam simpul (1, 3, 5, 7, 8, 9, masing-masing 24 *byte*: 144 *byte*) tidak terjangkau. Variabel yang harus *dipakai bersama* oleh pemanggilan-pemanggilan rekursif harus dikirim melalui pointer (atau disimpan dalam objek bersama); sebuah salinan hanya membawa informasi turun, tidak pernah kembali naik.

12.13.7 Uji diri

> **UJI DIRI.** Jadikan list itu *sirkular* (Subbab 9.6): `right` simpul terakhir harus menunjuk ke *head*, dan `left` milik *head* ke simpul terakhir. Dua baris apa yang harus ditambahkan `convertToDLL` sesudah fungsi bantunya kembali? Bagaimana Anda lalu mencetak list itu sekali saja, tanpa berputar selamanya?
>
> *Jawaban.* Sesudah fungsi bantu, `prev` menunjuk ke simpul terakhir, jadi: `if (head != nullptr) { head->left = prev; prev->right = head; }`. Untuk mencetak, pakailah perulangan `do–while` yang berangkat dari `head` dan berhenti ketika kembali ke `head`: `Node *p = head; do { printf("%d ", p->data); p = p->right; } while (p != head);` (dijaga oleh `head != nullptr`). Pembebasan juga harus berhenti di *head*, misalnya dengan lebih dulu memutus lingkaran dengan `prev->right = nullptr`.

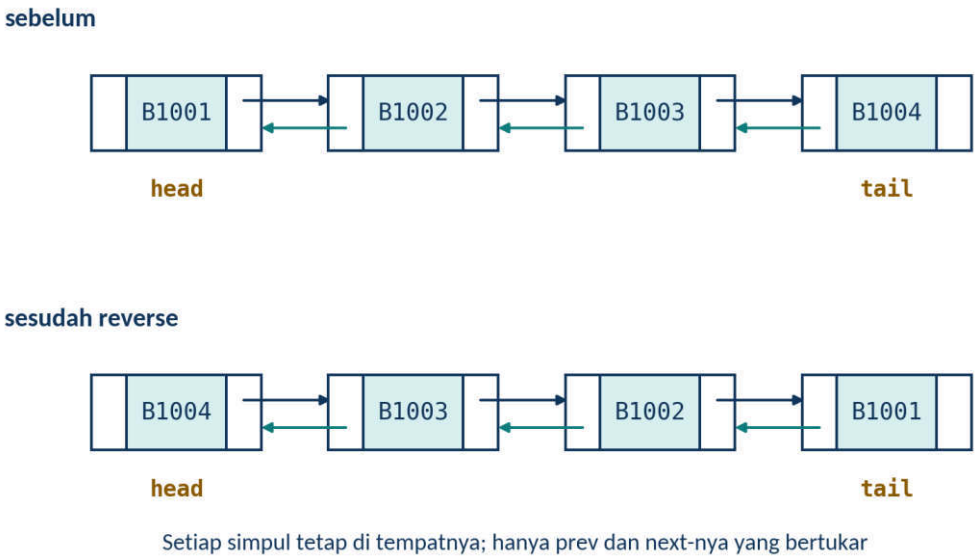
12.14 Soal Baru yang Dikerjakan: *Double Linked List* dan List Sirkular

Sepuluh soal sejauh ini menelusuri pohon. Dua soal berikutnya kembali ke list Bab 9, satu dengan dua tautan per simpul dan satu tanpa ujung, karena pertanyaan-pertanyaan yang sama muncul di sana dalam suasana yang lebih sederhana: pointer mana yang harus disimpan sebelum penugasan mana, dan kapan sebuah penelusuran berhenti.

12.14.1 Soal 12.11 — Membalik *double linked list* di tempat

> **SOAL.** Pustaka Ceria menyimpan riwayat baca setiap anggota sebagai *double linked list* berisi pengenal buku, yang terlama lebih dulu, dengan `head` dan `tail`. Baliklah list itu di tempat, sehingga buku terbaru berada paling depan, tanpa membuat simpul apa pun dan tanpa memindahkan pengenal buku dari satu simpul ke simpul lain.

Dengan kata-kata kita sendiri. Pada *single linked list*, pembalikan memerlukan tiga pointer dan urutan yang cermat (Soal 8.10). *Double linked list* membuatnya lebih sederhana, karena setiap simpul sudah mengenal kedua tetangganya. Membalik list berarti bahwa, untuk setiap simpul, "simpul sesudahku" dan "simpul sebelumku" bertukar peran. Jadi seluruh algoritmenya adalah: *di setiap simpul, tukarkan `prev` dan `next`; lalu tukarkan `head` dan `tail`.* **Gambar 12.11** memperlihatkan list sebelum dan sesudahnya.



Gambar 12.11 Soal 12.11: setiap simpul tetap di tempatnya dalam memori dan tetap memuat bukunya; hanya kedua panahnya yang bertukar peran, dan `head` serta `tail` bertukar.

Dengan tangan. Satu-satunya hal yang pelik adalah cara berpindah ke simpul berikutnya setelah panah-panahnya ditukar. Ambillah list `B1001 B1002 B1003 B1004`. Di `B1001`, simpan dulu `saved = next`, yaitu `B1002`; lalu tukarkan, sehingga `next` menjadi `nullptr` (`prev` yang lama) dan `prev` menjadi `B1002`. Sekarang pindahlah ke `saved`, `next` yang *lama*, yaitu `B1002`. Di sana, simpan `B1003`, tukarkan, pindah ke `B1003`; dan seterusnya. Di `B1004`, `saved` bernilai `nullptr` dan perulangan berakhir. Akhirnya *head* lama `B1001` menjadi *tail* dan *tail* lama `B1004` menjadi *head*.

Penyelesaiannya. Kode 12.37 adalah program C++-nya; versi Java dan Python, **Kode 12.38** dan **Kode 12.39**, mencetak baris-baris yang sama.

Kode 12.37 `p1211_reverse_dll.cpp`: Soal 12.11, menukar kedua panah setiap simpul.

```cpp
// p1211_reverse_dll.cpp - Problem 12.11: reverse a doubly linked list in place.
#include <cstdio>
#include <string>
using namespace std;

struct DNode {
    string book;
    DNode *prev;
    DNode *next;
};

struct DList {
    DNode *head = nullptr;
    DNode *tail = nullptr;
};

void pushBack(DList &list, const string &book) {
    DNode *fresh = new DNode{book, list.tail, nullptr};
    if (list.tail == nullptr) {
        list.head = fresh;                  // the list was empty
    } else {
        list.tail->next = fresh;
    }
    list.tail = fresh;
}

void reverse(DList &list) {
    DNode *cur = list.head;
    while (cur != nullptr) {
        DNode *saved = cur->next;           // 1. remember where to go next
        cur->next = cur->prev;              // 2. swap the two arrows
        cur->prev = saved;
        cur = saved;                        // 3. move on to the old "next"
    }
    DNode *oldHead = list.head;             // 4. the two ends swap roles
    list.head = list.tail;
    list.tail = oldHead;
}
```

Kode 12.37 (Lanjutan) *p1211_reverse_dll.cpp: Soal 12.11, menukar kedua panah setiap simpul.*

```
void printBoth(const char *what, const DList &list) {
    printf("%-9s forward: ", what);
    for (DNode *p = list.head; p != nullptr; p = p->next) {
        printf("%s ", p->book.c_str());
    }
    printf("| backward: ");
    for (DNode *p = list.tail; p != nullptr; p = p->prev) {
        printf("%s ", p->book.c_str());
    }
    printf("\n");
}

void clear(DList &list) {
    while (list.head != nullptr) {
        DNode *next = list.head->next;
        delete list.head;
        list.head = next;
    }
    list.tail = nullptr;
}

int main() {
    DList history, one, none;
    for (const char *b : {"B1001", "B1002", "B1003", "B1004"}) {
        pushBack(history, b);
    }
    printBoth("start", history);
    reverse(history);
    printBoth("reversed", history);
    reverse(history);
    printBoth("twice", history);
    pushBack(one, "B2001");
    reverse(one);
    printBoth("one node", one);
    reverse(none);
    printBoth("empty", none);
    clear(history);
    clear(one);
    return 0;
}
```

```
start     forward: B1001 B1002 B1003 B1004 | backward: B1004 B1003 B1002 B1001
reversed  forward: B1004 B1003 B1002 B1001 | backward: B1001 B1002 B1003 B1004
twice     forward: B1001 B1002 B1003 B1004 | backward: B1004 B1003 B1002 B1001
one node  forward: B2001 | backward: B2001
empty     forward: | backward:
```

Setiap baris mencetak list itu dua kali, sekali sepanjang `next` dari *head* dan sekali sepanjang `prev` dari *tail*, sehingga satu panah yang rusak pun akan tampak. Membalik dua kali mengembalikan list semula; list satu simpul dan list kosong ditangani oleh kode yang sama tanpa kasus khusus, karena perulangannya berjalan sekali saja atau tidak sama sekali, dan menukar `head` dengan `tail` tidak berbahaya ketika keduanya sama. **Biaya.** Waktu $\Theta(n)$, memori tambahan $O(1)$.

Kode 12.38 `P1211ReverseDll.java`: Soal 12.11 dalam Java.

```java
// P1211ReverseDll.java - Problem 12.11: reverse a doubly linked list in place.
public class P1211ReverseDll {

    static class DNode {
        String book;
        DNode prev;
        DNode next;

        DNode(String book, DNode prev, DNode next) {
            this.book = book;
            this.prev = prev;
            this.next = next;
        }
    }

    static class DList {
        DNode head;
        DNode tail;
    }

    static void pushBack(DList list, String book) {
        DNode fresh = new DNode(book, list.tail, null);
        if (list.tail == null) {
            list.head = fresh;                    // the list was empty
        } else {
            list.tail.next = fresh;
        }
        list.tail = fresh;
    }

    static void reverse(DList list) {
        DNode cur = list.head;
        while (cur != null) {
            DNode saved = cur.next;               // 1. remember where to go next
            cur.next = cur.prev;                  // 2. swap the two arrows
            cur.prev = saved;
            cur = saved;                          // 3. move on to the old "next"
        }
        DNode oldHead = list.head;                // 4. the two ends swap roles
        list.head = list.tail;
        list.tail = oldHead;
    }
```

Kode 12.38 (Lanjutan) *P1211ReverseDll.java: Soal 12.11 dalam Java.*

```
    static void printBoth(String what, DList list) {
        System.out.printf("%-9s forward: ", what);
        for (DNode p = list.head; p != null; p = p.next) {
            System.out.print(p.book + " ");
        }
        System.out.print("| backward: ");
        for (DNode p = list.tail; p != null; p = p.prev) {
            System.out.print(p.book + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        DList history = new DList(), one = new DList(), none = new DList();
        for (String b : new String[] {"B1001", "B1002", "B1003", "B1004"}) {
            pushBack(history, b);
        }
        printBoth("start", history);
        reverse(history);
        printBoth("reversed", history);
        reverse(history);
        printBoth("twice", history);
        pushBack(one, "B2001");
        reverse(one);
        printBoth("one node", one);
        reverse(none);
        printBoth("empty", none);
    }
}
```

Kode 12.39 `p1211_reverse_dll.py`: Soal 12.11 dalam Python.

```python
# p1211_reverse_dll.py - Problem 12.11: reverse a doubly linked list in place.


class DNode:
    def __init__(self, book, prev=None, nxt=None):
        self.book = book
        self.prev = prev
        self.next = nxt


class DList:
    def __init__(self):
        self.head = None
        self.tail = None


def push_back(lst, book):
    fresh = DNode(book, lst.tail, None)
    if lst.tail is None:
        lst.head = fresh                    # the list was empty
    else:
        lst.tail.next = fresh
    lst.tail = fresh


def reverse(lst):
    cur = lst.head
    while cur is not None:
        saved = cur.next                    # 1. remember where to go next
        cur.next = cur.prev                 # 2. swap the two arrows
        cur.prev = saved
        cur = saved                         # 3. move on to the old "next"
    lst.head, lst.tail = lst.tail, lst.head  # 4. the two ends swap roles


def print_both(what, lst):
    print(f"{what:<9} forward: ", end="")
    p = lst.head
    while p is not None:
        print(p.book, end=" ")
        p = p.next
    print("| backward: ", end="")
    p = lst.tail
    while p is not None:
        print(p.book, end=" ")
        p = p.prev
```

Kode 12.39 (Lanjutan) *p1211_reverse_dll.py: Soal 12.11 dalam Python.*

```
    print()


def main():
    history, one, none = DList(), DList(), DList()
    for b in ("B1001", "B1002", "B1003", "B1004"):
        push_back(history, b)
    print_both("start", history)
    reverse(history)
    print_both("reversed", history)
    reverse(history)
    print_both("twice", history)
    push_back(one, "B2001")
    reverse(one)
    print_both("one node", one)
    reverse(none)
    print_both("empty", none)


if __name__ == "__main__":
    main()
```

Penyelesaian keliru yang menggoda. Sesudah menukar panah-panahnya, terasa wajar untuk melanjutkan dengan `cur = cur->next`, seperti setiap penelusuran list dalam buku ini. Tetapi `next` baru saja ditukar: kini ia memuat `prev` yang *lama*. Pada simpul pertama nilainya `nullptr`, dan perulangan berhenti sesudah satu simpul. `wrong/p1211_wrong.cpp` mencetak (penelusuran mundurnya dibatasi sembilan langkah, supaya berhenti):

```
start     forward: B1001 B1002 B1003 B1004 | backward: B1004 B1003 B1002 B1001
reversed  forward: B1004 | backward: B1001 B1002 B1001 B1002 B1001 B1002 B1001 B1002 B1001
==27913==ERROR: LeakSanitizer: detected memory leaks
SUMMARY: AddressSanitizer: 144 byte(s) leaked in 3 allocation(s).
```

Hanya `B1001` yang diubah. *Head*-nya kini `B1004`, yang `next`-nya masih `nullptr`, sehingga penelusuran maju hanya mencetak satu buku. Penelusuran mundur berangkat dari `B1001`, yang `prev`-nya kini menunjuk ke `B1002`, yang `prev`-nya masih menunjuk ke `B1001`: kedua simpul saling menunjuk, dan penelusuran tanpa penjaga sembilan langkah tidak akan pernah berakhir. Terakhir, program membebaskan list dengan mengikuti `next` dari *head*, hanya mencapai `B1004`, dan membocorkan ketiga simpul lainnya. Aturannya adalah aturan Subbab 8.1: simpan pointer yang akan Anda perlukan (`saved`), dan pakailah salinan simpanan itu, bukan medan yang baru saja Anda timpa.

> **UJI DIRI.** Tulislah pembalikan itu secara *rekursif*: `DNode *reverseRec(DNode *cur)` menukar panah-panah `cur` dan setiap simpul sesudahnya lalu mengembalikan *head* yang baru. Apa *base case*-nya?
>
> *Jawaban.* `if (cur == nullptr) return nullptr; DNode *old = cur->next; cur->next = cur->prev; cur->prev = old; if (old == nullptr) return cur; return reverseRec(old);`. *Base case*-nya adalah list kosong dan simpul terakhir: simpul terakhir list lama, sesudah ditukar, menjadi *head* yang baru. (Pemanggil masih harus menyetel `tail` yang baru menjadi *head* yang lama.) Rekursi ini adalah *tail call*: tidak ada yang terjadi sesudah pemanggilan rekursif kembali, dan karena itulah versi perulangannya begitu mudah ditulis (Subbab 10.11).

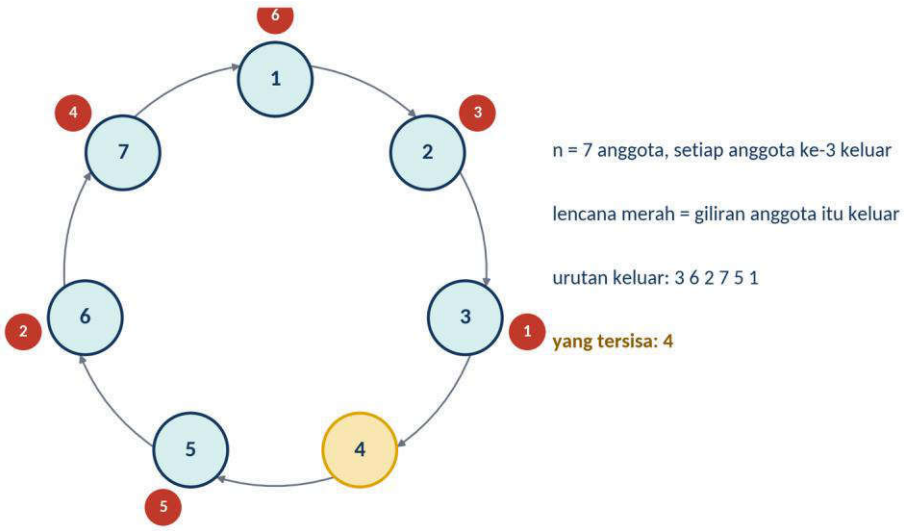
12.14.2 Soal 12.12 — Lingkaran Josephus

> **SOAL.** Klub baca Pustaka Ceria menerima satu eksemplar awal sebuah novel baru. Ke-n anggotanya, bernomor 1 sampai n, duduk melingkar. Mulai dari anggota 1, mereka berhitung mengelilingi lingkaran, dan setiap anggota ke-k yang terhitung keluar dari lingkaran; hitungan dilanjutkan dari anggota berikutnya. Anggota terakhir yang tersisa mendapat eksemplar itu. Dengan memakai *linked list* sirkular, cetaklah urutan keluarnya para anggota dan nomor anggota yang tersisa. Periksalah anggota yang tersisa itu dengan rumus rekursif.

Dengan kata-kata kita sendiri. Inilah *soal Josephus*, dinamai menurut sejarawan Flavius Josephus, sebuah soal klasik untuk list sirkular sekaligus rekursi [14]. List sirkular (Subbab 9.6 membangun jenis yang bertaut ganda; satu tautan per simpul sudah cukup di sini) adalah model yang wajar: `next` milik anggota terakhir menunjuk kembali ke anggota 1, sehingga berhitung "mengelilingi lingkaran" cukup berarti mengikuti `next` terus-menerus. Membuang seorang anggota adalah pemutusan tautan dari Bab 7: untuk membuang simpul `out`, kita memerlukan simpul *sebelum*-nya, `prev`, lalu menyetel `prev->next = out->next`.

Satu-satunya keputusan rancangan adalah di mana `prev` dimulai. Untuk membuang anggota ke-k yang dihitung mulai dari anggota 1, kita harus berdiri di anggota *sebelum*-nya, yaitu $k-1$ langkah sesudah anggota sebelum anggota 1, yakni simpul terakhir lingkaran. Jadi `buildCircle` mengembalikan simpul terakhir, dan kita mulai dari sana. Lalu setiap putaran memajukan `prev` sebanyak $k-1$ kali, membuang `prev->next`, dan hitungan berikutnya dimulai dari anggota sesudah yang dibuang, yang lagi-lagi adalah `prev->next`. Lingkaran tinggal satu anggota ketika `next` anggota itu adalah dirinya sendiri.

Dengan tangan, dengan $n = 7$ dan $k = 3$, seperti pada **Gambar 12.12**. `prev` dimulai di 7. Dua langkah (ke 1, ke 2) dan 3 keluar. Dari 2, dua langkah (4, 5) dan 6 keluar. Dari 5, dua langkah (7, 1) dan 2 keluar. Dari 1, dua langkah (4, 5) dan 7 keluar. Dari 5, dua langkah (1, 4) dan 5 keluar. Dari 4, dua langkah (1, 4) dan 1 keluar. Anggota 4 sendirian: dialah yang tersisa.



Gambar 12.12 Lingkaran Josephus untuk $n = 7$, $k = 3$: lencana merah di samping setiap anggota adalah putaran saat anggota itu keluar. Anggota 4 tidak pernah terhitung ketiga dan mendapatkan bukunya.

Rumusnya. Beri nomor anggota mulai dari 0, bukan 1, dan misalkan $J(n)$ adalah posisi anggota yang tersisa dalam lingkaran berisi n anggota ketika hitungan dimulai di posisi 0. Dengan satu anggota, yang tersisa berada di posisi 0. Dengan n anggota, anggota pertama yang keluar berada di posisi $k-1$ (modulo n), dan hitungan berlanjut dari posisi k dengan lingkaran berisi $n-1$ anggota. Lingkaran yang lebih kecil itu adalah soal yang sama, hanya dinomori ulang: posisi 0-nya

adalah posisi k kita, posisi 1-nya adalah posisi $k+1$ kita, dan seterusnya mengelilingi lingkaran. Jadi yang tersisa dari lingkaran kecil, pada posisi $J(n-1)$ *miliknya*, berada di posisi $J(n-1)+k$ *kita*, modulo n:

$$J(1) = 0, \qquad J(n) = (J(n-1) + k) \bmod n \quad (n \geq 2). \tag{12.8}$$

Untuk $n = 7$ dan $k = 3$: $J(1) = 0$, $J(2) = 3 \bmod 2 = 1$, $J(3) = 4 \bmod 3 = 1$, $J(4) = 4 \bmod 4 = 0$, $J(5) = 3 \bmod 5 = 3$, $J(6) = 6 \bmod 6 = 0$, $J(7) = 3 \bmod 7 = 3$. Posisi 3 dihitung dari 0 adalah anggota 4, sesuai dengan lingkaran tadi. Persamaan (12.8) mengubah simulasi $O(nk)$ menjadi aritmetika $O(n)$, dan merupakan contoh yang bagus untuk gagasan pokok Bab 10: selesaikan soal dengan menyelesaikan salinannya yang lebih kecil.

Penyelesaiannya. **Kode 12.40** menjalankan lingkaran dan rumusnya untuk lima kasus dan membandingkan keduanya.

Kode 12.40 `p1212_josephus.cpp`: Soal 12.12, list sirkular dan rumus rekursif, dibandingkan.

```
// p1212_josephus.cpp - Problem 12.12: who is left when every k-th member of a
// circle leaves? A circular list, and a recursive formula to check it.
#include <cstdio>

struct CNode {
    int member;
    CNode *next;
};

// Build the circle 1, 2, ..., n and return its LAST node (whose next is 1).
CNode *buildCircle(int n) {
    CNode *first = new CNode{1, nullptr};
    CNode *last = first;
    for (int m = 2; m <= n; m++) {
        last->next = new CNode{m, nullptr};
        last = last->next;
    }
    last->next = first;                          // close the circle
    return last;
}

int josephusList(int n, int k) {
    CNode *prev = buildCircle(n);  // the node BEFORE the next one counted
    printf("n=%d, k=%d, leaving in order:", n, k);
    while (prev->next != prev) {                 // more than one member remains
        for (int i = 1; i < k; i++) prev = prev->next;   // count k-1 members
        CNode *out = prev->next;                 // the k-th member leaves
        prev->next = out->next;                  // link around it
        printf(" %d", out->member);
        delete out;
    }
    int survivor = prev->member;
    delete prev;
    printf(" | survivor %d\n", survivor);
    return survivor;
}

// J(1) = 0 and J(n) = (J(n-1) + k) mod n, counting members from 0.
int josephusFormula(int n, int k) {
    if (n == 1) return 0;
    return (josephusFormula(n - 1, k) + k) % n;
}
```

Kode 12.40 (Lanjutan) *p1212_josephus.cpp: Soal 12.12, list sirkular dan rumus rekursif, dibandingkan.*

```cpp
int main() {
    const int cases[][2] = {{7, 3}, {5, 2}, {6, 1}, {1, 4}, {10, 3}};
    for (const auto &c : cases) {
        int a = josephusList(c[0], c[1]);
        int b = josephusFormula(c[0], c[1]) + 1;  // +1: members count from 1
        printf("  formula says %d: %s\n", b, a == b ? "agree" : "DISAGREE");
    }
    return 0;
}
```

```
n=7, k=3, leaving in order: 3 6 2 7 5 1 | survivor 4
  formula says 4: agree
n=5, k=2, leaving in order: 2 4 1 5 | survivor 3
  formula says 3: agree
n=6, k=1, leaving in order: 1 2 3 4 5 | survivor 6
  formula says 6: agree
n=1, k=4, leaving in order: | survivor 1
  formula says 1: agree
n=10, k=3, leaving in order: 3 6 9 2 7 1 8 5 10 | survivor 4
  formula says 4: agree
```

Kasus pertama adalah yang dihitung dengan tangan. $k = 1$ mengeluarkan para anggota secara berurutan dan menyisakan yang terakhir; $n = 1$ menyisakan satu-satunya anggota tanpa satu hitungan pun; dan pada setiap kasus list dan Persamaan (12.8) sepakat. Program C++ membebaskan setiap simpul yang dibuangnya dan, akhirnya, anggota yang tersisa, sehingga Valgrind tidak melaporkan kebocoran. **Biaya.** Simulasi list mengerjakan $k - 1$ langkah untuk masing-masing dari $n - 1$ pembuangan: $\Theta(nk)$. Rumusnya melakukan n pemanggilan: waktu $\Theta(n)$ dan stack $O(n)$.

Kode 12.41 dan **Kode 12.42** adalah versi Java dan Python-nya; keduanya mencetak baris-baris yang persis sama.

Kode 12.41 `P1212Josephus.java`: Soal 12.12 dalam Java.

```java
// P1212Josephus.java - Problem 12.12: who is left when every k-th member of a
// circle leaves? A circular list, and a recursive formula to check it.
public class P1212Josephus {

    static class CNode {
        int member;
        CNode next;

        CNode(int member) {
            this.member = member;
        }
    }

    // Build the circle 1, 2, ..., n and return its LAST node (whose next is 1).
    static CNode buildCircle(int n) {
        CNode first = new CNode(1);
        CNode last = first;
        for (int m = 2; m <= n; m++) {
            last.next = new CNode(m);
            last = last.next;
        }
        last.next = first;                          // close the circle
        return last;
    }

    static int josephusList(int n, int k) {
        CNode prev = buildCircle(n);  // the node BEFORE the next one counted
        System.out.printf("n=%d, k=%d, leaving in order:", n, k);
        while (prev.next != prev) {  // more than one member remains
            for (int i = 1; i < k; i++) prev = prev.next;   // count k-1 members
            CNode out = prev.next;                  // the k-th member leaves
            prev.next = out.next;                   // link around it
            System.out.print(" " + out.member);
        }
        int survivor = prev.member;
        System.out.println(" | survivor " + survivor);
        return survivor;
    }

    // J(1) = 0 and J(n) = (J(n-1) + k) mod n, counting members from 0.
    static int josephusFormula(int n, int k) {
        if (n == 1) return 0;
        return (josephusFormula(n - 1, k) + k) % n;
    }
```

Kode 12.41 (Lanjutan) *P1212Josephus.java: Soal 12.12 dalam Java.*

```java
    public static void main(String[] args) {
        int[][] cases = {{7, 3}, {5, 2}, {6, 1}, {1, 4}, {10, 3}};
        for (int[] c : cases) {
            int a = josephusList(c[0], c[1]);
            // +1: members are numbered from 1
            int b = josephusFormula(c[0], c[1]) + 1;
            System.out.printf("  formula says %d: %s%n", b,
                    a == b ? "agree" : "DISAGREE");
        }
    }
}
```

Kode 12.42 `p1212_josephus.py`: Soal 12.12 dalam Python.

```python
# p1212_josephus.py - Problem 12.12: who is left when every k-th member of a
# circle leaves? A circular list, and a recursive formula to check it.


class CNode:
    def __init__(self, member):
        self.member = member
        self.next = None


def build_circle(n):
    """Build the circle 1, 2, ..., n and return its LAST node (whose next is 1)."""
    first = CNode(1)
    last = first
    for m in range(2, n + 1):
        last.next = CNode(m)
        last = last.next
    last.next = first                           # close the circle
    return last


def josephus_list(n, k):
    prev = build_circle(n)  # the node BEFORE the next one counted
    print(f"n={n}, k={k}, leaving in order:", end="")
    while prev.next is not prev:                # more than one member remains
        for _ in range(k - 1):                  # count k-1 members
            prev = prev.next
        out = prev.next                         # the k-th member leaves
        prev.next = out.next                    # link around it
        print(f" {out.member}", end="")
    survivor = prev.member
    prev.next = None                            # break the last self-loop
    print(f" | survivor {survivor}")
    return survivor


def josephus_formula(n, k):
    """J(1) = 0 and J(n) = (J(n-1) + k) mod n, counting members from 0."""
    if n == 1:
        return 0
    return (josephus_formula(n - 1, k) + k) % n
```

Kode 12.42 (Lanjutan) *p1212_josephus.py: Soal 12.12 dalam Python.*

```python
def main():
    for n, k in ((7, 3), (5, 2), (6, 1), (1, 4), (10, 3)):
        a = josephus_list(n, k)
        b = josephus_formula(n, k) + 1          # +1: members are numbered from 1
        print(f"  formula says {b}: {'agree' if a == b else 'DISAGREE'}")


if __name__ == "__main__":
    main()
```

Versi Python menyetel `prev.next = None` pada akhirnya. Satu simpul yang `next`-nya adalah dirinya sendiri merupakan *reference cycle* (siklus referensi); *garbage collector* Python akan mengambilnya kembali pada suatu saat, tetapi memutus siklus itu membuat penghitungan referensi langsung membebaskannya [10].

Penyelesaian keliru yang menggoda. Mulailah `prev` di anggota 1 alih-alih di anggota sebelumnya: `CNode *prev = buildCircle(n)->next;`. Setiap hitungan kini terlambat satu anggota, dan `wrong/p1212_wrong.cpp` mencetak:

```
n=7, k=3, leaving in order: 4 7 3 1 6 2 | survivor 5
  formula says 4: DISAGREE
n=5, k=2, leaving in order: 3 5 2 1 | survivor 4
  formula says 3: DISAGREE
n=6, k=1, leaving in order: 2 3 4 5 6 | survivor 1
  formula says 6: DISAGREE
n=1, k=4, leaving in order: | survivor 1
  formula says 1: agree
n=10, k=3, leaving in order: 4 7 10 3 8 2 9 6 1 | survivor 5
  formula says 4: DISAGREE
```

Program itu berjalan, mencetak barisan yang masuk akal, dan tidak pernah *crash*; hanya pemeriksaan yang independen yang membongkarnya, dan itu pun hanya untuk $n \geq 2$. Kasus $k = 1$ memperlihatkan kesalahannya paling gamblang: "setiap anggota pertama keluar" seharusnya mengeluarkan 1 lebih dulu, tetapi yang keluar 2. Pelajarannya adalah nilai dari perhitungan kedua yang independen atas jawaban yang sama.

> **UJI DIRI.** Pakailah Persamaan (12.8) untuk mencari anggota yang tersisa untuk $n = 10$, $k = 2$. Lalu carilah untuk $n = 16$ dan untuk $n = 17$, $k = 2$, dan tebaklah polanya.
>
> *Jawaban.* Untuk $k = 2$: $J(1..10) = 0,0,2,0,2,4,6,0,2,4$, jadi yang tersisa dari 10 anggota adalah anggota 5. $J(16) = 0$ (anggota 1) dan $J(17) = 2$ (anggota 3). Pola untuk $k = 2$: tuliskan $n = 2^m + \ell$ dengan $0 \leq \ell < 2^m$; yang tersisa adalah anggota $2\ell + 1$ [14]. Untuk $n = 10 = 8 + 2$ hasilnya 5.

12.15 Soal Baru yang Dikerjakan: Rekursi

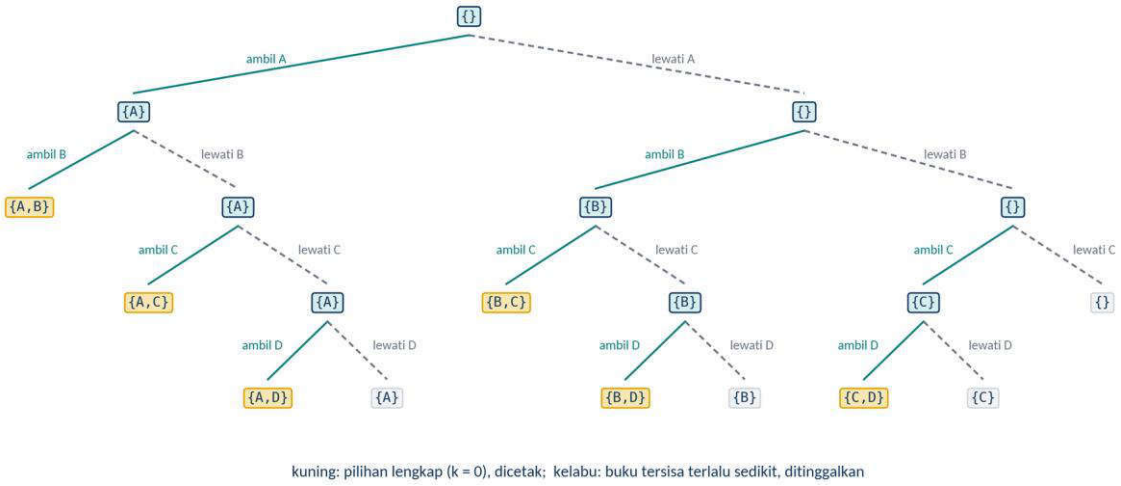
12.15.1 Soal 12.13 — Memilih buku

> **SOAL.** Seorang anggota boleh meminjam tiga dari lima buku baru `B1001` sampai `B1005`. Tulislah fungsi rekursif yang mencetak setiap pilihan tiga buku yang mungkin, masing-masing sekali, dengan buku-buku setiap pilihan dalam urutan katalog, dan hitunglah banyaknya pilihan. Periksalah banyaknya dengan rumus rekursif koefisien binomial $\binom{n}{k}$.

Dengan kata-kata kita sendiri. Urutan tidak penting (mengambil `B1001` lalu `B1002` sama dengan sebaliknya), jadi yang kita inginkan adalah *kombinasi*. Gagasan rekursifnya adalah keputusan ya-atau-tidak tentang satu buku setiap kalinya. Lihatlah buku pertama yang belum diputuskan, `books[start]`. Setiap pilihan entah **mengambil**-nya, lalu memerlukan $k - 1$ buku lagi dari buku-buku sesudahnya, atau **melewati**-nya, lalu memerlukan semua k buku dari buku-buku sesudahnya. Kedua cabang tidak pernah menghasilkan pilihan yang sama (yang satu memuat buku itu, yang lain tidak), dan bersama-sama keduanya menghasilkan setiap pilihan. Menghitung kedua cabang itu memberikan rekurens segitiga Pascal yang terkenal:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \qquad \binom{n}{0} = \binom{n}{n} = 1. \tag{12.9}$$

***Base case*-nya.** Jika $k = 0$, pilihannya lengkap: cetaklah. Jika buku yang tersisa kurang dari k, tidak ada pilihan yang dapat dilengkapi di cabang ini: menyerahlah (ini jalan buntu, dan kembali lebih awal seperti ini disebut *pruning*, pemangkasan). **Gambar 12.13** menggambar seluruh rekursinya untuk kasus yang lebih kecil, dua buku dari empat, supaya muat dalam satu halaman.



Gambar 12.13 Rekursi `choose` untuk dua buku dari A, B, C, D. Setiap pemanggilan entah mengambil atau melewati buku berikutnya; keenam daun kuning adalah keenam pilihan, dan daun kelabu adalah cabang yang ditinggalkan karena buku yang tersisa terlalu sedikit.

***Backtracking*.** Pilihan-pilihan yang sudah dibuat disimpan dalam satu list bersama, `picked`. Sebelum cabang "ambil", fungsi menambahkan bukunya; sesudah cabang itu kembali, fungsi *membuang* buku itu lagi, sehingga cabang "lewati" dimulai dari keadaan yang persis sama seperti seandainya buku itu tidak pernah diambil. Pola tambah, jelajahi, buang ini disebut *backtracking* (runut balik), dan pembuangan itulah langkah yang paling mudah terlupa.

Kode 12.43 adalah program C++ untuk Soal 12.13.

Kode 12.43 `p1213_choose_books.cpp`: Soal 12.13, setiap pilihan tiga buku dengan rekursi ambil-atau-lewati dan *backtracking*.

```cpp
// p1213_choose_books.cpp - Problem 12.13: all ways to choose k of n books.
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

// Choose k more books from books[start..]; picked holds the choices made so far.
void choose(const vector<string> &books, int start, int k,
            vector<string> &picked, int &ways) {
    if (k == 0) {                                   // a complete choice: print it
        ways++;
        printf("%2d:", ways);
        for (const string &b : picked) printf(" %s", b.c_str());
        printf("\n");
        return;
    }
    int left = (int)books.size() - start;           // books not yet considered
    if (left < k) return;                           // too few left: a dead end
    picked.push_back(books[start]);                 // branch 1: take books[start]
    choose(books, start + 1, k - 1, picked, ways);
    picked.pop_back();                              // undo that choice
    choose(books, start + 1, k, picked, ways);      // branch 2: skip books[start]
}

// C(n, k) = C(n-1, k-1) + C(n-1, k): the same two branches, only counted.
long long binom(int n, int k) {
    if (k == 0 || k == n) return 1;
    return binom(n - 1, k - 1) + binom(n - 1, k);
}

int main() {
    vector<string> books = {"B1001", "B1002", "B1003", "B1004", "B1005"};
    vector<string> picked;
    int ways = 0;
    choose(books, 0, 3, picked, ways);
    printf("ways printed: %d, C(5,3) = %lld, C(10,4) = %lld, C(20,10) = %lld\n",
           ways, binom(5, 3), binom(10, 4), binom(20, 10));
    return 0;
}
```

```
 1: B1001 B1002 B1003
 2: B1001 B1002 B1004
 3: B1001 B1002 B1005
 4: B1001 B1003 B1004
 5: B1001 B1003 B1005
 6: B1001 B1004 B1005
 7: B1002 B1003 B1004
 8: B1002 B1003 B1005
 9: B1002 B1004 B1005
10: B1003 B1004 B1005
ways printed: 10, C(5,3) = 10, C(10,4) = 210, C(20,10) = 184756
```

Kesepuluh pilihan keluar dalam urutan katalog, karena cabang “ambil” dijelajahi sebelum cabang “lewati”, dan banyaknya sesuai dengan $\binom{5}{3} = 10$. Program yang sama juga menghitung $\binom{10}{4} = 210$ dan $\binom{20}{10} = 184\,756$ dengan Persamaan (12.9). Bilangan terakhir itu menyembunyikan sebuah peringatan dari Bab 10: `binom` rekursif mengembalikan 1 di setiap daunnya, sehingga untuk mencapai 184 756 ia melakukan $2 \cdot 184\,756 - 1 = 369\,511$ pemanggilan, banyak di antaranya mengulang submasalah yang sama, persis seperti Fibonacci naif pada Subbab 10.5. Menyimpan setiap $\binom{i}{j}$ yang sudah dihitung, dalam sebuah tabel, akan mengurangi pemanggilannya menjadi sekitar $n \cdot k$.

Kode 12.44 dan **Kode 12.45** adalah versi Java dan Python-nya; keduanya mencetak baris-baris yang persis sama.

Kode 12.44 `P1213ChooseBooks.java`: Soal 12.13 dalam Java.

```java
// P1213ChooseBooks.java - Problem 12.13: all ways to choose k of n books.
import java.util.ArrayList;
import java.util.List;

public class P1213ChooseBooks {

    static int ways = 0;  // how many choices were printed

    // Choose k more books from books[start..]; picked holds the choices so far.
    static void choose(String[] books, int start, int k, List<String> picked) {
        if (k == 0) {                                  // a complete choice: print it
            ways++;
            System.out.printf("%2d:", ways);
            for (String b : picked) System.out.print(" " + b);
            System.out.println();
            return;
        }
        int left = books.length - start;               // books not yet considered
        if (left < k) return;                          // too few left: a dead end
        picked.add(books[start]);                      // branch 1: take books[start]
        choose(books, start + 1, k - 1, picked);
        picked.remove(picked.size() - 1);              // undo that choice
        choose(books, start + 1, k, picked);           // branch 2: skip books[start]
    }

    // C(n, k) = C(n-1, k-1) + C(n-1, k): the same two branches, only counted.
    static long binom(int n, int k) {
        if (k == 0 || k == n) return 1;
        return binom(n - 1, k - 1) + binom(n - 1, k);
    }

    public static void main(String[] args) {
        String[] books = {"B1001", "B1002", "B1003", "B1004", "B1005"};
        choose(books, 0, 3, new ArrayList<>());
        System.out.printf("ways printed: %d, C(5,3) = %d, C(10,4) = %d, "
                + "C(20,10) = %d%n",
                ways, binom(5, 3), binom(10, 4), binom(20, 10));
    }
}
```

Kode 12.45 `p1213_choose_books.py`: Soal 12.13 dalam Python.

```python
# p1213_choose_books.py - Problem 12.13: all ways to choose k of n books.


def choose(books, start, k, picked, ways):
    """Choose k more books from books[start:]; picked holds the choices so far.
    ways is a one-element list, so that every call adds to the same counter."""
    if k == 0:                                  # a complete choice: print it
        ways[0] += 1
        print(f"{ways[0]:2d}:", " ".join(picked))
        return
    left = len(books) - start                   # books not yet considered
    if left < k:                                # too few left: a dead end
        return
    picked.append(books[start])                 # branch 1: take books[start]
    choose(books, start + 1, k - 1, picked, ways)
    picked.pop()                                # undo that choice
    choose(books, start + 1, k, picked, ways)   # branch 2: skip books[start]


def binom(n, k):
    """C(n, k) = C(n-1, k-1) + C(n-1, k): the same two branches, only counted."""
    if k == 0 or k == n:
        return 1
    return binom(n - 1, k - 1) + binom(n - 1, k)


def main():
    books = ["B1001", "B1002", "B1003", "B1004", "B1005"]
    ways = [0]
    choose(books, 0, 3, [], ways)
    print(f"ways printed: {ways[0]}, C(5,3) = {binom(5, 3)}, "
          f"C(10,4) = {binom(10, 4)}, C(20,10) = {binom(20, 10)}")


if __name__ == "__main__":
    main()
```

Penghitungnya dipakai bersama oleh semua pemanggilan: referensi `int &ways` di C++, medan `static` di Java, list satu elemen di Python.

Biaya. Ada $\binom{n}{k}$ pilihan, masing-masing dicetak dalam $O(k)$, dan dengan pemangkasan, jalan-jalan buntu menambah paling banyak faktor konstan; tidak ada algoritme yang mencetak setiap pilihan yang dapat lebih baik.

Penyelesaian keliru yang menggoda. Lupakan `pop_back` sesudah cabang "ambil". `wrong/p1213_wrong.cpp` mencetak sepuluh baris, tetapi isinya kacau:

```
 1: B1001 B1002 B1003
 2: B1001 B1002 B1003 B1004
 3: B1001 B1002 B1003 B1004 B1005
 4: B1001 B1002 B1003 B1004 B1005 B1003 B1004
```

(dan seterusnya, setiap baris lebih panjang daripada sebelumnya). Banyaknya pilihan lengkap tetap 10, karena penghitungannya hanya bergantung pada k dan `start`, yang dikirim *by value* dan dipulihkan secara otomatis ketika sebuah pemanggilan kembali. List `picked` *dipakai bersama*, sehingga tidak ada yang memulihkannya kecuali `pop_back` kita sendiri. Setiap buku yang pernah diambil tetap tertinggal di dalamnya. Aturan umumnya: apa pun yang diterima pemanggilan rekursif *by value* dipulihkan dengan cuma-cuma; apa pun yang diubahnya pada keadaan bersama harus dipulihkannya sendiri.

UJI DIRI. Berapa pemanggilan `choose` yang dilakukan untuk $n = 4$, $k = 2$ (hitunglah di **Gambar 12.13**)? Dan apa yang harus diubah pada `choose` untuk mencetak setiap pilihan *paling banyak* k buku, termasuk pilihan kosong?

Jawaban. Gambar itu memuat 16 kotak: 16 pemanggilan, 6 di antaranya mencetak dan 3 jalan buntu. Untuk "paling banyak k buku", cetaklah `picked` di *setiap* pemanggilan, sebelum kedua cabang, dan hentikan rekursi ketika $k = 0$ atau ketika tidak ada buku yang tersisa: itu mencetak $\binom{4}{0} + \binom{4}{1} + \binom{4}{2} = 1 + 4 + 6 = 11$ pilihan untuk $n = 4, k = 2$.

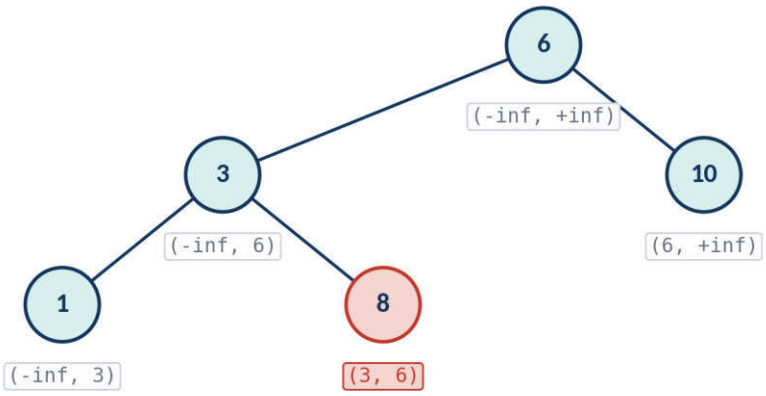
12.16 Soal Baru yang Dikerjakan: Pohon

12.16.1 Soal 12.14 — Benarkah ini *binary search tree*?

SOAL. Sebuah pohon nomor katalog dibangun dengan tangan, bukan dengan `insert`. Tulislah fungsi yang memutuskan apakah pohon itu *binary search tree*, tanpa dua nilai yang sama.

Dengan kata-kata kita sendiri. Kita harus memeriksa Persamaan (12.3) di setiap simpul, terhadap *semua* keturunannya, bukan hanya anak-anaknya. Kiat yang membuat ini cukup satu penelusuran, bukan banyak, adalah mengirim batas-batas ke bawah. Akar boleh memuat nilai apa pun. Semua yang ada di subpohon kirinya harus di bawah nilai akar; semua yang ada di subpohon kanannya harus di atasnya. Dalam perjalanan turun, setiap pemanggilan mengetahui sebuah interval terbuka $(low, high)$ tempat nilainya harus berada: ke kiri, nilai simpul menjadi `high` yang baru; ke kanan, nilai itu menjadi `low` yang baru. Pohon kosong adalah pohon pencarian (*base case*: benar).

Dengan tangan. **Gambar 12.14** memperlihatkan pohon yang dibuat untuk mengelabui pemeriksaan yang ceroboh: 6 dengan anak 3 dan 10, dan 3 dengan anak 1 dan 8. Setiap simpul benar *dibandingkan dengan anak-anaknya sendiri*: $1 < 3 < 8$ dan $3 < 6 < 10$. Tetapi 8 berada di subpohon kiri 6, jadi ia harus di bawah 6, dan ternyata tidak. Dengan batas-batas, pemanggilan pada 8 menerima interval (3,6): ia datang dari kiri 6 (jadi $high = 6$) lalu dari kanan 3 (jadi $low = 3$), dan $8 \notin (3,6)$.



Gambar 12.14 Interval yang diterima setiap pemanggilan `inRange`. Setiap pasangan induk–anak berurutan, tetapi nilai 8 berada di luar intervalnya (3,6), sehingga pohon itu bukan *binary search tree*.

Kode 12.46 adalah program C++ untuk Soal 12.14.

Kode 12.46 `p1214_is_bst.cpp`: Soal 12.14, setiap simpul diperiksa terhadap interval yang diwarisi dari leluhurnya.

```cpp
// p1214_is_bst.cpp - Problem 12.14: is this binary tree a binary search tree?
#include <climits>
#include "tree_kit.hpp"

// Every value in this subtree must lie strictly between low and high.
bool inRange(Node *root, long long low, long long high) {
    if (root == nullptr) return true;
    if (root->data <= low || root->data >= high) return false;
    return inRange(root->left, low, root->data) &&     // left: below this value
           inRange(root->right, root->data, high);     // right: above this value
}

bool isBST(Node *root) {
    return inRange(root, LLONG_MIN, LLONG_MAX);        // no limits at the root
}

Node *make(int data, Node *left, Node *right) {        // build a tree by hand
    return new Node{data, left, right};
}

int main() {
    Node *sample = nullptr;
    buildSample(&sample);
    // 6 with left child 3 (children 1 and 8) and right child 10:
    // the 8 is on the wrong side of the 6.
    Node *sneaky = make(6, make(3, make(1, nullptr, nullptr),
                                   make(8, nullptr, nullptr)),
                           make(10, nullptr, nullptr));
    // 6 whose left child is another 6: equal values are not allowed.
    Node *twin = make(6, make(6, nullptr, nullptr), nullptr);
    Node *empty = nullptr;
    printf("sample tree  : %s\n", isBST(sample) ? "BST" : "not a BST");
    printf("sneaky tree  : %s\n", isBST(sneaky) ? "BST" : "not a BST");
    printf("twin sixes   : %s\n", isBST(twin) ? "BST" : "not a BST");
    printf("empty tree   : %s\n", isBST(empty) ? "BST" : "not a BST");
    printf("sneaky inorder: ");
    inOrder(sneaky);
    printf("\n");
    clear(&sample);
    clear(&sneaky);
    clear(&twin);
    return 0;
}
```

```
sample tree  : BST
sneaky tree  : not a BST
twin sixes   : not a BST
empty tree   : BST
sneaky inorder: 1 3 8 6 10
```

Pohon contoh lolos; pohon pengelabu gagal, demikian pula pohon dengan dua nilai yang sama (intervalnya terbuka, sehingga kesamaan ditolak); pohon kosong lolos. Baris terakhir mencetak *inorder* pohon pengelabu, `1 3 8 6 10`, yang tidak menaik: itulah pengujian kedua yang setara, yang dipakai uji diri di bawah. Batas-batasnya dimulai dari `LLONG_MIN` dan `LLONG_MAX`, yang berada di luar jangkauan nilai `int` mana pun, sehingga tidak pernah menolak nilai sungguhan; Java memakai `Long.MIN_VALUE` dan `Long.MAX_VALUE`, dan Python `-math.inf` dan `math.inf`. **Biaya.** $\Theta(n)$ pada kasus terburuk: setiap simpul diperiksa sekali.

Kode 12.47 dan **Kode 12.48** adalah versi Java dan Python-nya; keduanya mencetak baris-baris yang persis sama.

Kode 12.47 `P1214IsBst.java`: Soal 12.14 dalam Java.

```java
// P1214IsBst.java - Problem 12.14: is this binary tree a binary search tree?
public class P1214IsBst extends TreeKit {

    // Every value in this subtree must lie strictly between low and high.
    static boolean inRange(Node root, long low, long high) {
        if (root == null) return true;
        if (root.data <= low || root.data >= high) return false;
        return inRange(root.left, low, root.data)       // left: below this value
                && inRange(root.right, root.data, high); // right: above this value
    }

    static boolean isBST(Node root) {
        return inRange(root, Long.MIN_VALUE, Long.MAX_VALUE);   // no limits yet
    }

    static Node make(int data, Node left, Node right) {   // build a tree by hand
        return new Node(data, left, right);
    }

    static String verdict(Node root) {
        return isBST(root) ? "BST" : "not a BST";
    }

    public static void main(String[] args) {
        Node sample = buildSample();
        // 6 with left child 3 (children 1 and 8) and right child 10:
        // the 8 is on the wrong side of the 6.
        Node sneaky = make(6, make(3, make(1, null, null), make(8, null, null)),
                make(10, null, null));
        // 6 whose left child is another 6: equal values are not allowed.
        Node twin = make(6, make(6, null, null), null);
        System.out.println("sample tree  : " + verdict(sample));
        System.out.println("sneaky tree  : " + verdict(sneaky));
        System.out.println("twin sixes   : " + verdict(twin));
        System.out.println("empty tree   : " + verdict(null));
        System.out.print("sneaky inorder: ");
        inOrder(sneaky);
        System.out.println();
    }
}
```

Kode 12.48 `p1214_is_bst.py`: Soal 12.14 dalam Python.

```python
# p1214_is_bst.py - Problem 12.14: is this binary tree a binary search tree?
import math
from tree_kit import *


def in_range(root, low, high):
    """Every value in this subtree must lie strictly between low and high."""
    if root is None:
        return True
    if root.data <= low or root.data >= high:
        return False
    return (in_range(root.left, low, root.data) and      # left: below this value
            in_range(root.right, root.data, high))       # right: above this value


def is_bst(root):
    return in_range(root, -math.inf, math.inf)            # no limits at the root


def main():
    sample = build_sample()
    # 6 with left child 3 (children 1 and 8) and right child 10:
    # the 8 is on the wrong side of the 6.
    sneaky = Node(6, Node(3, Node(1), Node(8)), Node(10))
    # 6 whose left child is another 6: equal values are not allowed.
    twin = Node(6, Node(6))
    verdict = lambda t: "BST" if is_bst(t) else "not a BST"
    print("sample tree  :", verdict(sample))
    print("sneaky tree  :", verdict(sneaky))
    print("twin sixes   :", verdict(twin))
    print("empty tree   :", verdict(None))
    print("sneaky inorder: ", end="")
    in_order(sneaky)
    print()


if __name__ == "__main__":
    main()
```

Penyelesaian keliru yang menggoda. Bandingkan setiap simpul hanya dengan anak-anaknya sendiri, lalu lakukan rekursi:

```
if (root->left != nullptr && root->left->data >= root->data) return false;
if (root->right != nullptr && root->right->data <= root->data) return false;
return isBST(root->left) && isBST(root->right);
```

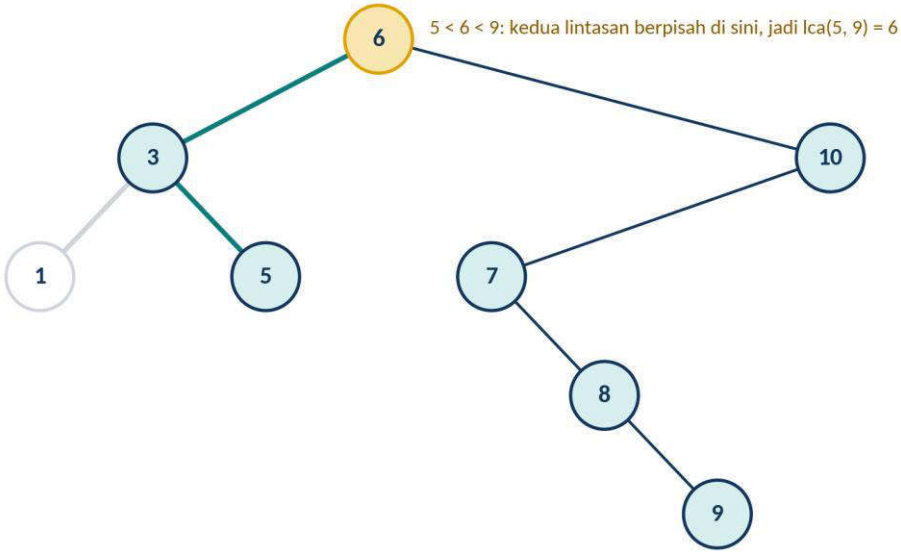
`wrong/p1214_wrong.cpp` menjawab `sneaky tree  : BST`. Pemeriksaan ini hanya menegakkan aturan untuk induk dan anak, padahal aturan Persamaan (12.3) menyangkut leluhur dan keturunan. Akibatnya bukan sekadar teori: pada pohon pengelabu, `search(root, 8)` ke kanan di 6 (karena $8 > 6$), menemukan 10, ke kiri ke `nullptr`, dan melaporkan bahwa 8 tidak ada, padahal 8 ada di pohon.

> **UJI DIRI.** Tulislah pengujian *inorder*: telusuri pohon secara *inorder* dan periksa bahwa setiap nilai lebih besar daripada nilai yang dikunjungi sebelumnya. Harus berupa apa "nilai sebelumnya" itu, dan bagaimana ia dipakai bersama oleh pemanggilan-pemanggilan?
>
> *Jawaban.* Nilai sebelumnya harus dipakai bersama oleh semua pemanggilan dan bertahan melewati kembalinya mereka, persis seperti `prev` pada Soal 12.10: kirimkan `long long *prev` (atau `long long &prev`), dimulai dari `LLONG_MIN`. Di setiap simpul, sesudah subpohon kiri: `if (root->data <= *prev) return false; *prev = root->data;` lalu periksa subpohon kanan. Pengujian ini mengunjungi nilai-nilai dalam urutan 1, 3, 8, 6, dan gagal di 6 karena $6 \leq 8$.

12.16.2 Soal 12.15 — Leluhur bersama terendah

> **SOAL.** Dalam *binary search tree* yang memuat kedua nilai a dan b, carilah *leluhur bersama terendah* (*lowest common ancestor*) keduanya: simpul terdalam yang memuat a dan b di subpohonnya (sebuah simpul dihitung berada di dalam subpohonnya sendiri).

Dengan kata-kata kita sendiri. Turunlah dari akar. Selama a dan b keduanya lebih kecil daripada simpul saat ini, keduanya berada di subpohon kirinya, demikian pula leluhur bersama terendahnya: ke kiri. Selama keduanya lebih besar, ke kanan. Di simpul pertama tempat keduanya *tidak* berada di sisi yang sama (yang satu lebih kecil dan yang lain lebih besar, atau salah satunya *adalah* simpul itu), lintasan keduanya berpisah, dan simpul itulah jawabannya. **Gambar 12.15** memperlihatkan kasus $a = 5$, $b = 9$: di 6, 5 ke kiri dan 9 ke kanan, jadi jawabannya langsung 6.



Gambar 12.15 Lintasan dari akar ke 5 (hijau toska) dan ke 9 (biru tua) hanya berbagi akar: karena $5 < 6 < 9$, kedua nilai berpisah di 6, yang merupakan leluhur bersama terendahnya.

Dengan tangan pada pasangan-pasangan lainnya: untuk 1 dan 5, keduanya di bawah 6 (ke kiri), dan di 3 keduanya berpisah: 3. Untuk 8 dan 9, keduanya di atas 6, lalu keduanya di bawah 10, lalu keduanya di atas 7; di 8 salah satunya *adalah* simpul itu: 8. Untuk 7 dan 9: 7. Untuk 1 dan 10: berpisah di akar: 6. Untuk 9 dan 9: simpul 9 itu sendiri.

Kode 12.49 adalah program C++ untuk Soal 12.15.

Kode 12.49 `p1215_lca.cpp`: Soal 12.15, satu penelusuran turun sampai kedua nilai berpisah.

```cpp
// p1215_lca.cpp - Problem 12.15: the lowest common ancestor of two BST values.
#include "tree_kit.hpp"

// The deepest node with both a and b in its subtree (both must be in the tree).
Node *lca(Node *root, int a, int b) {
    if (root == nullptr) return nullptr;
    if (a < root->data && b < root->data) {     // both on the left
        return lca(root->left, a, b);
    }
    if (a > root->data && b > root->data) {     // both on the right
        return lca(root->right, a, b);
    }
    return root;                 // they part here, or one of them is here
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    const int pairs[][2] = {{1, 5}, {5, 9}, {8, 9}, {7, 9}, {1, 10}, {9, 9}};
    for (const auto &p : pairs) {
        printf("lca(%d, %d) = %d\n", p[0], p[1], lca(root, p[0], p[1])->data);
    }
    clear(&root);
    return 0;
}
```

```
lca(1, 5) = 3
lca(5, 9) = 6
lca(8, 9) = 8
lca(7, 9) = 7
lca(1, 10) = 6
lca(9, 9) = 9
```

Keenam jawaban sesuai dengan perhitungan tangan. Fungsi ini menempuh satu lintasan, sehingga berbiaya $O(h)$; pada pohon biner umum, tanpa urutan pohon pencarian sebagai penunjuk arah, mencari leluhur bersama terendah memerlukan penelusuran seluruh pohon, $\Theta(n)$.

Kode 12.50 dan **Kode 12.51** adalah versi Java dan Python-nya; keduanya mencetak baris-baris yang persis sama.

Kode 12.50 `P1215Lca.java`: Soal 12.15 dalam Java.

```java
// P1215Lca.java - Problem 12.15: the lowest common ancestor of two BST values.
public class P1215Lca extends TreeKit {

    // The deepest node with both a and b in its subtree (both must be present).
    static Node lca(Node root, int a, int b) {
        if (root == null) return null;
        if (a < root.data && b < root.data) {       // both on the left
            return lca(root.left, a, b);
        }
        if (a > root.data && b > root.data) {       // both on the right
            return lca(root.right, a, b);
        }
        return root;                // they part here, or one of them is here
    }

    public static void main(String[] args) {
        Node root = buildSample();
        int[][] pairs = {{1, 5}, {5, 9}, {8, 9}, {7, 9}, {1, 10}, {9, 9}};
        for (int[] p : pairs) {
            Node answer = lca(root, p[0], p[1]);
            System.out.printf("lca(%d, %d) = %d%n", p[0], p[1], answer.data);
        }
    }
}
```

Kode 12.51 `p1215_lca.py`: Soal 12.15 dalam Python.

```python
# p1215_lca.py - Problem 12.15: the lowest common ancestor of two values in a BST.
from tree_kit import *


def lca(root, a, b):
    """The deepest node with both a and b in its subtree (both must be in it)."""
    if root is None:
        return None
    if a < root.data and b < root.data:          # both on the left
        return lca(root.left, a, b)
    if a > root.data and b > root.data:          # both on the right
        return lca(root.right, a, b)
    return root                    # they part here, or one of them is here


def main():
    root = build_sample()
    for a, b in ((1, 5), (5, 9), (8, 9), (7, 9), (1, 10), (9, 9)):
        print(f"lca({a}, {b}) = {lca(root, a, b).data}")


if __name__ == "__main__":
    main()
```

Penyelesaian keliru yang menggoda. Arahkan dengan satu nilai saja, seperti `search`: `if (a < root->data) go left; if (a > root->data) go right;`. `wrong/p1215_wrong.cpp` menjawab `lca(1, 5) = 1`, `lca(5, 9) = 5`, dan `lca(1, 10) = 1`, dan benar pada tiga lainnya secara kebetulan. Mengarahkan dengan a saja sekadar berjalan ke a dan mengembalikannya; jawabannya benar hanya bila b kebetulan terletak di bawah a. Penelusuran harus berlanjut hanya selama *kedua* nilai menunjuk ke arah yang sama.

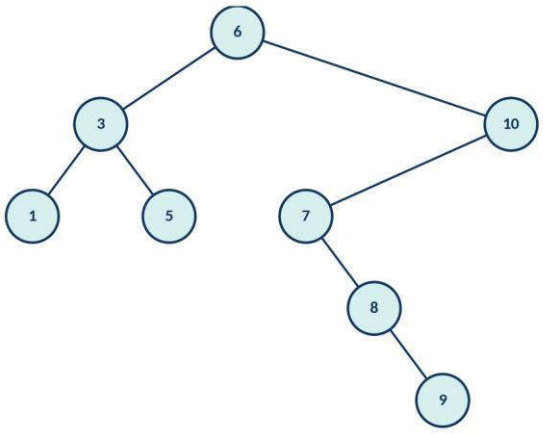
> **UJI DIRI.** Pakailah leluhur bersama terendah untuk menghitung banyaknya sisi di antara dua simpul, $dist(a, b)$, dengan memakai kedalaman $d(x)$ sebuah simpul. Berapa $dist(5{,}9)$?
>
> *Jawaban.* $dist(a, b) = d(a) + d(b) - 2\,d(lca(a, b))$: naik dari a ke leluhur bersama lalu turun ke b. Untuk 5 dan 9: $2 + 4 - 2 \cdot 0 = 6$ sisi (5, 3, 6, 10, 7, 8, 9).

12.16.3 Soal 12.16 — Mencetak pohon tingkat demi tingkat

> **SOAL.** Cetaklah pohon satu tingkat per baris, setiap baris diawali nomor tingkatnya, dengan satu penelusuran pohon. Laporkan juga lebar tingkat yang terlebar.

Dengan kata-kata kita sendiri. Mencetak setiap tingkat dengan Soal 12.2 berbiaya sampai $\Theta(n^2)$. Penelusuran *level order* pada Subbab 11.6.3 memakai queue (Bab 5): ambil simpul dari depan, cetak, taruh anak-anaknya di belakang; sehingga seluruh tingkat d keluar dari queue sebelum simpul mana pun dari tingkat $d+1$. Untuk mengetahui *di mana sebuah tingkat berakhir*, perhatikan bahwa pada awal setiap tingkat queue memuat tepat tingkat itu, sehingga ukurannya pada saat itu adalah lebar tingkat tersebut: ambillah tepat sebanyak itu simpul, maka tingkat itu selesai.

Dengan tangan. **Gambar 12.16** mengikuti queue satu tingkat demi satu tingkat.



tingkat	lebar	dikeluarkan (dicetak)	dimasukkan
0	1	6	3 10
1	2	3 10	1 5 7
2	3	1 5 7	8
3	1	8	9
4	1	9	-

Gambar 12.16 Soal 12.16: pada awal setiap tingkat, queue memuat tepat tingkat itu; ukurannya adalah lebar tingkat tersebut. Simpul yang diambil dari queue dicetak, dan anak-anaknya dimasukkan untuk tingkat berikutnya.

Pada awalnya queue memuat `6`, lebar 1: ambil 6, masukkan 3 dan 10. Queue kini memuat `3 10`, lebar 2: ambil keduanya, masukkan 1 dan 5 (anak-anak 3) dan 7 (anak 10). Lebar 3: ambil `1 5 7`, masukkan 8. Lalu `8`, masukkan 9. Lalu `9`, dan queue kosong.

Kode 12.52 adalah program C++ untuk Soal 12.16.

Kode 12.52 `p1216_levels.cpp`: Soal 12.16, penelusuran *level order* yang mencatat lebar setiap tingkat.

```cpp
// p1216_levels.cpp - Problem 12.16: print a tree level by level with one queue.
#include <queue>
#include "tree_kit.hpp"

// Breadth-first: all of level d leave the queue before any node of level d+1.
int printLevels(Node *root) {  // returns the width of the widest level
    int widest = 0;
    if (root == nullptr) return widest;
    std::queue<Node *> q;
    q.push(root);
    for (int level = 0; !q.empty(); level++) {
        int width = (int)q.size();            // exactly the nodes of this level
        if (width > widest) widest = width;
        printf("level %d:", level);
        for (int i = 0; i < width; i++) {
            Node *cur = q.front();
            q.pop();
            printf(" %d", cur->data);
            if (cur->left != nullptr) q.push(cur->left);     // they belong to
            if (cur->right != nullptr) q.push(cur->right);   // the next level
        }
        printf("\n");
    }
    return widest;
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    printf("widest level has %d nodes\n", printLevels(root));
    Node *full = nullptr;
    insertAll(&full, {8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15});
    printf("widest level has %d nodes\n", printLevels(full));
    clear(&root);
    clear(&full);
    return 0;
}
```

```
level 0: 6
level 1: 3 10
level 2: 1 5 7
level 3: 8
level 4: 9
widest level has 3 nodes
level 0: 8
level 1: 4 12
level 2: 2 6 10 14
level 3: 1 3 5 7 9 11 13 15
widest level has 8 nodes
```

Tingkat terlebar pohon contoh adalah tingkat 2, dengan 3 simpul. Pohon kedua, yang dibangun dari 8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15, adalah pohon *sempurna* bertinggi 3, dan tingkat-tingkatnya memuat 1,2,4,8 simpul, yaitu maksimum 2^k yang diizinkan Subbab 12.3.1 pada tingkat k. **Biaya.** Setiap simpul masuk dan keluar queue sekali: waktu $\Theta(n)$, dan queue memuat paling banyak satu tingkat ditambah awal tingkat berikutnya, memori $O(\text{lebar})$, yang untuk pohon sempurna sekitar $n/2$.

Kode 12.53 dan **Kode 12.54** adalah versi Java dan Python-nya; keduanya mencetak baris-baris yang persis sama.

Kode 12.53 `P1216Levels.java`: Soal 12.16 dalam Java.

```java
// P1216Levels.java - Problem 12.16: print a tree level by level with one queue.
import java.util.ArrayDeque;

public class P1216Levels extends TreeKit {

    // Breadth-first: all of level d leave the queue before any node of level d+1.
    static int printLevels(Node root) {  // returns the width of the widest level
        int widest = 0;
        if (root == null) return widest;
        ArrayDeque<Node> q = new ArrayDeque<>();
        q.add(root);
        for (int level = 0; !q.isEmpty(); level++) {
            int width = q.size();               // exactly the nodes of this level
            if (width > widest) widest = width;
            System.out.print("level " + level + ":");
            for (int i = 0; i < width; i++) {
                Node cur = q.poll();
                System.out.print(" " + cur.data);
                if (cur.left != null) q.add(cur.left);     // they belong to
                if (cur.right != null) q.add(cur.right);   // the next level
            }
            System.out.println();
        }
        return widest;
    }

    public static void main(String[] args) {
        Node root = buildSample();
        System.out.println("widest level has " + printLevels(root) + " nodes");
        Node full = insertAll(null, 8, 4, 12, 2, 6, 10, 14,
                1, 3, 5, 7, 9, 11, 13, 15);
        System.out.println("widest level has " + printLevels(full) + " nodes");
    }
}
```

Kode 12.54 `p1216_levels.py`: Soal 12.16 dalam Python.

```python
# p1216_levels.py - Problem 12.16: print a tree level by level with one queue.
from collections import deque
from tree_kit import *


def print_levels(root):
    """Breadth-first: every node of level d leaves the queue before level d+1.
    Returns the width of the widest level."""
    widest = 0
    if root is None:
        return widest
    q = deque([root])
    level = 0
    while q:
        width = len(q)                          # exactly the nodes of this level
        widest = max(widest, width)
        print(f"level {level}:", end="")
        for _ in range(width):
            cur = q.popleft()
            print(f" {cur.data}", end="")
            if cur.left is not None:            # they belong to
                q.append(cur.left)
            if cur.right is not None:           # the next level
                q.append(cur.right)
        print()
        level += 1
    return widest


def main():
    root = build_sample()
    print(f"widest level has {print_levels(root)} nodes")
    full = insert_all(None, [8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15])
    print(f"widest level has {print_levels(full)} nodes")


if __name__ == "__main__":
    main()
```

Penyelesaian keliru yang menggoda. Bacalah ukurannya di dalam syarat perulangan: `for (int i = 0; i < (int)q.size(); i++)`. Ukuran itu berubah selama perulangan berjalan, karena setiap simpul yang diambil memasukkan anak-anaknya, dan `wrong/p1216_wrong.cpp` mencetak:

```
level 0: 6 3 10
level 1: 1 5
level 2: 7
level 3: 8
level 4: 9
```

Sesudah 6 diambil, queue membesar menjadi dua, sehingga perulangan berjalan lagi dan mengambil 3; queue membesar lagi, dan seterusnya sampai *i* menyusul ukurannya. Tingkat-tingkatnya bercampur. Lebarnya harus dibaca *sekali*, sebelum tingkat itu dimulai, ke dalam variabel yang tidak berubah.

> **UJI DIRI.** Ubahlah `printLevels` sehingga fungsi itu mencetak tingkat-tingkatnya dalam urutan *zigzag*: tingkat 0 dari kiri ke kanan, tingkat 1 dari kanan ke kiri, tingkat 2 dari kiri ke kanan, dan seterusnya. Apa yang dicetaknya untuk pohon contoh?
>
> *Jawaban.* Kumpulkan simpul-simpul setiap tingkat ke dalam larik kecil persis seperti sekarang, lalu cetak larik itu maju pada tingkat genap dan mundur pada tingkat ganjil; queue-nya sendiri tidak berubah. Untuk pohon contoh: `6`, `10 3`, `1 5 7`, `8`, `9`.

12.17 Perbandingan Soal-Soal dalam Bab Ini

Tabel 12.5 menjajarkan keenam belas soal. Bacalah kolom terakhirnya: kebanyakan penyelesaian keliru yang menggoda gagal pada masukan yang tidak ada dalam contoh soalnya sendiri, dan sebagian bahkan lolos dari contoh itu (pemeriksaan keseimbangan yang keliru, jumlah yang keliru pada pemanggilan pertamanya, urutan Josephus yang keliru). Itulah argumen terkuat untuk Langkah 6 dan 7 metode kita.

Tabel 12.5 Keenam belas soal bab ini: struktur yang dipakai masing-masing, biayanya, dan masukan yang membuat penyelesaian keliru yang menggoda gagal. n adalah banyaknya simpul dan h tinggi pohon.

Soal	Struktur	Waktu	Memori tambahan	Masukan yang membuat penyelesaian keliru gagal
12.1	pohon, *preorder*	$\Theta(n)$	stack $O(h)$	setiap pohon (daun memiliki anak `nullptr`)
12.2	pohon, dibatasi kedalaman	O(simpul di atas tingkat k)	$O(k)$	tingkat mana pun di bawah akar
12.3	pohon, jumlah *postorder*	$\Theta(n)$	$O(h)$	pemanggilan kedua mana pun
12.4	pohon, pembaruan *preorder*	$\Theta(n)$	$O(h)$	setiap pohon (hanya salinan yang berubah)
12.5	pohon, tukar di setiap simpul	$\Theta(n)$	$O(h)$	simpul mana pun yang punya anak kiri
12.6	pohon, tinggi	$O(n\log n)$; satu lintasan $\Theta(n)$	$O(h)$	pohon seimbang mana pun, pohon kosong
12.7	BST, satu lintasan	$O(h)$	$O(h)$	penghapusan mana pun di bawah akar
12.8	BST, satu lintasan	$O(h)$	$O(h)$	simpul mana pun di lintasan yang kehilangan satu anak
12.9	BST, `search`	$O(h)$	$O(h)$	nilai yang tidak ada
12.10	pohon ke list, *inorder*	$\Theta(n)$	$O(h)$, tanpa simpul baru	setiap pohon dengan dua tingkat atau lebih
12.11	*double linked list*	$\Theta(n)$	$O(1)$	list mana pun dengan dua simpul atau lebih
12.12	list sirkular; rekurens	$\Theta(nk)$; $\Theta(n)$	$O(n)$	setiap lingkaran berisi dua atau lebih
12.13	rekursi, *backtracking*	$\Theta\left(k\binom{n}{k}\right)$	$O(n)$	setiap masukan (tidak ada yang dipulihkan)
12.14	pohon dengan batas	$\Theta(n)$	$O(h)$	nilai di sisi yang salah dari kakeknya
12.15	BST, satu lintasan	$O(h)$	$O(h)$	dua nilai di sisi yang berbeda
12.16	queue, *level order*	$\Theta(n)$	O(lebar)	tingkat mana pun dengan lebih dari satu simpul

Biaya-biaya itu mengulang tiga pola. **Setiap simpul sekali** berbiaya $\Theta(n)$, menurut Persamaan (12.1). **Satu lintasan dari akar** berbiaya $O(h)$, antara $\lg n$ dan n menurut Persamaan (12.4), dan karena itulah pohon seimbang penting. **Penelusuran di dalam penelusuran**, seperti pemeriksaan keseimbangan yang naif atau mencetak setiap tingkat dengan Soal 12.2, adalah pola yang harus diwaspadai: tanyakan nilai apa yang dihitung ulang oleh

penelusuran dalamnya, dan biarkan penelusuran luarnya *mengembalikan* nilai itu, seperti `checkedHeight` mengembalikan tinggi.

12.18 Profitina dalam Praktik

Profitina adalah platform visibilitas AI (GEO) yang berjalan di produksi di app.profitina.com, yang mengukur seberapa siap situs web sebuah merek untuk dikutip oleh mesin penjawab AI dan membantu memperbaiki apa yang ditemukannya. Kita menggambarkan bentuk soal-soal bab ini dalam produk semacam itu, bukan cara Profitina sendiri dibangun.

Diskusi berutas adalah pohon. Dalam komunitas *real-time* Profitina, setiap balasan menjawab tepat satu kiriman, yaitu induknya, dan kiriman tingkat teratas adalah akar: sebuah hutan pohon, satu per utas. Menampilkan utas dalam urutan baca adalah urutan *depth first* Soal 12.1; indentasi sebuah balasan adalah tingkatnya, Soal 12.2; menghitung balasan di bawah sebuah kiriman adalah jumlah pada Soal 12.3 dengan setiap nilai 1, dan penghitung "N baru" pada kanal yang diikuti hanya menghitung akar-akarnya. Menghapus kiriman yang memiliki balasan memunculkan pertanyaan Soal 12.7, apa yang terjadi pada anak-anaknya: buang subpohonnya, naikkan balasan-balasannya, atau tinggalkan penanda.

Situs web ditelusuri tingkat demi tingkat. Pemindaian situs dimulai dari halaman beranda dan mengikuti tautan; halaman yang berjarak satu klik adalah tingkat 1, dua klik tingkat 2, dan pemindai yang berhenti pada kedalaman tertentu menelusuri situs dengan queue, seperti pada Soal 12.16. (Situs sungguhan bertaut balik dan menyilang, sehingga berupa graf, pokok bahasan Bab 13.)

Rekursi memiliki batas kedalaman. Server Profitina ditulis dalam Python, yang batas rekursi bawaannya sekitar 1000 pemanggilan (Subbab 10.12) [10]. Penampil rekursif memerlukan satu pemanggilan per tingkat, sehingga utas yang bersarang seribu balasan dalamnya akan memunculkan `RecursionError`; pertahanannya adalah yang ada pada Soal 12.1, stack eksplisit, atau batas kedalaman sarang.

Versi latihan yang dapat dijalankan dari gagasan-gagasan ini adalah Profitina Laundry, studi kasus seri buku ini; Soal Latihan 12.12 menyimpan pesanannya dalam *binary search tree* yang berkunci nomor pesanan, dan menanyakan kueri-kueri yang diperlukan meja depan.

12.19 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Tanpa *base case*, atau *base case* yang salah. Setiap fungsi pohon rekursif harus menyatakan apa yang terjadi pada `nullptr` (Soal 12.1), dan jawabannya harus nilai yang tidak mengubah apa pun di bawah langkah penggabungan: 0 untuk $+$, benar untuk `&&`, `INT_MIN` untuk max, -1 untuk tinggi (Soal 12.3 dan 12.6).

Menguji `k == 0` sebelum `root == nullptr`. Uji pointernya dulu, baru pakai (Soal 12.2). Hal yang sama berlaku untuk setiap `root->left->data`: sebuah anak mungkin tidak ada (Soal 12.8), dan `search` mungkin mengembalikan `nullptr` (Soal 12.9).

Berkomunikasi melalui variabel global. Jumlah berjalan di luar fungsi bertahan di antara pemanggilan (Soal 12.3). Kembalikan nilai dan gabungkan; pakailah variabel bersama hanya dengan sengaja, melalui pointer atau di dalam objek, ketika pemanggilan-pemanggilan itu memang harus berbagi (Soal 12.10 dan 12.13).

Mengirim simpul *by value*. `triple(Node node)` mengubah sebuah salinan (Soal 12.4). Terimalah simpul melalui pointer atau referensi.

Menukar tanpa menyimpan. `a = b; b = a;` menghilangkan `a` yang lama (Soal 12.5). Simpan lebih dulu, atau biarkan penugasan *tuple* Python menyimpannya untuk Anda.

Membuang subpohon yang dikembalikan. Fungsi yang mungkin mengganti simpul yang diterimanya harus mengembalikan penggantinya, dan setiap pemanggil harus memasangnya kembali: `root->left = delNode(root->left, x)` (Soal 12.7). Hal yang sama berlaku untuk `root = delNode(root, x)` di `main`.

Memakai fungsi BST pada pohon yang bukan BST. Cermin, pohon yang dinegasikan, atau pohon yang dibangun dengan tangan mungkin melanggar Persamaan (12.3); `search` lalu melewatkan nilai-nilai yang sebenarnya ada (Soal 12.5 dan 12.14).

Hanya memeriksa induk dan anak. Aturan BST menyangkut semua keturunan; kirimkan batas-batas ke bawah (Soal 12.14).

Mengikuti pointer yang baru saja ditimpa. Sesudah menukar `prev` dan `next`, pindahlah ke `next` lama yang disimpan (Soal 12.11).

Lupa memulihkan dalam *backtracking*. Apa pun yang ditambahkan sebuah pemanggilan ke list bersama harus dibuangnya sebelum kembali (Soal 12.13).

Rekursi yang terlalu dalam. Kedalaman rekursi sama dengan tinggi pohon; pada rantai yang panjang, pakailah stack eksplisit atau queue, terutama di Python, yang batas bawaannya sekitar 1000 pemanggilan.

Hanya menguji contohnya. Penyelesaian keliru pada Soal 12.3, 12.6, dan 12.12 lolos dari contohnya sendiri. Ujilah pohon kosong, satu simpul, satu anak di salah satu sisi, nilai yang tidak ada, dan rantai panjang, dan jalankan pengujian C++ sekali di bawah AddressSanitizer atau Valgrind.

12.20 Ringkasan Bab

- Tujuh langkah menyelesaikan soal rekursif pada list atau pohon (Subbab 12.2): sebuah contoh, jawaban untuk struktur kosong, pohon dengan argumen setiap pemanggilan, "akar, jawaban kiri, jawaban kanan", pemasangan kembali subpohon yang dikembalikan, kasus-kasus tepi, dan pengujian di bawah pemeriksa memori.
- Setiap simpul sekali berbiaya $\Theta(n)$, Persamaan (12.1); satu lintasan berbiaya $O(h)$, dengan h antara $\lceil \lg(n+1) \rceil - 1$ dan $n-1$, Persamaan (12.4).
- Rekursi sederhana menggabungkan nilai yang dikembalikan: *depth first* adalah *preorder* (Soal 12.1), sebuah tingkat dicapai dengan mengirim $k-1$ ke bawah (Soal 12.2), jumlah dan tinggi mengikuti Persamaan (12.5) dan Persamaan (12.2) (Soal 12.3); pelipattigaan mempertahankan urutan BST menurut Persamaan (12.6) (Soal 12.4); cermin memerlukan pointer yang disimpan dan bukan lagi pohon pencarian (Soal 12.5).
- Pohon kosong itu seimbang; fungsi bantu yang memanggil `height` berbiaya Persamaan (12.7), dan tinggi dengan penanda -2 memeriksa keseimbangan dalam $\Theta(n)$ (Soal 12.6).
- Penghapusan memiliki tiga kasus dan setiap induk memasang kembali apa yang dikembalikan anaknya (Soal 12.7); induk adalah simpul terakhir sebelum nilai itu pada lintasan pencariannya (Soal 12.8); hasil pencarian diuji sebelum dipakai (Soal 12.9); penelusuran *inorder* dengan `prev` bersama meratakan pohon menjadi *double linked list* terurut di tempat (Soal 12.10).

- *Double linked list* dibalik dengan menukar panah-panah setiap simpul (Soal 12.11); lingkaran Josephus adalah list sirkular dan rekurens Persamaan (12.8) (Soal 12.12); rekursi ambil-atau-lewati mendaftar kombinasi, yang banyaknya dihitung dengan Persamaan (12.9), asalkan keadaan bersama dipulihkan (Soal 12.13).
- Pemeriksaan BST mengirim interval ke bawah, karena Persamaan (12.3) menyangkut semua keturunan (Soal 12.14); leluhur bersama terendah adalah tempat dua lintasan pencarian berpisah (Soal 12.15); queue yang ukurannya dibaca sekali per tingkat mencetak pohon tingkat demi tingkat (Soal 12.16).
- Graf, yang mengizinkan siklus, dibahas di Bab 13; *heap* di Bab 14.

12.21 Soal Latihan

Soal Latihan 12.1 sampai 12.4 meminta kode; penyelesaiannya adalah program-program pada Subbab 12.22. Yang lainnya meminta penalaran dan dijawab dengan kata-kata.

1. **(Daun.)** Tulislah `int countLeaves(Node *root)`, banyaknya simpul yang tidak memiliki anak. Apa yang dikembalikannya untuk pohon contoh dan untuk pohon kosong?
2. **(Pembalikan list secara rekursif.)** Baliklah *single linked list* secara rekursif: `LNode *reverseList(LNode *head)` mengembalikan *head* yang baru. Ujilah pada 1, 2, 3, 4, 5.
3. **(Terkecil ke-k.)** Kembalikan nilai terkecil ke-k dari *binary search tree* ($k = 1$ adalah minimumnya), atau -1 jika pohon itu memiliki kurang dari k simpul, dengan berhenti begitu nilai itu ditemukan. Berikan hasilnya untuk $k = 1, \ldots, 9$ pada pohon contoh.
4. **(Pohon yang sama.)** Tulislah `bool sameTree(Node *a, Node *b)`: benar jika kedua pohon memiliki bentuk yang sama dan nilai-nilai yang sama di tempat yang sama. Bandingkan pohon contoh dengan salinan keduanya, dan dengan pohon yang dibangun dari 6, 3, 10, 1, 5, 7, 8.
5. **(Penelusuran dengan tangan.)** Sisipkan 50, 30, 70, 20, 40, 60, 80, 35 ke dalam *binary search tree* kosong. Gambarlah, dan berikan *preorder*, *inorder*, *postorder*, dan *level order*-nya, tingginya, dan daun-daunnya.
6. **(Penghapusan dengan tangan.)** Pada pohon Soal Latihan 12.5, hapuslah 30, lalu 50, dengan aturan penerus Soal 12.7. Gambarlah pohonnya sesudah setiap penghapusan dan berikan *preorder*-nya.
7. **(Seimbangkah?)** Apakah pohon Soal Latihan 12.5 seimbang tingginya? Satu nilai mana yang, bila disisipkan, akan membuatnya tidak seimbang, dan di simpul mana?
8. **(Urutan terburuk.)** Nilai $1, 2, \ldots, n$ disisipkan dalam urutan menaik. Berapa tinggi pohonnya, dan berapa banyak perbandingan yang dilakukan oleh n penyisipan itu seluruhnya? Berapa biaya versi fungsi bantu Soal 12.6 pada pohon ini, dan mengapa?
9. **(*Double linked list* sirkular.)** Dalam *double linked list* sirkular dengan pointer `head` ke satu simpul, tuliskan penugasan-penugasan pointer yang menyisipkan simpul baru `p` *sebelum* `head` (yaitu di akhir list), dan sebutkan apa yang berubah ketika list-nya kosong.
10. **(Josephus dengan rumus.)** Pakailah Persamaan (12.8) untuk mencari anggota yang tersisa untuk $n = 8$, $k = 3$, dan periksalah jawaban Anda dengan menyimulasikan lingkarannya dengan tangan.
11. **(Kembali dari list ke pohon.)** Soal 12.10 mengubah pohon menjadi *double linked list* terurut. Jelaskan algoritme rekursif yang mengubah larik terurut berisi n nilai menjadi *binary search tree* bertinggi $\lceil \lg(n+1) \rceil - 1$, dan sebutkan waktu jalannya.
12. **(Profitina Laundry.)** Profitina Laundry menyimpan pesanan hari itu dalam *binary search tree* berkunci nomor pesanan, setiap simpul memuat nomor pesanan dan beratnya dalam

kilogram. Dengan fungsi-fungsi bab ini, jelaskan cara (a) mencari sebuah pesanan, (b) mendaftar semua pesanan menurut nomor yang menaik, (c) menghitung berat total hari itu, (d) menghitung banyaknya pesanan yang nomornya terletak antara a dan b tanpa mengunjungi seluruh pohon, dan (e) membuang pesanan yang sudah diambil. Berikan biaya masing-masing.

12.22 Penyelesaian Soal Latihan

Soal Latihan 1 sampai 4. **Kode 12.55**, **Kode 12.56**, dan **Kode 12.57** memuat keempat fungsi itu dan mencetak baris-baris yang sama.

Kode 12.55 `ex12_solutions.cpp`: soal latihan pemrograman Bab 12 dalam C++.

```
// ex12_solutions.cpp - the programming exercises of Chapter 12 (12.1 to 12.4).
#include "tree_kit.hpp"

// Exercise 12.1: the number of leaves (nodes with no children).
int countLeaves(Node *root) {
    if (root == nullptr) return 0;
    if (root->left == nullptr && root->right == nullptr) return 1;
    return countLeaves(root->left) + countLeaves(root->right);
}

// Exercise 12.2: reverse a singly linked list recursively; return the new head.
struct LNode {
    int data;
    LNode *next;
};

LNode *reverseList(LNode *head) {
    if (head == nullptr || head->next == nullptr) return head;
    LNode *newHead = reverseList(head->next);   // reverse everything after head
    head->next->next = head;                    // the old second node points back
    head->next = nullptr;                       // head is now the last node
    return newHead;
}

// Exercise 12.3: the k-th smallest value (k = 1, 2, ...), or -1 if none.
int kthSmallest(Node *root, int &k) {           // k counts down during the walk
    if (root == nullptr) return -1;
    int found = kthSmallest(root->left, k);
    if (found != -1) return found;
    if (--k == 0) return root->data;            // this is the k-th node in order
    return kthSmallest(root->right, k);
}

// Exercise 12.4: same shape and same values in the same places?
bool sameTree(Node *a, Node *b) {
    if (a == nullptr || b == nullptr) return a == b;   // both empty, or not
    return a->data == b->data && sameTree(a->left, b->left) &&
           sameTree(a->right, b->right);
}

int main() {
    Node *root = nullptr, *other = nullptr, *third = nullptr;
    buildSample(&root);
    printf("12.1 leaves of the sample tree: %d, of the empty tree: %d\n",
           countLeaves(root), countLeaves(nullptr));
```

Kode 12.55 (Lanjutan) *ex12_solutions.cpp: soal latihan pemrograman Bab 12 dalam C++.*

```
    LNode *head = nullptr;
    for (int v = 5; v >= 1; v--) head = new LNode{v, head};   // 1 2 3 4 5
    head = reverseList(head);
    printf("12.2 reversed:");
    for (LNode *p = head; p != nullptr; p = p->next) printf(" %d", p->data);
    printf("\n");
    while (head != nullptr) {
        LNode *next = head->next;
        delete head;
        head = next;
    }

    printf("12.3 k-th smallest:");
    for (int k = 1; k <= 9; k++) {
        int left = k;
        printf(" %d", kthSmallest(root, left));
    }
    printf("\n");

    buildSample(&other);
    insertAll(&third, {6, 3, 10, 1, 5, 7, 8});
    printf("12.4 sample vs same inserts: %s, sample vs one node fewer: %s\n",
           sameTree(root, other) ? "same" : "different",
           sameTree(root, third) ? "same" : "different");
    clear(&root);
    clear(&other);
    clear(&third);
    return 0;
}
```

Kode 12.56 `Ex12Solutions.java`: soal latihan pemrograman Bab 12 dalam Java.

```java
// Ex12Solutions.java - the programming exercises of Chapter 12 (12.1 to 12.4).
public class Ex12Solutions extends TreeKit {

    // Exercise 12.1: the number of leaves (nodes with no children).
    static int countLeaves(Node root) {
        if (root == null) return 0;
        if (root.left == null && root.right == null) return 1;
        return countLeaves(root.left) + countLeaves(root.right);
    }

    // Exercise 12.2: reverse a singly linked list recursively; return new head.
    static class LNode {
        int data;
        LNode next;

        LNode(int data, LNode next) {
            this.data = data;
            this.next = next;
        }
    }

    static LNode reverseList(LNode head) {
        if (head == null || head.next == null) return head;
        LNode newHead = reverseList(head.next);   // reverse everything after head
        head.next.next = head;                    // the old second node points back
        head.next = null;                         // head is now the last node
        return newHead;
    }

    // Exercise 12.3: the k-th smallest value (k = 1, 2, ...), or -1 if none.
    static int remaining;                         // counts down during the walk

    static int kthSmallest(Node root) {
        if (root == null) return -1;
        int found = kthSmallest(root.left);
        if (found != -1) return found;
        if (--remaining == 0) return root.data;   // this is the k-th node in order
        return kthSmallest(root.right);
    }
```

Kode 12.56 (Lanjutan) *Ex12Solutions.java: soal latihan pemrograman Bab 12 dalam Java.*

```
    // Exercise 12.4: same shape and same values in the same places?
    static boolean sameTree(Node a, Node b) {
        if (a == null || b == null) return a == b;    // both empty, or not
        return a.data == b.data && sameTree(a.left, b.left)
                && sameTree(a.right, b.right);
    }

    public static void main(String[] args) {
        Node root = buildSample();
        System.out.printf("12.1 leaves of the sample tree: %d, "
                + "of the empty tree: %d%n", countLeaves(root), countLeaves(null));

        LNode head = null;
        for (int v = 5; v >= 1; v--) head = new LNode(v, head);    // 1 2 3 4 5
        head = reverseList(head);
        System.out.print("12.2 reversed:");
        for (LNode p = head; p != null; p = p.next) System.out.print(" " + p.data);
        System.out.println();

        System.out.print("12.3 k-th smallest:");
        for (int k = 1; k <= 9; k++) {
            remaining = k;
            System.out.print(" " + kthSmallest(root));
        }
        System.out.println();

        Node other = buildSample();
        Node third = insertAll(null, 6, 3, 10, 1, 5, 7, 8);
        System.out.printf("12.4 sample vs same inserts: %s, "
                + "sample vs one node fewer: %s%n",
                sameTree(root, other) ? "same" : "different",
                sameTree(root, third) ? "same" : "different");
    }
}
```

Kode 12.57 `ex12_solutions.py`: soal latihan pemrograman Bab 12 dalam Python.

```python
# ex12_solutions.py - the programming exercises of Chapter 12 (12.1 to 12.4).
from tree_kit import *


def count_leaves(root):
    """Exercise 12.1: the number of leaves (nodes with no children)."""
    if root is None:
        return 0
    if root.left is None and root.right is None:
        return 1
    return count_leaves(root.left) + count_leaves(root.right)


class LNode:
    def __init__(self, data, nxt=None):
        self.data = data
        self.next = nxt


def reverse_list(head):
    """Exercise 12.2: reverse a singly linked list recursively; return new head."""
    if head is None or head.next is None:
        return head
    new_head = reverse_list(head.next)       # reverse everything after head
    head.next.next = head                    # the old second node points back
    head.next = None                         # head is now the last node
    return new_head


def kth_smallest(root, k):
    """Exercise 12.3: the k-th smallest value (k = 1, 2, ...), or -1 if none.
    k is a one-element list, so that every call counts down the same number."""
    if root is None:
        return -1
    found = kth_smallest(root.left, k)
    if found != -1:
        return found
    k[0] -= 1
    if k[0] == 0:                            # this is the k-th node in order
        return root.data
    return kth_smallest(root.right, k)
```

Kode 12.57 (Lanjutan) *ex12_solutions.py: soal latihan pemrograman Bab 12 dalam Python.*

```
def same_tree(a, b):
    """Exercise 12.4: same shape and same values in the same places?"""
    if a is None or b is None:
        return a is b                       # both empty, or not
    return (a.data == b.data and same_tree(a.left, b.left)
            and same_tree(a.right, b.right))


def main():
    root = build_sample()
    print(f"12.1 leaves of the sample tree: {count_leaves(root)}, "
          f"of the empty tree: {count_leaves(None)}")

    head = None
    for v in range(5, 0, -1):
        head = LNode(v, head)               # 1 2 3 4 5
    head = reverse_list(head)
    print("12.2 reversed:", end="")
    p = head
    while p is not None:
        print(f" {p.data}", end="")
        p = p.next
    print()

    print("12.3 k-th smallest:", end="")
    for k in range(1, 10):
        print(f" {kth_smallest(root, [k])}", end="")
    print()

    other = build_sample()
    third = insert_all(None, [6, 3, 10, 1, 5, 7, 8])
    word = lambda same: "same" if same else "different"
    print(f"12.4 sample vs same inserts: {word(same_tree(root, other))}, "
          f"sample vs one node fewer: {word(same_tree(root, third))}")


if __name__ == "__main__":
    main()
```

```
12.1 leaves of the sample tree: 3, of the empty tree: 0
12.2 reversed: 5 4 3 2 1
12.3 k-th smallest: 1 3 5 6 7 8 9 10 -1
12.4 sample vs same inserts: same, sample vs one node fewer: different
```

Beberapa kata tentang masing-masing. **(1)** Daun dikenali dari dua anak `nullptr` dan dihitung 1; pohon kosong dihitung 0; simpul lainnya menghitung daun-daun subpohonnya. Pohon contoh memiliki tiga daun, 1, 5, dan 9. **(2)** Baliklah dulu semua yang ada sesudah *head*; simpul kedua yang lama lalu menjadi simpul *terakhir* dari sisa yang sudah dibalik, sehingga `head->next->next = head` menggantungkan *head* lama sesudahnya, dan `head->next = nullptr` mengakhiri list di situ. *Base case*-nya, list berisi nol atau satu simpul, sudah terbalik. List 1 sampai 5 menjadi 5, 4, 3, 2, 1. Kedalaman rekursinya n, sehingga untuk list yang panjang perulangan pada Soal 8.10 lebih aman. **(3)** Penelusuran *inorder* mengunjungi nilai-nilai dalam urutan menaik (Subbab 12.3.1), sehingga simpul ke-k yang dikunjungi memuat nilai terkecil ke-k. Penghitungnya harus dipakai bersama oleh semua pemanggilan (referensi di C++, medan di Java, list satu elemen di Python), dan penelusuran berhenti begitu nilainya ditemukan, dengan biaya $O(h + k)$. Untuk $k = 9$ pohon itu hanya memiliki delapan nilai, sehingga jawabannya -1. **(4)** Dua pohon kosong itu sama, pohon kosong dan pohon tak kosong tidak, dan dua pohon tak kosong sama bila akarnya sama dan subpohon kiri serta kanannya sama. Kedua pohon yang dibangun dari penyisipan yang sama itu sama; membuang penyisipan terakhir, 9, membuatnya berbeda.

Soal Latihan 5. Akarnya 50; 30 dan 70 anak-anaknya; 20 dan 40 anak-anak 30; 60 dan 80 anak-anak 70; 35 adalah anak kiri 40. *Preorder* `50 30 20 40 35 70 60 80`; *inorder* `20 30 35 40 50 60 70 80`; *postorder* `20 35 40 30 60 80 70 50`; *level order* `50 30 70 20 40 60 80 35`. Tingginya 3 (lintasan 50, 30, 40, 35), dan daun-daunnya 20, 35, 60, dan 80.

Soal Latihan 6. Menghapus 30 (dua anak): penerusnya adalah nilai terkecil di subpohon kanannya, 40, 35: ke kiri dari 40 ke 35, yang tidak memiliki anak kiri, sehingga penerusnya 35. Salin 35 ke simpul 30 dan hapus 35 yang lama, sebuah daun. *Preorder* `50 35 20 40 70 60 80`. Menghapus 50 (dua anak): penerusnya 60, simpul paling kiri subpohon kanan. Salin 60 ke akar, hapus 60 yang lama, sebuah daun. *Preorder* `60 35 20 40 70 80`.

Soal Latihan 7. Tinggi subpohon-subpohonnya: simpul 40 memiliki 0 dan -1; simpul 30 memiliki 0 dan 1; simpul 70 memiliki 0 dan 0; akar memiliki 2 dan 1. Setiap selisih paling banyak 1, sehingga pohon itu seimbang tingginya. Menyisipkan 36 (atau nilai apa pun antara 35 dan 40, atau di bawah 35 tetapi di atas 30, seperti 33) menggantungkan sebuah simpul di bawah 35: lalu simpul 40 memiliki tinggi 1 dan -1, selisih 2, dan ia tidak seimbang (demikian pula simpul 30, dengan tinggi 0 dan 2).

Soal Latihan 8. Setiap nilai baru ke kanan di setiap simpul, sehingga pohonnya berupa rantai bertinggi $n-1$, batas atas Persamaan (12.4), dan penyisipan-penyisipan itu melakukan $0+1+\cdots+(n-1)=\frac{n(n-1)}{2}$ perbandingan, $\Theta(n^2)$. Fungsi bantu Soal 12.6 hanya berbiaya $\Theta(n)$ di sini: untuk $n \geq 3$ akarnya sendiri bertinggi -1 dan $n-2$, dan `&&` langsung berhenti.

Soal Latihan 9. Misalkan `last = head->prev` (dalam list sirkular, simpul sebelum *head* adalah yang terakhir). Lalu `p->next = head; p->prev = last; last->next = p; head->prev = p;`, empat penugasan, dua panah di setiap sisi. *Head*-nya tidak berubah. Jika list-nya kosong (`head == nullptr`), tidak ada `last`: simpul baru menjadi seluruh lingkaran, `p->next = p; p->prev = p; head = p;`.

Soal Latihan 10. $J(1)=0$, $J(2)=3 \bmod 2=1$, $J(3)=4 \bmod 3=1$, $J(4)=4 \bmod 4=0$, $J(5)=3 \bmod 5=3$, $J(6)=6 \bmod 6=0$, $J(7)=3 \bmod 7=3$, $J(8)=6 \bmod 8=6$: yang tersisa adalah anggota 7. Menyimulasikan 1 sampai 8 dengan $k=3$: 3, 6, 1, 5, 2, 8, 4 keluar, dalam urutan itu, dan 7 tersisa.

Soal Latihan 11. Ambillah elemen tengah sebagai akar, dan bangunlah subpohon kiri dan kanan secara rekursif dari elemen-elemen sebelum dan sesudahnya; rentang kosong memberikan pohon kosong. Setiap elemen menjadi satu simpul dengan kerja tetap per pemanggilan, sehingga waktunya $\Theta(n)$, seperti pada Persamaan (12.1). Kedua belahan berselisih ukuran paling banyak satu di setiap simpul, sehingga pohonnya seimbang tingginya dengan tinggi sekecil mungkin, $\lceil \lg(n+1) \rceil - 1$. Bersama Soal 12.10, cara ini membangun ulang pohon pencarian mana pun menjadi pohon yang serendah-rendahnya dalam waktu linear.

Soal Latihan 12. (a) Cari menurut nomor pesanan: $O(h)$. (b) Penelusuran *inorder*: $\Theta(n)$. (c) Jumlah pada Soal 12.3 diterapkan pada berat, dalam `double`: $\Theta(n)$. (d) Di simpul bernomor x, ke kiri hanya jika $a < x$, ke kanan hanya jika $x < b$, dan hitung x jika $a \leq x \leq b$; hanya kedua lintasan batas dan simpul-simpul di dalam rentang yang dikunjungi, $O(h+m)$ untuk m pesanan yang terhitung. (e) Penghapusan pada Soal 12.7: $O(h)$. Pesanan biasanya datang dengan nomor yang menaik, yang membangun rantai pada Soal Latihan 8 dan mengubah setiap $O(h)$ menjadi $O(n)$: alasan praktis mengapa `std::map` dan `TreeMap` menjaga pohonnya tetap seimbang [1], [5].

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] Free Software Foundation, "Program Instrumentation Options," in *Using the GNU Compiler Collection (GCC)*, GCC 16.2, Aug. 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://gcc.gnu.org/onlinedocs/

[3] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annual Technical Conf.*, Boston, MA, USA, 2012, pp. 309–318.

[4] Valgrind Developers, *Valgrind User Manual*, release 3.27.1, May 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://valgrind.org/docs/manual/

[5] M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, 4th ed. Boston, MA, USA: Pearson, 2014.

[6] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Python*. Hoboken, NJ, USA: Wiley, 2013.

[7] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011.

[8] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[9] Oracle, "Class ArrayDeque," *Java SE 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/

[10] Python Software Foundation, "Evaluation order," "sys.getrecursionlimit," "collections.deque," and "gc: Garbage Collector interface," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/

[11] G. M. Adelson-Velsky and E. M. Landis, "An algorithm for the organization of information," *Soviet Mathematics Doklady*, vol. 3, pp. 1259–1263, 1962.

[12] T. N. Hibbard, "Some combinatorial properties of certain trees with applications to searching and sorting," *Journal of the ACM*, vol. 9, no. 1, pp. 13–28, 1962.

[13] cppreference.com, "new expression," "delete expression," and "std::queue." Accessed: Sep. 24, 2026. [Online]. Available: https://en.cppreference.com/w/cpp

[14] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1994.

Binary search tree, penelusurannya, penghapusan dengan penerus, dan cara menghitung pemanggilan dengan pohon rekursi dibahas dalam Cormen dkk. [1], dan pohon beserta sejarahnya dalam Knuth [8]; penghapusan dengan menyalin penerus berasal dari Hibbard [12], dan pohon yang seimbang tingginya dari Adelson-Velsky dan Landis [11]. Soal Josephus dan rekurensnya diuraikan lengkap dalam Graham, Knuth, dan Patashnik [14]. Weiss [5], Goodrich dkk. [6], serta Sedgewick dan Wayne [7] menyajikan struktur-struktur yang sama berturut-turut dalam C++, Python, dan Java. Aturan C++ untuk `new` dan `delete` mengikuti cppreference [13], stack dan queue Java mengikuti spesifikasi API-nya [9], dan urutan evaluasi serta batas rekursi Python mengikuti dokumentasinya [10]. Pemeriksa memori dijelaskan dalam manual GCC [2], manual Valgrind [4], dan makalah asli AddressSanitizer [3].

Bab 13
Graf

Di akhir bab ini Anda akan sanggup menyatakan dengan tepat apa itu graf dan memakai kosakatanya tanpa ragu: simpul (*vertex*), sisi (*edge*), berarah dan tak berarah, berbobot, derajat, lintasan, siklus, komponen terhubung, dan graf asiklik berarah; membuktikan dan memakai lema jabat tangan; menyimpan satu graf yang sama sebagai matriks ketetanggaan, list ketetanggaan, dan list sisi, serta memilih di antaranya dengan menghitung memori dan waktu; menelusuri graf secara melebar (*breadth-first*) dengan sebuah *queue*, dan memakai penelusuran itu untuk menemukan jarak dalam jumlah langkah serta rute dengan langkah paling sedikit; menelusurinya secara mendalam (*depth-first*), baik secara rekursif maupun dengan *stack* eksplisit, mencatat waktu penemuan dan waktu selesai, lalu memakainya untuk menemukan siklus; memecah graf menjadi komponen-komponen terhubungnya; menyusun tugas-tugas sebuah alur kerja dalam urutan yang menghormati setiap prasyarat, dengan algoritma Kahn maupun dengan penelusuran mendalam, serta mendeteksi kapan urutan semacam itu tidak ada; dan menemukan rute berbobot terpendek dengan algoritma Dijkstra dan sebuah *priority queue*. Satu graf, yaitu peta kampus perpustakaan Pustaka Ceria beserta perpustakaan-perpustakaan cabangnya, menjadi benang merah seluruh bab, sehingga setiap representasi dan setiap algoritma terlihat bekerja pada gambar yang sama. Setiap prosedur ditelusuri dengan tangan, digambar, dihitung ongkosnya, lalu diprogram dalam C++, Java, dan Python, dan setiap keluaran yang dicetak adalah keluaran sungguhan.

13.1 Dari Mana Kita Melanjutkan

Setiap struktur dalam buku ini sejauh ini menata elemen-elemennya dalam bentuk yang sangat khusus. Larik (Bab 1) dan *linked list* (Bab 7 dan 9) menjajarkannya dalam satu garis: setiap elemen mempunyai paling banyak satu elemen sebelumnya dan paling banyak satu elemen sesudahnya. *Stack* dan *queue* (Bab 5) mempertahankan garis itu tetapi membatasi di mana kita boleh menyentuhnya. Pohon (*tree*, Bab 11) melonggarkan garis itu menjadi hierarki: sebuah elemen boleh mempunyai banyak anak, tetapi tetap hanya mempunyai paling banyak satu orang tua, dan hanya ada tepat satu cara berjalan dari akar ke elemen mana pun.

Dunia tidak selalu serapi itu. Jalan setapak antargedung di sebuah kampus tidak membentuk garis ataupun hierarki; dari perpustakaan pusat Anda dapat berjalan langsung ke kafetaria, atau memutar lewat kantor administrasi dan gedung teknik, dan sebagian jalan setapak menutup menjadi putaran. Pertemanan, rute penerbangan, halaman web yang saling bertaut, tugas-tugas proyek yang harus menunggu tugas lain: semuanya adalah kumpulan benda beserta *hubungan* antara pasangan-pasangan benda, tanpa batasan berapa banyak hubungan yang boleh dimiliki sebuah benda dan tanpa larangan hubungan-hubungan itu membentuk putaran. Struktur data yang menangkap persis hal ini adalah *graf*, dan graf adalah struktur paling umum dalam buku ini. List adalah graf, pohon adalah graf, dan demikian pula hampir segala hal lainnya.

Dua alat dari bab-bab terdahulu memikul sebagian besar beban bab ini. *Queue* dari Bab 5, yang pertama masuk pertama keluar, adalah mesin penelusuran melebar di Subbab 13.6: ia membuat penelusuran menyebar dari titik awalnya dalam cincin-cincin berjarak sama. *Stack* dari Bab 5, yang terakhir masuk pertama keluar, bersama rekursi dari Bab 10, yang sebenarnya adalah *stack*

terselubung (Subbab 5.10 memperlihatkan *call stack* bekerja), adalah mesin penelusuran mendalam di Subbab 13.7. Alat ketiga tiba menjelang akhir: *priority queue*, yang dipratinjau di Subbab 5.15 dan dibangun dengan sungguh-sungguh sebagai *heap* di Bab 14. Subbab 13.10 memakai *priority queue* yang disediakan pustaka setiap bahasa, sehingga algoritma lintasan terpendek Dijkstra dapat dipahami sekarang dan mesinnya dipelajari di bab berikutnya.

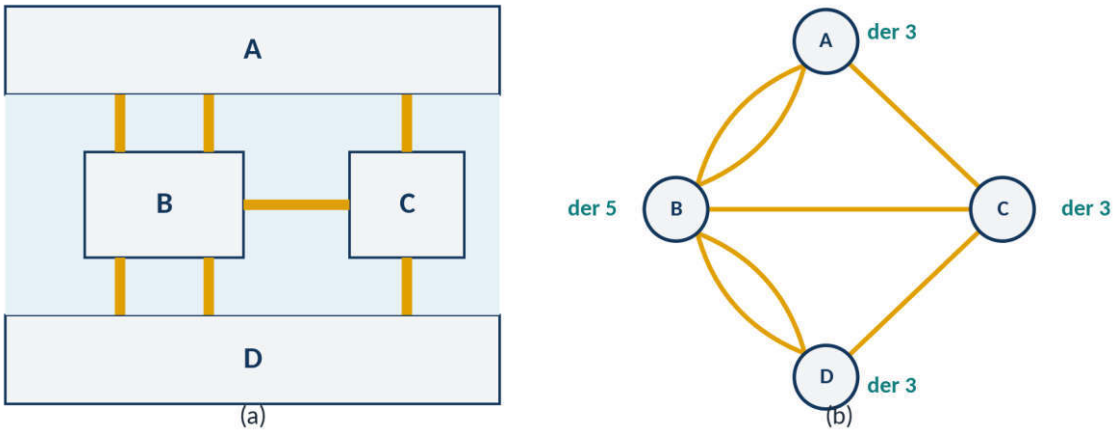
Pertanyaan yang dapat dijawab oleh graf ternyata sangat praktis. Dapatkah saya pergi dari sini ke sana? Rute mana yang paling sedikit berganti jalan? Rute mana yang terpendek dalam meter? Gedung mana yang terputus dari yang lain? Dalam urutan apa sebuah buku baru harus diproses agar tidak ada langkah yang mendahului langkah yang menjadi prasyaratnya? Apakah kumpulan ketergantungan ini mengandung putaran, sehingga tidak ada urutan yang mungkin? Setiap subbab dalam bab ini menjawab salah satu pertanyaan tersebut pada peta kampus Pustaka Ceria, dan Subbab 13.11 memperlihatkan pertanyaan yang sama diajukan, dan dijawab dengan algoritma yang sama, pada jaringan jalan seluruh planet dan pada jejaring sosial berisi lebih dari tiga setengah miliar orang.

13.2 Kisah Pembuka: Tujuh Jembatan dan Sebuah Peta Kampus

13.2.1 Jembatan-jembatan Königsberg

Teori graf mempunyai hari lahir yang luar biasa tepat. Pada abad kedelapan belas, kota Königsberg di Prusia (kini Kaliningrad di Rusia) berdiri di kedua tepi sungai Pregel dan di dua pulau di tengahnya. Tujuh jembatan menghubungkan keempat daratan itu: dua dari tepi utara ke pulau yang lebih besar, dua dari tepi selatan ke pulau yang lebih besar, satu di antara kedua pulau, dan satu dari masing-masing tepi ke pulau yang lebih kecil. Warga kota gemar berjalan-jalan pada Minggu sore, dan sebuah teka-teki pun beredar: adakah jalan-jalan keliling kota yang melintasi setiap jembatan tepat satu kali?

Tak seorang pun dapat menemukannya, dan tak seorang pun dapat membuktikan bahwa jalan semacam itu tidak ada, sampai matematikawan Swiss Leonhard Euler menggarap pertanyaan itu dan menyampaikan jawabannya kepada Akademi Ilmu Pengetahuan di Sankt Peterburg pada tahun 1736; jawaban itu dicetak dalam terbitan Akademi beberapa tahun kemudian [1]. Langkah menentukan Euler adalah membuang segala sesuatu yang tidak penting. Bentuk pulau, panjang jembatan, lebar sungai, jalan-jalan di setiap tepi: tak satu pun memengaruhi jawabannya. Yang penting hanyalah daratan *mana* saja yang ada dan *pasangan* daratan mana yang dihubungkan jembatan, serta berapa kali. **Gambar 13.1** memperlihatkan kota itu disederhanakan dalam dua langkah. Panel (a) adalah peta skematis dengan keempat daratan diberi label A (tepi utara), B (pulau besar), C (pulau kecil), dan D (tepi selatan), serta ketujuh jembatan digambar dengan warna kuning tua. Panel (b) hanya menyimpan apa yang disimpan Euler: satu titik untuk setiap daratan dan satu garis untuk setiap jembatan.



Gambar 13.1 Tujuh jembatan Königsberg: (a) peta skematis dengan empat daratan A sampai D dan tujuh jembatan berwarna kuning tua; (b) keadaan yang sama sebagai graf, dengan banyaknya jembatan yang menyentuh setiap daratan.

Pada panel (b) kita dapat bernalar tentang jalan-jalan tanpa melihat kotanya sama sekali. Misalkan ada jalan-jalan yang melintasi setiap jembatan tepat satu kali. Setiap kali jalan-jalan itu *melewati* sebuah daratan, ia datang melalui satu jembatan dan pergi melalui jembatan lain, sehingga jembatan-jembatan daratan itu terpakai dua demi dua. Satu-satunya tempat sebuah jembatan dapat terpakai "sendirian" adalah daratan tempat jalan-jalan dimulai dan daratan tempat ia berakhir. Jadi setiap daratan selain titik awal dan titik akhir harus menyentuh jembatan dalam jumlah *genap*. Padahal di Königsberg keempat daratan menyentuh jembatan dalam jumlah ganjil: A, C, dan D masing-masing tiga jembatan, dan B lima jembatan. Paling banyak dua di antaranya dapat menjadi titik awal dan titik akhir, sehingga sedikitnya dua daratan tersisa dengan jumlah ganjil yang tidak dapat dipenuhi. Jalan-jalan itu mustahil, dan argumen ini adalah bukti, bukan sekadar kegagalan mencari dengan cukup gigih.

Argumen Euler mengandung, dalam bentuk mini, hampir semua yang dikerjakan bab ini. Ia mengganti keadaan nyata yang ruwet dengan *titik dan hubungan*. Ia menghitung, untuk setiap titik, berapa banyak hubungan yang menyentuhnya, bilangan yang akan kita sebut *derajat*. Dan ia bernalar tentang *jalan* di sepanjang hubungan-hubungan itu. Itulah teori graf, dan 290 tahun kemudian teori ini melukiskan jaringan jalan, jejaring sosial, web, dan ketergantungan antartugas dalam sebuah proyek.

13.2.2 Dari jembatan ke perpustakaan cabang

Perpustakaan Pustaka Ceria, yang telah menemani kita sejak Bab 1, kini telah berkembang. Selain gedung utamanya, ia sekarang mengelola dua perpustakaan cabang di dalam kampus, satu di gedung teknik dan satu di fakultas kedokteran, serta cabang ketiga di pusat kota, di gedung annex kota, bersebelahan dengan gudang tempat buku-buku baru diterima. Buku berpindah antarcabang setiap hari, pembaca berjalan dari satu gedung ke gedung lain, dan para pustakawan terus mengajukan pertanyaan yang terdengar seperti teka-teki Euler: bagaimana mengantar buku dari perpustakaan pusat ke cabang kedokteran, rute mana yang terpendek, gedung mana yang sama sekali tidak dapat dicapai dengan berjalan kaki, dan dalam urutan apa sebuah buku yang baru tiba harus diproses sebelum dapat berdiri di rak?

Untuk menjawabnya, kita melakukan apa yang dilakukan Euler. Kita melupakan bentuk gedung dan lengkungan jalan, dan hanya menyimpan gedung-gedung beserta jalan setapak di antaranya, masing-masing jalan setapak diberi label panjangnya dalam meter. Hasilnya, *peta kampus*, muncul

di Subbab 13.4 dan menjadi contoh berjalan untuk seluruh bab. Namun, lebih dahulu kita memerlukan kosakatanya.

13.3 Kosakata Graf

13.3.1 Simpul dan sisi

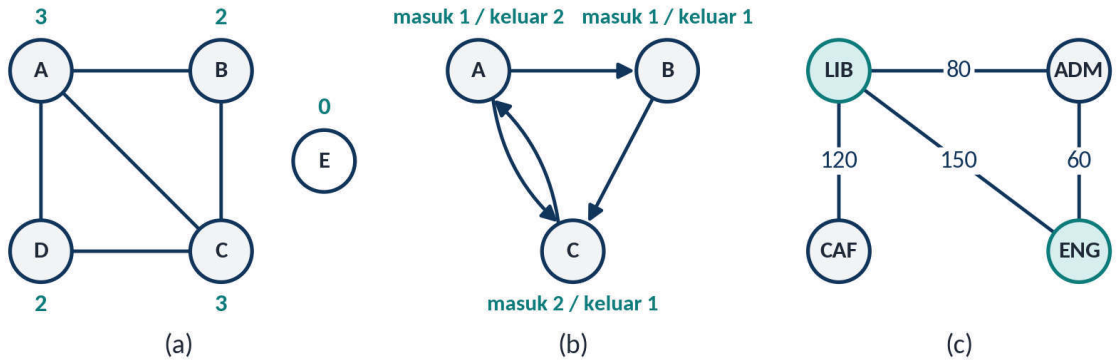
DEFINISI. Sebuah *graf* G terdiri atas himpunan berhingga V yang anggotanya disebut *simpul* (*vertex*) dan himpunan E yang anggotanya disebut *sisi* (*edge*), dengan setiap sisi menghubungkan sepasang simpul. Secara formal, graf ditulis seperti pada Persamaan (13.1).

$$G = (V, E), \qquad E \subseteq \big\{\{u, v\} : u, v \in V,\ u \neq v\big\} \tag{13.1}$$

Dalam kata-kata, Persamaan (13.1) menyatakan: graf adalah pasangan yang terdiri atas himpunan simpul dan himpunan sisi, dan setiap sisi adalah pasangan tak terurut $\{u, v\}$ dari dua simpul yang berbeda. Simpul adalah "benda"-nya (daratan, gedung, orang, halaman web, tugas); sisi adalah hubungan di antara mereka (jembatan, jalan setapak, pertemanan, tautan, prasyarat). Simpul juga disebut *node* atau titik, dan sisi juga disebut *arc*, busur, atau *link*; bab ini memakai *simpul* dan *sisi* secara konsisten, dan menyisakan kata *node* untuk simpul-simpul *linked list*. Banyaknya simpul ditulis $|V|$ dan banyaknya sisi $|E|$; dalam teks dan dalam rumus waktu jalan kita sering menulis V dan E saja untuk bilangan-bilangan ini, seperti yang dilakukan Cormen dkk. [2].

Bila sisi $\{u, v\}$ ada, u dan v disebut *bertetangga* (*adjacent*), dan sisi itu dikatakan *bersisian* (*incident*) dengan keduanya. Sisi-sisi pada Persamaan (13.1) mempunyai dua sifat lain yang patut diberi nama. Pertama, $u \neq v$: sisi dari sebuah simpul ke dirinya sendiri, yang disebut *gelang* (*self-loop*), tidak diperbolehkan. Kedua, E adalah *himpunan*, sehingga pasangan yang sama tidak dapat muncul dua kali: tidak ada *sisi ganda* (*parallel edges*). Graf dengan kedua batasan ini disebut *graf sederhana*. Graf Königsberg pada **Gambar 13.1**(b) *bukan* graf sederhana, karena A dan B dihubungkan oleh dua jembatan; graf seperti itu disebut *multigraf*. Semua graf mulai dari sini adalah graf sederhana, sebagaimana diasumsikan oleh hampir semua program.

CONTOH. Pada **Gambar 13.2**(a), $V = \{A, B, C, D, E\}$ dan $E = \big\{\{A, B\}, \{B, C\}, \{C, D\}, \{D, A\}, \{A, C\}\big\}$, jadi $|V| = 5$ dan $|E| = 5$. A dan B bertetangga; A dan E tidak. Simpul E tidak mempunyai tetangga sama sekali: ia adalah *simpul terasing* (*isolated vertex*), dan ia tetap merupakan simpul graf itu.



Gambar 13.2 Tiga graf kecil: (a) graf tak berarah dengan derajat setiap simpul tertulis di sampingnya, termasuk simpul terasing E; (b) graf berarah dengan derajat-masuk dan derajat-keluar setiap simpul; (c) graf berbobot, sepotong peta kampus dengan panjang jalan setapak dalam meter.

13.3.2 Berarah dan tak berarah

Sisi-sisi pada Persamaan (13.1) tidak mempunyai arah: $\{u, v\}$ dan $\{v, u\}$ adalah himpunan yang sama, sehingga sisi itu dapat dilalui ke dua arah. Graf semacam itu disebut *tak berarah* (*undirected*). Jalan setapak itu tak berarah: bila Anda dapat berjalan dari perpustakaan ke kafetaria, Anda juga dapat berjalan kembali.

Namun, banyak hubungan hanya berlaku satu arah. Jalan satu arah, tautan dari satu halaman web ke halaman lain, "mengikuti" di jejaring sosial, dan "tugas u harus selesai sebelum tugas v dimulai" semuanya mempunyai arah. Dalam *graf berarah* (*directed graph* atau *digraph*), setiap sisi adalah pasangan *terurut* (u, v), digambar sebagai panah dari u ke v; u adalah *pangkal*-nya (*tail*) dan v *ujung*-nya (*head*). Sisi (u, v) tidak mengakibatkan adanya sisi (v, u). Pada **Gambar 13.2**(b) terdapat panah A ke B, A ke C, C ke A, dan B ke C; Anda dapat pergi dari B ke C, tetapi tidak dari C ke B dalam satu langkah.

Graf tak berarah selalu dapat dipandang sebagai graf berarah yang setiap sisinya muncul ke dua arah: jalan setapak $\{u, v\}$ menjadi dua panah (u, v) dan (v, u). Persis beginilah program-program dalam bab ini menyimpan graf tak berarah, dan karena itulah banyak hasil muncul dengan faktor 2.

13.3.3 Graf berbobot

Sering kali sebuah sisi membawa sebuah bilangan: panjang jalan setapak, waktu tempuh sebuah jalan, harga penerbangan, kapasitas sebuah pipa. *Graf berbobot* (*weighted graph*) mempunyai *fungsi bobot* w yang memberi setiap sisi (u, v) sebuah bilangan $w(u, v)$. Pada **Gambar 13.2**(c), $w(\text{LIB,ADM}) = 80$: jalan setapak dari perpustakaan pusat ke kantor administrasi panjangnya 80 meter. Graf *tak berbobot* dapat dipandang sebagai graf berbobot yang semua sisinya berbobot 1, sehingga "panjang" sebuah rute hanyalah banyaknya sisi yang dipakainya. Subbab 13.6 dan 13.10 memperlihatkan bahwa kedua pandangan tentang panjang ini, menghitung sisi dan menjumlahkan bobot, menuntun ke dua algoritma yang berbeda.

13.3.4 Derajat dan lema jabat tangan

Derajat (*degree*) sebuah simpul v dalam graf tak berarah, ditulis $\deg(v)$, adalah banyaknya sisi yang bersisian dengannya, yang dalam graf sederhana sama dengan banyaknya tetangga. Pada **Gambar 13.2**(a), $\deg(A) = 3$ (tetangganya B, C, D), $\deg(B) = 2$, $\deg(C) = 3$, $\deg(D) = 2$, dan $\deg(E) = 0$. Jumlahkan: $3 + 2 + 3 + 2 + 0 = 10$, tepat dua kali banyaknya sisi, yaitu 5. Itu bukan kebetulan.

> **LEMA JABAT TANGAN.** Dalam setiap graf tak berarah, jumlah derajat semua simpul sama dengan dua kali banyaknya sisi:

$$\sum_{v \in V} \deg(v) = 2|E| \tag{13.2}$$

Bukti. Hitunglah pasangan (simpul, sisi yang bersisian dengan simpul itu) dengan dua cara. Dihitung simpul demi simpul, simpul v menyumbang $\deg(v)$ pasangan, sehingga totalnya adalah ruas kiri Persamaan (13.2). Dihitung sisi demi sisi, setiap sisi $\{u, v\}$ menyumbang tepat dua pasangan, satu untuk u dan satu untuk v, sehingga totalnya $2|E|$. Kedua cara menghitung pasangan yang sama, jadi hasilnya sama. ■

Nama lema ini berasal dari sebuah pesta: bila setiap tamu menghitung berapa tangan yang telah dijabatnya, totalnya dua kali banyaknya jabat tangan, karena setiap jabat tangan melibatkan dua

tangan. Dua akibatnya layak diingat. Pertama, jumlah derajat selalu genap, sehingga *banyaknya simpul berderajat ganjil selalu genap*: graf dengan derajat 3, 3, 3, 3, 3 tidak mungkin ada, karena jumlahnya, 15, ganjil. Kedua, lema ini adalah uji gratis bagi program yang membangun graf: jumlahkan panjang semua list ketetanggaan dan bandingkan dengan dua kali banyaknya sisi. Program di Subbab 13.5.5 melakukan persis hal itu.

Dalam graf berarah, satu bilangan tidak cukup. *Derajat-masuk* (*in-degree*) $\deg^-(v)$ menghitung panah yang berakhir di v, dan *derajat-keluar* (*out-degree*) $\deg^+(v)$ menghitung panah yang berawal di v. Pada **Gambar 13.2**(b), A berderajat-masuk 1 (dari C) dan berderajat-keluar 2 (ke B dan C). Setiap panah mempunyai tepat satu pangkal dan satu ujung, sehingga argumen penghitungan ganda yang sama menghasilkan

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E| \tag{13.3}$$

Untuk **Gambar 13.2**(b): derajat-masuk $1 + 1 + 2 = 4$, derajat-keluar $2 + 1 + 1 = 4$, dan panahnya ada empat. Subbab 13.9 membangun satu algoritma utuh, pengurutan topologis Kahn, di atas derajat-masuk.

13.3.5 Lintasan dan siklus

Lintasan (*path*) dari u ke v adalah barisan simpul $v_0, v_1, \ldots, v_k$ dengan $v_0 = u$ dan $v_k = v$ yang setiap dua simpul berurutannya dihubungkan oleh sebuah sisi (dalam graf berarah, oleh panah yang menunjuk ke depan). *Panjang* lintasan adalah banyaknya sisi yang dipakainya, yaitu k. Satu simpul saja adalah lintasan berpanjang 0 dari dirinya ke dirinya. Bila ada lintasan dari u ke v, maka v *dapat dicapai* (*reachable*) dari u. Lintasan disebut *sederhana* bila tidak ada simpul yang muncul dua kali. Dalam graf berbobot kita juga berbicara tentang *bobot* lintasan, yaitu jumlah bobot sisi-sisinya; agar tidak rancu, bab ini menyebut banyaknya sisi sebagai banyaknya *langkah* (*hop*) setiap kali ada bobot.

Siklus (*cycle*) adalah lintasan berpanjang sedikitnya 1 yang berawal dan berakhir di simpul yang sama; dalam graf tak berarah kita juga mensyaratkan sedikitnya tiga simpul berbeda dan tidak ada sisi yang terulang, sehingga berjalan di sepanjang satu sisi lalu langsung kembali tidak dihitung. Pada **Gambar 13.2**(a), A, B, C, A adalah siklus berpanjang 3, dan A, B, C, D, A adalah siklus berpanjang 4. Graf tanpa siklus disebut *asiklik*. Pohon-pohon pada Bab 11 (Subbab 11.2) persis merupakan graf tak berarah yang terhubung dan asiklik, dan karena itulah setiap algoritma pohon dapat berjalan dari akar ke simpul mana pun tanpa pernah bertemu simpul yang sama dua kali. Graf umum tidak memiliki jaminan seperti itu, dan setiap algoritma dalam bab ini harus mengingat simpul mana yang sudah pernah dilihatnya, atau ia akan berputar mengelilingi sebuah siklus selamanya.

13.3.6 Keterhubungan dan komponen terhubung

Graf tak berarah disebut *terhubung* (*connected*) bila setiap simpul dapat dicapai dari setiap simpul lain. **Gambar 13.2**(a) tidak terhubung, karena tidak ada yang mencapai E. Graf yang tak terhubung terpecah menjadi beberapa bagian:

> **DEFINISI.** *Komponen terhubung* (*connected component*) sebuah graf tak berarah adalah himpunan simpul maksimal yang setiap simpulnya dapat dicapai dari setiap simpul lain di dalamnya. “Maksimal” berarti tidak ada simpul di luar himpunan itu yang dapat ditambahkan tanpa merusak sifat tersebut.

Gambar 13.2(a) mempunyai dua komponen, $\{A, B, C, D\}$ dan $\{E\}$. Setiap simpul terletak di tepat satu komponen, karena "dapat dicapai dari" adalah *relasi ekuivalensi* pada simpul-simpul graf tak berarah: setiap simpul mencapai dirinya sendiri (lintasan berpanjang 0), bila u mencapai v maka v mencapai u (telusuri lintasannya mundur), dan bila u mencapai v serta v mencapai x maka u mencapai x (sambung kedua lintasan). Komponen-komponen adalah kelas-kelas ekuivalensinya. Graf terhubung hanyalah graf yang mempunyai tepat satu komponen. Subbab 13.8 menghitung komponen dengan satu penelusuran untuk setiap komponen.

13.3.7 Graf asiklik berarah

Graf berarah yang tidak mempunyai siklus berarah disebut *graf asiklik berarah* (*directed acyclic graph*), yang lazim disingkat *DAG*. DAG memodelkan ketergantungan: panah (u, v) menyatakan "u harus mendahului v", dan sebuah siklus akan menyatakan bahwa sebuah tugas harus mendahului dirinya sendiri, sesuatu yang mustahil. Prasyarat dalam kurikulum perguruan tinggi, urutan kompilasi berkas-berkas sebuah program, langkah-langkah sebuah resep, dan, di Subbab 13.9, langkah-langkah yang mengubah buku yang baru tiba menjadi buku yang berdiri di rak, semuanya adalah DAG. Fakta utama tentang DAG, yang dibuktikan di Subbab 13.9, adalah bahwa simpul-simpulnya selalu dapat didaftar sedemikian rupa sehingga setiap panah menunjuk ke depan dalam daftar itu, dan bahwa graf berarah mempunyai daftar semacam itu hanya bila ia asiklik.

13.3.8 Berapa banyak sisi yang dapat dimiliki graf? Jarang dan padat

Dalam graf tak berarah sederhana, setiap sisi adalah pasangan dua simpul berbeda, dan pasangan semacam itu ada sebanyak $\binom{|V|}{2}$, sehingga

$$|E| \leq \binom{|V|}{2} = \frac{|V|(|V|-1)}{2} \tag{13.4}$$

dan dalam graf berarah sederhana setiap pasangan terurut boleh membawa sebuah panah, sehingga $|E| \leq |V|(|V|-1)$. Graf yang banyak sisinya mendekati maksimum disebut *padat* (*dense*); graf yang sisinya jauh lebih sedikit, biasanya hanya sebuah konstanta kecil kali $|V|$, disebut *jarang* (*sparse*). Satu bilangan yang berguna adalah *kerapatan* (*density*)

$$\rho = \frac{|E|}{|V|(|V|-1)/2} \tag{13.5}$$

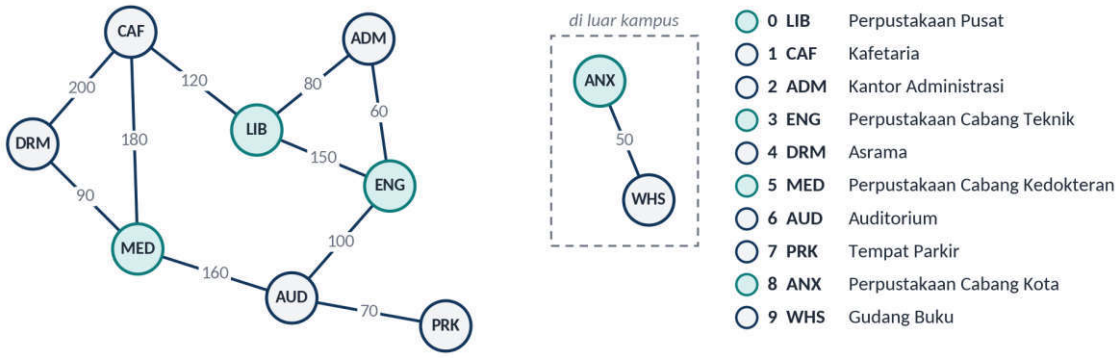
yang bernilai 0 untuk graf tanpa sisi dan 1 untuk *graf lengkap*, yang setiap pasangan simpulnya bertetangga. Pembedaan ini bukan sekadar urusan akademis: Subbab 13.5 memperlihatkan bahwa cara terbaik menyimpan graf bergantung padanya, dan Subbab 13.11 memperlihatkan bahwa graf-graf yang paling penting dalam praktik, jaringan jalan dan jejaring sosial, sangatlah jarang. Dalam jaringan jalan, hampir setiap persimpangan hanya mempunyai paling banyak empat atau lima ruas jalan, berapa pun luas negaranya, sehingga $|E|$ tumbuh seperti $|V|$, bukan seperti $|V|^2$.

> **DENGAN BAHASA SEHARI-HARI.** Graf adalah sekumpulan titik dan sekumpulan garis di antara pasangan-pasangan titik. Garis boleh berpanah (berarah) dan boleh berangka (berbobot). Derajat sebuah titik adalah banyaknya garis yang menyentuhnya, dan seluruh derajat bersama-sama menghitung setiap garis dua kali. Lintasan adalah cara berjalan di

sepanjang garis; siklus adalah jalan yang kembali ke tempat ia berawal. Titik-titik yang dapat saling dicapai membentuk sebuah komponen.

13.4 Contoh Berjalan: Peta Kampus Pustaka Ceria

Gambar 13.3 adalah graf yang digarap oleh seluruh sisa bab ini. Graf ini mempunyai sepuluh simpul, yaitu gedung-gedung jaringan Pustaka Ceria, dinomori 0 sampai 9 sesuai urutan yang dipakai program, masing-masing dengan kode tiga huruf: perpustakaan pusat LIB (0), kafetaria CAF (1), kantor administrasi ADM (2), perpustakaan cabang teknik ENG (3), asrama DRM (4), perpustakaan cabang kedokteran MED (5), auditorium AUD (6), tempat parkir PRK (7), perpustakaan cabang kota ANX (8), dan gudang buku WHS (9). Kode-kodenya berasal dari nama bahasa Inggris gedung-gedung itu, sama seperti yang dicetak program. Keempat perpustakaan digambar berwarna hijau toska. Kesebelas sisinya adalah jalan setapak, masing-masing diberi label panjangnya dalam meter.



Gambar 13.3 Peta kampus Pustaka Ceria, contoh berjalan bab ini: sepuluh gedung (perpustakaan berwarna hijau toska) dan sebelas jalan setapak beserta panjangnya dalam meter. Annex kota dan gudang berada di luar kampus dan belum terhubung dengannya.

Beberapa ciri peta ini sengaja dipilih agar setiap algoritma mempunyai sesuatu untuk diperlihatkan.

- **Graf ini tak berarah dan berbobot.** Jalan setapak dapat dilalui ke dua arah, dan masing-masing mempunyai panjang. Subbab 13.6 sampai 13.8 mengabaikan panjangnya; Subbab 13.10 memakainya.
- **Graf ini mengandung siklus.** LIB, ADM, ENG, LIB adalah segitiga, CAF, DRM, MED, CAF adalah segitiga lain, dan putaran panjang LIB, CAF, MED, AUD, ENG, LIB melingkari bagian tengah. Penelusuran apa pun yang lupa gedung mana yang sudah dikunjunginya akan berputar-putar selamanya.
- **Graf ini tidak terhubung.** Annex kota (ANX) dan gudang (WHS) dihubungkan satu sama lain oleh jalan setapak sepanjang 50 meter, tetapi tidak terhubung ke apa pun di kampus. Peta ini mempunyai dua komponen terhubung, dan Subbab 13.8 menyatukannya dengan sebuah jalan *shuttle*.
- **Langkah paling sedikit dan meter paling sedikit tidak sejalan.** Dari LIB ke PRK, rute LIB, ENG, AUD, PRK memakai 3 jalan setapak dan $150 + 100 + 70 = 320$ meter, sedangkan LIB, ADM, ENG, AUD, PRK memakai 4 jalan setapak tetapi hanya $80 + 60 + 100 + 70 = 310$ meter. Penelusuran melebar (Subbab 13.6) menemukan rute pertama; algoritma Dijkstra (Subbab 13.10) menemukan rute kedua.

Mari terapkan kosakata Subbab 13.3 pada peta ini. $|V| = 10$ dan $|E| = 11$. Derajat-derajatnya adalah $\deg(\text{LIB}) = 3$, $\deg(\text{CAF}) = 3$, $\deg(\text{ADM}) = 2$, $\deg(\text{ENG}) = 3$, $\deg(\text{DRM}) = 2$, $\deg(\text{MED}) =$

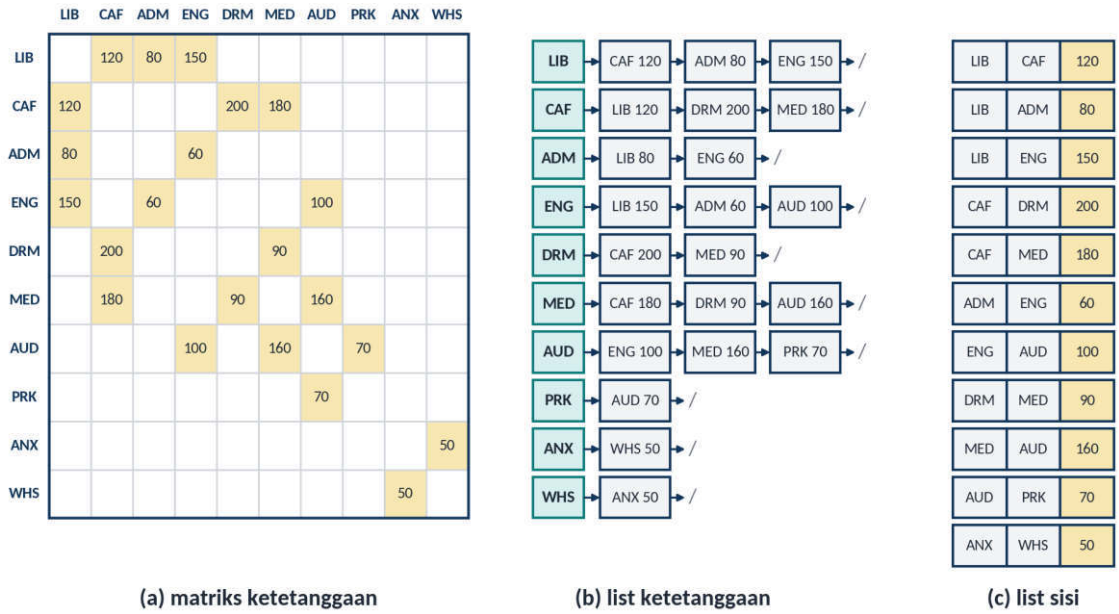
3, deg(AUD) = 3, deg(PRK) = 1, deg(ANX) = 1, dan deg(WHS) = 1. Jumlahnya 22, yaitu 2×11, sesuai tuntutan Persamaan (13.2). Delapan simpul (LIB, CAF, ENG, MED, AUD, PRK, ANX, dan WHS) berderajat ganjil, banyaknya genap, sesuai akibat pertama lema itu. Menurut Persamaan (13.4), graf sederhana dengan 10 simpul mempunyai paling banyak $10 \times 9/2 = 45$ sisi, sehingga menurut Persamaan (13.5) kerapatan peta ini adalah $\rho = 11/45 \approx 0.24$. Peta ini kecil, tetapi tiga perempat jalan setapak yang mungkin sudah tidak ada, dan peta sungguhan jauh lebih jarang lagi.

13.5 Merepresentasikan Graf di Memori

Gambar dibuat untuk manusia. Program memerlukan graf di memori, dalam bentuk yang menjawab pertanyaan-pertanyaannya dengan cepat. Dua pertanyaan yang diajukan setiap algoritma graf, berulang-ulang, adalah:

1. *Adakah sisi dari u ke v, dan bila ada, berapa bobotnya?*
2. *Siapa saja tetangga u?*

Ada tiga representasi klasik, dan masing-masing menjawab kedua pertanyaan itu dengan harga yang berbeda. **Gambar 13.4** memperlihatkan ketiganya untuk peta kampus.



Gambar 13.4 Peta kampus yang disimpan dengan tiga cara: (a) matriks ketetanggaan, dengan panjang jalan setapak pada sel-sel yang diarsir; (b) list ketetanggaan, satu rantai entri (tetangga, panjang) untuk setiap simpul; (c) list sisi, satu rekaman untuk setiap jalan setapak.

13.5.1 Matriks ketetanggaan

Nomorilah simpul-simpul $0,1,\ldots,|V|-1$. *Matriks ketetanggaan* (*adjacency matrix*) adalah tabel `mat` berukuran $|V| \times |V|$ yang sel pada baris u dan kolom v-nya menggambarkan sisi dari u ke v. Untuk graf tak berbobot, sel berisi 1 bila sisi itu ada dan 0 bila tidak; untuk peta kampus, sel berisi panjang jalan setapak, dengan 0 berarti "tidak ada jalan setapak" (pilihan yang aman karena setiap panjang sungguhan bernilai positif). **Gambar 13.4**(a) memperlihatkan matriks itu. Baris LIB mempunyai entri di kolom CAF (120), ADM (80), dan ENG (150), ketiga tetangga perpustakaan pusat, dan tidak ada yang lain.

Tiga sifat dapat dibaca langsung dari gambar.

- **Diagonalnya kosong**, karena graf sederhana tidak mempunyai gelang.
- **Matriksnya simetris**: sel pada baris u, kolom v sama dengan sel pada baris v, kolom u, karena sisi tak berarah disimpan ke dua arah. Graf berarah hanya mengisi sel untuk panahnya, sehingga matriksnya pada umumnya tidak simetris.
- **Sebagian besar sel kosong.** Dari 100 sel, hanya 22 yang diarsir, dua untuk setiap jalan setapak dari 11 jalan setapak yang ada.

Matriks menjawab pertanyaan 1 dalam satu langkah: untuk mengetahui apakah ENG dan AUD terhubung, lihat satu sel, yang memakan waktu $O(1)$, persis seperti mengakses larik dengan indeks (Bab 1). Pertanyaan 2 lebih lambat: untuk mendaftar tetangga u, program harus memindai seluruh baris u, semua $|V|$ sel, sekalipun u hanya mempunyai satu tetangga. Jadi mendaftar tetangga u memakan waktu $\Theta(|V|)$, dan menghitung derajat juga $\Theta(|V|)$. Memorinya $|V|^2$ sel berapa pun banyaknya sisi; untuk graf dengan $|V| = 10^6$ simpul, itu berarti 10^{12} sel, satu terabita bahkan bila satu sel hanya satu bita.

13.5.2 List ketetanggaan

List ketetanggaan (*adjacency list*) menyimpan, untuk setiap simpul u, hanya tetangga yang benar-benar dimiliki u: sebuah larik `adj` berisi $|V|$ list, dengan `adj[u]` memuat satu entri untuk setiap sisi yang keluar dari u. Untuk graf berbobot, setiap entri adalah pasangan (tetangga, bobot). **Gambar 13.4**(b) menggambar setiap list sebagai rantai dengan gaya Bab 7, dan *linked list* memang salah satu cara mengimplementasikannya; program-program bab ini memakai larik yang dapat tumbuh (`std::vector` di C++, `ArrayList` di Java, `list` di Python), yang lebih cepat ditelusuri karena alasan *cache* yang dijelaskan Subbab 7.3. Sisi tak berarah $\{u, v\}$ disimpan dua kali, sekali di `adj[u]` dan sekali di `adj[v]`, sehingga semua list bersama-sama memuat $2|E|$ entri, yang menurut Persamaan (13.2) tepat sama dengan jumlah semua derajat.

Kini harganya terbalik. Pertanyaan 2 semurah-murahnya: tetangga u hanyalah entri-entri `adj[u]`, didaftar dalam waktu $\Theta(\deg(u))$, dan derajatnya adalah panjang list itu, tersedia dalam $O(1)$. Pertanyaan 1 menuntut pemindaian `adj[u]`, yaitu $O(\deg(u))$: murah bila derajatnya kecil, seperti pada peta jalan, dan lambat untuk simpul dengan jutaan tetangga, misalnya seorang selebritas di jejaring sosial. Memorinya satu kepala list per simpul ditambah satu entri per ujung sisi, total $\Theta(|V| + |E|)$.

13.5.3 List sisi

Representasi yang paling sederhana adalah *list sisi* (*edge list*): satu rekaman (u, v, w) untuk setiap sisi, dalam urutan apa saja, seperti pada **Gambar 13.4**(c). Beginilah biasanya graf datang: berkas ruas-ruas jalan, tabel pertemanan, daftar jalan setapak yang diketik seorang pustakawan. Memorinya $\Theta(|E|)$, dengan setiap sisi tak berarah disimpan satu kali saja. Namun, list sisi menjawab kedua pertanyaan dengan lambat: untuk mengetahui apakah u dan v terhubung, atau untuk mendaftar tetangga u, program harus memindai semua $|E|$ rekaman. Algoritma yang mengunjungi sisi-sisi dalam satu lintasan, dalam urutan apa pun datangnya, cocok dengan list sisi; algoritma pohon perentang minimum Kruskal, yang mula-mula mengurutkan semua sisi menurut bobotnya, adalah contoh klasiknya [2]. Penelusuran-penelusuran dalam bab ini tidak cocok dengannya, dan setiap program di sini mengubah list sisi menjadi list ketetanggaan terlebih dahulu, yang memakan waktu $\Theta(|V| + |E|)$.

13.5.4 Ongkos berdampingan

Memori yang diperlukan oleh dua representasi utama sebuah graf tak berarah adalah

$$S_{\text{matriks}} = |V|^2 \text{ sel}, \qquad S_{\text{list}} = |V| + 2|E| \text{ entri} \tag{13.6}$$

Untuk peta kampus, $S_{\text{matriks}} = 100$ sel, $S_{\text{list}} = 10 + 22 = 32$ (sepuluh kepala list dan 22 entri), dan list sisi memerlukan 11 rekaman. Kapan matriks lebih kecil? Dengan menuliskan $|V|^2 < |V| + 2|E|$ dan memakai $|E| \le |V|(|V| - 1)/2$ dari Persamaan (13.4), matriks hanya dapat menang bila $|E|$ mendekati maksimumnya, yaitu untuk graf padat; dan karena satu sel matriks sebesar satu bit lebih kecil daripada satu entri list sebesar 8 bita, titik impas untuk matriks 0/1 murni berada di sekitar kerapatan $1/64$ dari maksimum bila entri list selebar 8 bita. **Tabel 13.1** mengumpulkan semua ongkosnya.

Tabel 13.1 Ongkos ketiga representasi graf, untuk graf dengan V simpul dan E sisi.

Operasi	Matriks ketetanggaan	List ketetanggaan	List sisi
Memori	$\Theta(V^2)$	$\Theta(V + E)$	$\Theta(E)$
Apakah $\{u, v\}$ sebuah sisi?	$O(1)$	$O(\deg(u))$	$O(E)$
Mendaftar tetangga u	$\Theta(V)$	$\Theta(\deg(u))$	$\Theta(E)$
Derajat u	$\Theta(V)$	$O(1)$	$\Theta(E)$
Menambah sisi	$O(1)$	$O(1)$ teramortisasi	$O(1)$ teramortisasi
Menghapus sisi	$O(1)$	$O(\deg(u) + \deg(v))$	$O(E)$
Menambah simpul	$\Theta(V^2)$ (menyalin matriks)	$O(1)$ teramortisasi	$O(1)$
Mengunjungi setiap sisi sekali	$\Theta(V^2)$	$\Theta(V + E)$	$\Theta(E)$
Paling cocok untuk	graf padat, banyak pertanyaan sisi	graf jarang, penelusuran	berkas masukan, algoritma sisi demi sisi

Baris yang menentukan pilihan dalam bab ini adalah "mengunjungi setiap sisi sekali". Penelusuran melebar, penelusuran mendalam, pelabelan komponen, pengurutan topologis, dan algoritma Dijkstra pada dasarnya melakukan satu hal berulang-ulang: mengambil sebuah simpul dan melihat semua tetangganya. Dengan list ketetanggaan, melihat tetangga setiap simpul satu kali berongkos $\sum_u \Theta(1 + \deg(u)) = \Theta(V + E)$ menurut Persamaan (13.2). Dengan matriks, pekerjaan yang sama berongkos $\sum_u \Theta(V) = \Theta(V^2)$, karena setiap baris dipindai seluruhnya, termasuk sel-sel kosongnya. Pada peta kampus itu berarti 32 langkah melawan 100; pada jaringan jalan dengan 24 juta persimpangan (Subbab 13.11), itu adalah perbedaan antara sepersekian detik dan berjam-jam. Karena itulah setiap penelusuran dalam bab ini ditulis untuk list ketetanggaan.

> **PEGANGAN PRAKTIS.** Pakailah list ketetanggaan kecuali bila grafnya padat atau algoritmanya terutama bertanya "adakah sisi antara u dan v?". Pertahankan list sisi bila graf datang sebagai daftar sisi dan algoritmanya memproses sisi satu per satu.

13.5.5 Programnya dalam C++

Kode 13.1 membangun ketiga representasi peta kampus dari daftar sebelas jalan setapak yang sama, mencetaknya, menghitung setiap derajat dua kali (dari baris matriks dan dari panjang list, dan mengeluh bila keduanya berbeda), memeriksa lema jabat tangan, menjawab pertanyaan sisi "adakah jalan setapak ENG-AUD, dan LIB-DRM?" dengan ketiganya, lalu melaporkan memori yang dipakai masing-masing.

Kode 13.1 `graph_repr.cpp`: peta kampus sebagai matriks ketetanggaan, list ketetanggaan, dan list sisi.

```cpp
// graph_repr.cpp -- the Pustaka Ceria campus map stored three ways:
// adjacency matrix, adjacency list and edge list.
#include <cstdio>
#include <vector>

const int V = 10;
const char *CODE[V] = {"LIB", "CAF", "ADM", "ENG", "DRM",
                       "MED", "AUD", "PRK", "ANX", "WHS"};

struct Edge {            // one footpath: endpoints u, v and length in meters
    int u, v, w;
};

// The eleven footpaths of the campus map, in the order they are added.
const std::vector<Edge> PATHS = {
    {0, 1, 120}, {0, 2, 80},  {0, 3, 150}, {1, 4, 200}, {1, 5, 180}, {2, 3, 60},
    {3, 6, 100}, {4, 5, 90},  {5, 6, 160}, {6, 7, 70},  {8, 9, 50}};

struct Nbr {             // one entry of an adjacency list
    int to, w;
};

int main() {
    // 1. Adjacency matrix: mat[u][v] = length of the path u-v, 0 = no path.
    std::vector<std::vector<int>> mat(V, std::vector<int>(V, 0));
    // 2. Adjacency list: adj[u] holds one entry per neighbor of u.
    std::vector<std::vector<Nbr>> adj(V);
    // 3. Edge list: simply the edges themselves.
    std::vector<Edge> edges;

    for (const Edge &e : PATHS) {
        mat[e.u][e.v] = e.w;             // undirected: fill both cells
        mat[e.v][e.u] = e.w;
        adj[e.u].push_back({e.v, e.w});  // undirected: both lists
        adj[e.v].push_back({e.u, e.w});
        edges.push_back(e);              // stored once
    }
```

Kode 13.1 (Lanjutan) *graph_repr.cpp: peta kampus sebagai matriks ketetanggaan, list ketetanggaan, dan list sisi.*

```
    std::printf("Adjacency matrix (meters, . = no footpath)\n     ");
    for (int v = 0; v < V; ++v) std::printf("%5s", CODE[v]);
    std::printf("\n");
    for (int u = 0; u < V; ++u) {
        std::printf("%5s", CODE[u]);
        for (int v = 0; v < V; ++v) {
            if (mat[u][v] == 0) std::printf("%5s", ".");
            else std::printf("%5d", mat[u][v]);
        }
        std::printf("\n");
    }

    std::printf("\nAdjacency list\n");
    for (int u = 0; u < V; ++u) {
        std::printf("  %s:", CODE[u]);
        for (const Nbr &n : adj[u]) std::printf(" %s(%d)", CODE[n.to], n.w);
        std::printf("\n");
    }

    std::printf("\nEdge list");
    for (std::size_t i = 0; i < edges.size(); ++i) {
        if (i % 6 == 0) std::printf("\n ");   // six edges per printed line
        std::printf(" %s-%s(%d)", CODE[edges[i].u], CODE[edges[i].v], edges[i].w);
    }
    std::printf("\n");

    // Degree: count a row of the matrix, or read the size of a list.
    std::printf("\nDegrees:");
    int sum = 0;
    for (int u = 0; u < V; ++u) {
        int dm = 0;
        for (int v = 0; v < V; ++v) if (mat[u][v] != 0) ++dm;   // O(V)
        int dl = static_cast<int>(adj[u].size());                 // O(1)
        if (dm != dl) std::printf(" MISMATCH");
        std::printf(" %s=%d", CODE[u], dl);
        sum += dl;
    }
    std::printf("\nSum of degrees = %d = 2 x %zu edges\n", sum, edges.size());
```

Kode 13.1 (Lanjutan) *graph_repr.cpp: peta kampus sebagai matriks ketetanggaan, list ketetanggaan, dan list sisi.*

```
    // Is there a footpath ENG-AUD? And LIB-DRM?
    int qs[2][2] = {{3, 6}, {0, 4}};
    for (auto &q : qs) {
        int u = q[0], v = q[1];
        bool inMatrix = mat[u][v] != 0;               // one cell: O(1)
        bool inList = false;                          // scan u's list: O(deg u)
        for (const Nbr &n : adj[u]) if (n.to == v) inList = true;
        bool inEdges = false;                         // scan all edges: O(E)
        for (const Edge &e : edges)
            if ((e.u == u && e.v == v) || (e.u == v && e.v == u)) inEdges = true;
        std::printf("Footpath %s-%s? matrix: %s, list: %s, edge list: %s\n",
                    CODE[u], CODE[v], inMatrix ? "yes" : "no",
                    inList ? "yes" : "no", inEdges ? "yes" : "no");
    }

    std::printf("\nStorage: matrix %d cells, list %d entries, "
                "edge list %zu records\n", V * V, sum, edges.size());
    return 0;
}
```

```
Adjacency matrix (meters, . = no footpath)
       LIB  CAF  ADM  ENG  DRM  MED  AUD  PRK  ANX  WHS
  LIB    .  120   80  150    .    .    .    .    .    .
  CAF  120    .    .    .  200  180    .    .    .    .
  ADM   80    .    .   60    .    .    .    .    .    .
  ENG  150    .   60    .    .    .  100    .    .    .
  DRM    .  200    .    .    .   90    .    .    .    .
  MED    .  180    .    .   90    .  160    .    .    .
  AUD    .    .    .  100    .  160    .   70    .    .
  PRK    .    .    .    .    .    .   70    .    .    .
  ANX    .    .    .    .    .    .    .    .    .   50
  WHS    .    .    .    .    .    .    .    .   50    .

Adjacency list
  LIB: CAF(120) ADM(80) ENG(150)
  CAF: LIB(120) DRM(200) MED(180)
  ADM: LIB(80) ENG(60)
  ENG: LIB(150) ADM(60) AUD(100)
  DRM: CAF(200) MED(90)
  MED: CAF(180) DRM(90) AUD(160)
  AUD: ENG(100) MED(160) PRK(70)
  PRK: AUD(70)
  ANX: WHS(50)
  WHS: ANX(50)

Edge list
  LIB-CAF(120) LIB-ADM(80) LIB-ENG(150) CAF-DRM(200) CAF-MED(180) ADM-ENG(60)
  ENG-AUD(100) DRM-MED(90) MED-AUD(160) AUD-PRK(70) ANX-WHS(50)

Degrees: LIB=3 CAF=3 ADM=2 ENG=3 DRM=2 MED=3 AUD=3 PRK=1 ANX=1 WHS=1
Sum of degrees = 22 = 2 x 11 edges
Footpath ENG-AUD? matrix: yes, list: yes, edge list: yes
Footpath LIB-DRM? matrix: no, list: no, edge list: no

Storage: matrix 100 cells, list 22 entries, edge list 11 records
```

Beberapa rincian patut dibahas. Jalan-jalan setapak ditulis sekali saja, sebagai larik `PATHS` berisi rekaman `Edge`, dan semua yang lain diturunkan darinya; beginilah graf biasanya dibangun, karena graf datang sebagai list sisi. Matriksnya adalah `std::vector` berisi `std::vector<int>`, $|V|$ baris berisi $|V|$ nol, sehingga `mat[u][v]` terbaca seperti matematikanya. List ketetanggaan `adj` adalah `std::vector` berisi `std::vector<Nbr>`; `push_back` menambahkan tetangga dalam waktu $O(1)$ teramortisasi, trik penggandaan dari Subbab 5.5.1. Urutan entri dalam setiap list adalah urutan penambahan jalan setapak, dan karena itulah `adj[LIB]` terbaca CAF, ADM, ENG. Urutan itu sembarang, tetapi penting: ia menentukan urutan penelusuran-penelusuran pada subbab-subbab berikutnya mengunjungi tetangga, dan karena itu menentukan keluarannya secara persis.

Keluarannya menegaskan hitungan Subbab 13.4: derajat 3, 3, 2, 3, 2, 3, 3, 1, 1, 1, berjumlah 22, dua kali 11 sisi. Baris terakhir menerjemahkan Persamaan (13.6) menjadi angka: 100 sel, 22 entri list (ditambah sepuluh kepala list), 11 rekaman sisi.

13.5.6 Program yang sama dalam Java dan Python

Kode 13.2 dan **Kode 13.3** membangun ketiga representasi yang sama. Di Java, sisi dan entri tetangga adalah tipe `record`, kelas ringkas yang tak dapat diubah dan tersedia sejak Java 16; di Python, *tuple* biasa memainkan peran yang sama.

Kode 13.2 `GraphRepr.java`: ketiga representasi dalam Java.

```java
// GraphRepr.java -- the Pustaka Ceria campus map stored three ways:
// adjacency matrix, adjacency list and edge list.
import java.util.ArrayList;
import java.util.List;

public class GraphRepr {
    static final int V = 10;
    static final String[] CODE = {"LIB", "CAF", "ADM", "ENG", "DRM",
                                  "MED", "AUD", "PRK", "ANX", "WHS"};

    record Edge(int u, int v, int w) {}     // one footpath, length w meters
    record Nbr(int to, int w) {}            // one entry of an adjacency list

    static final Edge[] PATHS = {
        new Edge(0, 1, 120), new Edge(0, 2, 80), new Edge(0, 3, 150),
        new Edge(1, 4, 200), new Edge(1, 5, 180), new Edge(2, 3, 60),
        new Edge(3, 6, 100), new Edge(4, 5, 90), new Edge(5, 6, 160),
        new Edge(6, 7, 70), new Edge(8, 9, 50)};

    public static void main(String[] args) {
        int[][] mat = new int[V][V];                // 0 = no footpath
        List<List<Nbr>> adj = new ArrayList<>();
        for (int u = 0; u < V; u++) adj.add(new ArrayList<>());
        List<Edge> edges = new ArrayList<>();

        for (Edge e : PATHS) {
            mat[e.u()][e.v()] = e.w();              // undirected: both cells
            mat[e.v()][e.u()] = e.w();
            adj.get(e.u()).add(new Nbr(e.v(), e.w()));   // and both lists
            adj.get(e.v()).add(new Nbr(e.u(), e.w()));
            edges.add(e);                           // stored once
        }

        System.out.print("Adjacency matrix (meters, . = no footpath)\n     ");
        for (int v = 0; v < V; v++) System.out.printf("%5s", CODE[v]);
        System.out.println();
        for (int u = 0; u < V; u++) {
            System.out.printf("%5s", CODE[u]);
            for (int v = 0; v < V; v++) {
                if (mat[u][v] == 0) System.out.printf("%5s", ".");
                else System.out.printf("%5d", mat[u][v]);
            }
            System.out.println();
        }
```

Kode 13.2 (Lanjutan) *GraphRepr.java: ketiga representasi dalam Java.*

```
        System.out.println("\nAdjacency list");
        for (int u = 0; u < V; u++) {
            System.out.print("  " + CODE[u] + ":");
            for (Nbr n : adj.get(u))
                System.out.printf(" %s(%d)", CODE[n.to()], n.w());
            System.out.println();
        }

        System.out.print("\nEdge list");
        for (int i = 0; i < edges.size(); i++) {
            if (i % 6 == 0) System.out.print("\n ");  // six edges per printed line
            Edge e = edges.get(i);
            System.out.printf(" %s-%s(%d)", CODE[e.u()], CODE[e.v()], e.w());
        }
        System.out.println();

        System.out.print("\nDegrees:");
        int sum = 0;
        for (int u = 0; u < V; u++) {
            int dm = 0;
            for (int v = 0; v < V; v++) if (mat[u][v] != 0) dm++;   // O(V)
            int dl = adj.get(u).size();                              // O(1)
            if (dm != dl) System.out.print(" MISMATCH");
            System.out.printf(" %s=%d", CODE[u], dl);
            sum += dl;
        }
        System.out.printf("%nSum of degrees = %d = 2 x %d edges%n",
                sum, edges.size());

        int[][] qs = {{3, 6}, {0, 4}};             // ENG-AUD? LIB-DRM?
        for (int[] q : qs) {
            int u = q[0], v = q[1];
            boolean inMatrix = mat[u][v] != 0;
            boolean inList = false;
            for (Nbr n : adj.get(u)) if (n.to() == v) inList = true;
            boolean inEdges = false;
            for (Edge e : edges)
                if ((e.u() == u && e.v() == v) || (e.u() == v && e.v() == u))
                    inEdges = true;
            System.out.printf(
                    "Footpath %s-%s? matrix: %s, list: %s, edge list: %s%n",
                    CODE[u], CODE[v], inMatrix ? "yes" : "no",
                    inList ? "yes" : "no", inEdges ? "yes" : "no");
        }
```

Kode 13.2 (Lanjutan) *GraphRepr.java: ketiga representasi dalam Java.*

```
        System.out.printf("%nStorage: matrix %d cells, list %d entries, "
                + "edge list %d records%n", V * V, sum, edges.size());
    }
}
```

Kode 13.3 `graph_repr.py`: ketiga representasi dalam Python.

```
# graph_repr.py -- the Pustaka Ceria campus map stored three ways:
# adjacency matrix, adjacency list and edge list.

V = 10
CODE = ["LIB", "CAF", "ADM", "ENG", "DRM", "MED", "AUD", "PRK", "ANX", "WHS"]
# The eleven footpaths (u, v, meters), in the order they are added.
PATHS = [(0, 1, 120), (0, 2, 80), (0, 3, 150), (1, 4, 200), (1, 5, 180),
         (2, 3, 60), (3, 6, 100), (4, 5, 90), (5, 6, 160), (6, 7, 70),
         (8, 9, 50)]


def main():
    mat = [[0] * V for _ in range(V)]        # 0 = no footpath
    adj = [[] for _ in range(V)]             # adj[u] = list of (neighbor, meters)
    edges = []
    for u, v, w in PATHS:
        mat[u][v] = mat[v][u] = w            # undirected: both cells
        adj[u].append((v, w))                # and both lists
        adj[v].append((u, w))
        edges.append((u, v, w))              # stored once

    print("Adjacency matrix (meters, . = no footpath)")
    print("     " + "".join(f"{c:>5}" for c in CODE))
    for u in range(V):
        cells = "".join(f"{'.' if x == 0 else x:>5}" for x in mat[u])
        print(f"{CODE[u]:>5}{cells}")

    print("\nAdjacency list")
    for u in range(V):
        print(f"  {CODE[u]}:" + "".join(f" {CODE[v]}({w})" for v, w in adj[u]))

    print("\nEdge list", end="")
    for i, (u, v, w) in enumerate(edges):
        if i % 6 == 0:
            print("\n ", end="")             # six edges per printed line
        print(f" {CODE[u]}-{CODE[v]}({w})", end="")
    print()
```

Kode 13.3 (Lanjutan) *graph_repr.py: ketiga representasi dalam Python.*

```
    print("\nDegrees:", end="")
    total = 0
    for u in range(V):
        dm = sum(1 for x in mat[u] if x != 0)    # O(V)
        dl = len(adj[u])                         # O(1)
        if dm != dl:
            print(" MISMATCH", end="")
        print(f" {CODE[u]}={dl}", end="")
        total += dl
    print(f"\nSum of degrees = {total} = 2 x {len(edges)} edges")

    for u, v in [(3, 6), (0, 4)]:            # ENG-AUD? LIB-DRM?
        in_matrix = mat[u][v] != 0
        in_list = any(x == v for x, _ in adj[u])
        in_edges = any({a, b} == {u, v} for a, b, _ in edges)
        yn = lambda b: "yes" if b else "no"
        print(f"Footpath {CODE[u]}-{CODE[v]}? matrix: {yn(in_matrix)}, "
              f"list: {yn(in_list)}, edge list: {yn(in_edges)}")

    print(f"\nStorage: matrix {V * V} cells, list {total} entries, "
          f"edge list {len(edges)} records")


if __name__ == "__main__":
    main()
```

Keduanya mencetak keluaran yang persis sama dengan **Kode 13.1**. Satu catatan tentang Python: `[[0] * V for _ in range(V)]` membuat V baris yang terpisah. Bentuk yang tampak lebih pendek, `[[0] * V] * V`, akan membuat *satu* baris dan V referensi kepadanya, sehingga menulis `mat[0][1]` akan tampak mengubah semua baris sekaligus, sebuah jebakan klasik. Satu catatan lagi tentang memori: list berisi list di Python menyimpan satu pointer per sel dan satu objek per bilangan yang berbeda, sehingga matriks Python jauh lebih mahal daripada $|V|^2$ bita. Untuk graf besar di Python, matriks sebaiknya disimpan dalam larik NumPy atau, jauh lebih sering, tidak dibangun sama sekali.

13.6 Penelusuran Melebar (Breadth-First Search)

13.6.1 Gagasannya: riak di kolam

Jatuhkan sebutir batu ke kolam yang tenang. Sebuah cincin riak menyebar keluar, dan setiap titik air dicapai cincin itu pada saat yang sebanding dengan jaraknya dari batu. *Penelusuran melebar* (*breadth-first search*, BFS) menyebar di dalam graf dengan cara yang sama. Mulai dari simpul sumber s, ia lebih dahulu mengunjungi s, kemudian semua tetangga s (simpul-simpul yang berjarak satu sisi), kemudian semua simpul yang berjarak dua sisi, lalu tiga, dan seterusnya, menuntaskan setiap cincin sebelum memulai cincin berikutnya. Cincin-cincin itu disebut *lapisan* (*layer*) atau *aras* (*level*), dan nomor cincin tempat sebuah simpul berada adalah *jarak langkah*-nya (*hop distance*) dari s: banyaknya sisi paling sedikit pada lintasan mana pun dari s kepadanya.

Bagaimana program menuntaskan satu cincin sebelum memulai cincin berikutnya? Dengan *queue* dari Bab 5. Simpul-simpul dimasukkan ke *queue* menurut urutan penemuannya dan dikeluarkan dalam urutan yang sama, pertama masuk pertama keluar. Semua simpul cincin 1 ditemukan ketika sumber sedang diproses, sehingga semuanya masuk *queue* sebelum simpul mana pun dari cincin 2; simpul-simpul cincin 2 ditemukan ketika cincin 1 sedang diproses, sehingga semuanya masuk sebelum simpul mana pun dari cincin 3; dan seterusnya. *Queue* diam-diam menjaga cincin-cincin itu tetap berurutan. Algoritma ini diterbitkan secara terpisah oleh Edward F. Moore pada tahun 1959, yang memakainya untuk menemukan lintasan terpendek di dalam labirin [3], dan oleh C. Y. Lee pada tahun 1961, yang memakainya untuk merutekan kabel pada papan sirkuit [4].

13.6.2 Prosedurnya

BFS menyimpan dua bilangan untuk setiap simpul v: $d[v]$, jarak langkah dari sumber yang ditemukan sejauh ini, dengan -1 (atau ∞) berarti "belum ditemukan", dan $\pi[v]$, *orang tua* (*parent*) v, yaitu simpul yang menemukan v, dengan NIL berarti "tidak ada". **Pseudokode 13.1** memberikan prosedurnya; $G.adj[u]$ adalah list ketetanggaan u dan Q adalah sebuah *queue*.

Pseudokode 13.1 `BFS`: penelusuran melebar dari sumber s, menghitung jarak langkah d dan orang tua π.

```
BFS(G, s):
 1.  for each vertex v in G.V
 2.      v.d ← -1                    // -1 berarti "belum ditemukan"
 3.      v.π ← NIL
 4.  s.d ← 0
 5.  Q ← an empty queue
 6.  ENQUEUE(Q, s)
 7.  while Q is not empty
 8.      u ← DEQUEUE(Q)
 9.      for each v in G.adj[u]
10.          if v.d = -1             // v terlihat untuk pertama kali
11.              v.d ← u.d + 1
12.              v.π ← u
13.              ENQUEUE(Q, v)
```

Tiga hal tentang **Pseudokode 13.1** mudah terlewat.

- **Simpul ditandai ketika masuk *queue*, bukan ketika keluar.** Baris 11 mengisi $v.d$ saat v pertama kali terlihat, dan baris 10 menolak simpul mana pun yang d-nya sudah terisi. Jadi setiap simpul masuk *queue* paling banyak satu kali. Bila penanda baru dipasang di baris 8, ketika simpul dikeluarkan, simpul yang mempunyai beberapa tetangga yang sudah dikunjungi dapat dimasukkan beberapa kali; pada peta kampus, AUD akan dimasukkan dari ENG dan juga dari MED. Jawabannya tetap benar, tetapi *queue* dapat memuat sampai $|E|$ entri alih-alih $|V|$, dan pekerjaannya pun membengkak.
- $d[v]$ **sekaligus menjadi penanda "sudah dikunjungi".** Larik Boolean terpisah akan bekerja sama baiknya; memakai d menghemat memori dan menjaga dua fakta, "sudah terlihat" dan "seberapa jauh", agar tidak pernah bertentangan.
- **Orang tua membentuk sebuah pohon.** Setiap simpul yang ditemukan selain s mendapat tepat satu orang tua, yaitu simpul yang menemukannya, dan orang tua itu ditemukan lebih dahulu. Karena itu, mengikuti orang tua dari simpul mana pun akan mendaki, satu sisi demi satu sisi, kembali ke s tanpa pernah mengulang simpul. Sisi-sisi $\{\pi[v], v\}$ membentuk pohon berakar di s, yaitu *pohon BFS*, dan lintasan di pohon ini dari s ke v adalah lintasan terpendek dalam jumlah langkah (Subbab 13.6.4 membuktikannya).

13.6.3 Menelusuri BFS dari perpustakaan pusat

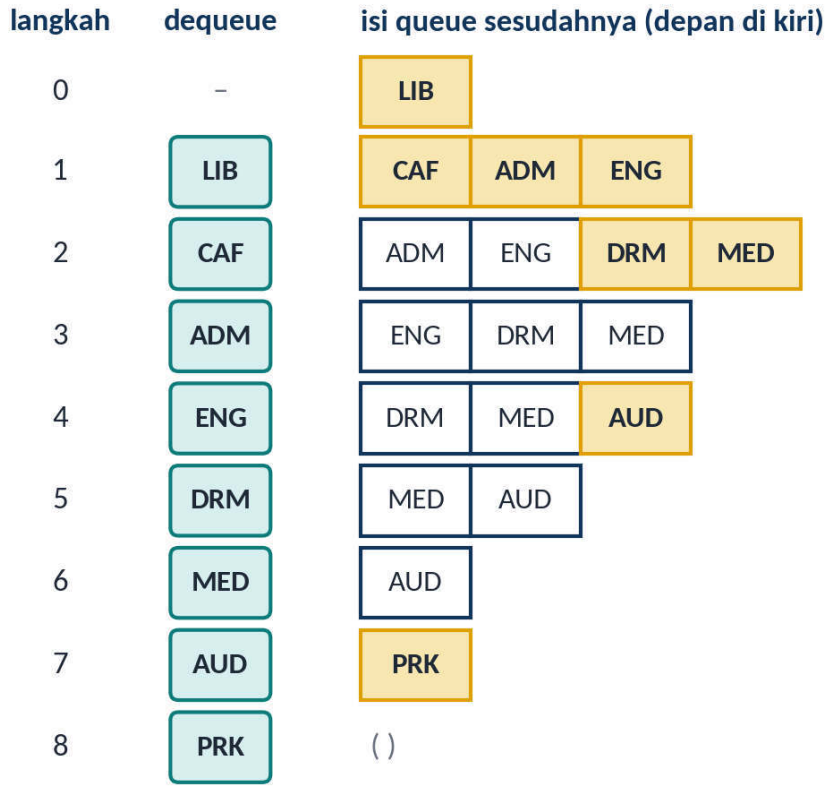
Mari jalankan **Pseudokode 13.1** pada peta kampus dengan sumber LIB, melihat tetangga menurut urutan list ketetanggaan pada **Gambar 13.4**(b). **Tabel 13.2** mencatat setiap langkah: simpul mana yang keluar dari *queue*, tetangga mana yang ditemukan (beserta jarak dan orang tua barunya), dan apa isi *queue* sesudahnya.

Tabel 13.2 Penelusuran melebar dari LIB pada peta kampus, langkah demi langkah.

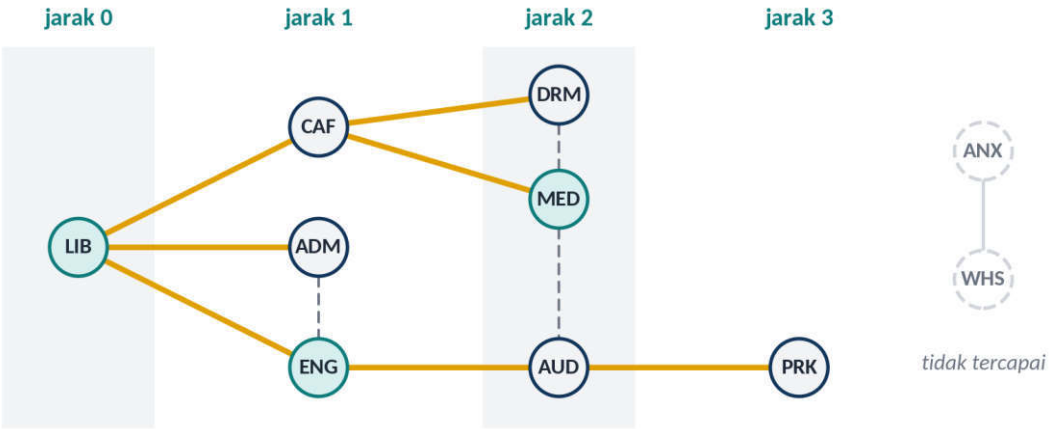
Langkah	Dikeluarkan u	$d[u]$	Tetangga u menurut urutan list	Baru ditemukan	*Queue* sesudahnya (depan lebih dulu)
0				LIB ($d = 0$)	LIB
1	LIB	0	CAF, ADM, ENG	CAF, ADM, ENG ($d = 1$, orang tua LIB)	CAF ADM ENG
2	CAF	1	LIB, DRM, MED	DRM, MED ($d = 2$, orang tua CAF)	ADM ENG DRM MED
3	ADM	1	LIB, ENG	tidak ada (keduanya sudah terlihat)	ENG DRM MED
4	ENG	1	LIB, ADM, AUD	AUD ($d = 2$, orang tua ENG)	DRM MED AUD
5	DRM	2	CAF, MED	tidak ada	MED AUD
6	MED	2	CAF, DRM, AUD	tidak ada	AUD
7	AUD	2	ENG, MED, PRK	PRK ($d = 3$, orang tua AUD)	PRK
8	PRK	3	AUD	tidak ada	(kosong)

Bacalah langkah-langkahnya perlahan. Pada langkah 1 ketiga tetangga LIB ditemukan bersama-sama dan masuk *queue* menurut urutan list. Pada langkah 2 CAF dikeluarkan; tetangganya LIB sudah ditemukan dan diabaikan, sedangkan DRM dan MED baru dan bergabung di *belakang queue*, di belakang ADM dan ENG yang masih menunggu. Itulah *queue* yang menjaga cincin-cincin tetap terpisah: ADM dan ENG (cincin 1) akan diproses sebelum DRM dan MED (cincin 2), walaupun DRM dan MED ditemukan lebih dulu. Pada langkah 3, tetangga ADM, yaitu LIB dan ENG, sudah dikenal semuanya, sehingga tidak terjadi apa-apa; perhatikan bahwa ENG sedang *berada di dalam queue*, belum diproses, dan tetap ditolak dengan benar karena ia sudah ditandai ketika masuk. Langkah 6 adalah penolakan yang menarik: MED melihat AUD, yang ditemukan ENG satu langkah sebelumnya, dan membiarkannya. *Queue* kosong sesudah langkah 8, dan penelusuran berhenti.

Gambar 13.5 memperlihatkan isi *queue* sesudah setiap langkah sebagai deretan sel, dengan simpul-simpul yang baru masuk berwarna kuning tua. **Gambar 13.6** menggambar ulang peta menurut cincin: LIB pada jarak 0; CAF, ADM, dan ENG pada jarak 1; DRM, MED, dan AUD pada jarak 2; PRK pada jarak 3. Sisi-sisi kuning tua adalah pohon BFS, satu sisi dari setiap simpul ke orang tuanya; sisi-sisi abu-abu putus-putus adalah jalan setapak yang dilihat BFS lalu ditolak karena kedua ujungnya sudah ditemukan. Dua simpul, ANX dan WHS, tidak pernah masuk *queue* sama sekali: tidak ada lintasan dari LIB ke sana, sehingga jaraknya tetap -1 (digambar putus-putus di sebelah kanan).



Gambar 13.5 Isi *queue* BFS sesudah setiap langkah **Tabel 13.2**: simpul yang dikeluarkan di kiri, isi *queue* di kanan dengan simpul yang baru dimasukkan berwarna kuning tua.



Gambar 13.6 Peta kampus yang disusun menurut jarak langkah dari LIB. Sisi kuning tua membentuk pohon BFS; sisi putus-putus sudah diperiksa tetapi menuju simpul yang telah ditemukan; ANX dan WHS tidak tercapai.

Sisi-sisi putus-putus pada **Gambar 13.6** sangat berarti. ADM-ENG dan DRM-MED menghubungkan dua simpul dari cincin yang *sama*, dan MED-AUD menghubungkan dua simpul cincin 2. Tidak ada sisi yang melompati cincin: tidak ada sisi yang menghubungkan cincin 0 dengan cincin 2, atau cincin 1 dengan cincin 3. Itu bukan kebetulan peta ini, melainkan hukum BFS, dan hukum itulah inti bukti berikut.

13.6.4 Mengapa BFS menemukan rute dengan langkah paling sedikit

Tulislah $\delta(s, v)$ untuk jarak langkah yang sesungguhnya, yaitu banyaknya sisi paling sedikit pada lintasan mana pun dari s ke v (tak hingga bila tidak ada). Kita ingin menunjukkan bahwa BFS

menghitung $d[v] = \delta(s, v)$ untuk setiap v. Argumennya terdiri atas tiga langkah pendek; Cormen dkk. memberikan versi yang sepenuhnya formal [2].

Langkah 1: $d[v]$ tidak pernah terlalu kecil. Setiap kali baris 11 mengisi $v.d \leftarrow u.d + 1$, rantai orang tua v adalah lintasan sungguhan dari s yang panjangnya tepat $v.d$ sisi. Ada lintasan sungguhan sepanjang $d[v]$ sisi, sehingga lintasan terpendek paling banyak sepanjang itu: $\delta(s, v) \leq d[v]$.

Langkah 2: *queue* selalu terurut menurut d, dengan paling banyak dua nilai yang bersebelahan. Pada setiap saat, bila *queue* memuat $v_1, v_2, \ldots, v_r$ dari depan ke belakang, maka $d[v_1] \leq d[v_2] \leq \cdots \leq d[v_r] \leq d[v_1] + 1$. Sifat ini berlaku di awal (satu simpul). Pengeluaran membuang elemen terdepan dan tidak dapat merusaknya. Pemasukan menambahkan simpul dengan $d = d[u] + 1$, dengan u adalah elemen terdepan yang lama, sehingga $d[u]$ paling besar sama dengan setiap nilai di *queue*, dan nilai baru itu sedikitnya sama dengan setiap nilai di *queue* serta paling besar $d[v_2] + 1$. Pada penelusuran di atas hal ini tampak di kolom terakhir **Tabel 13.2**: *queue* selalu memuat satu cincin, atau ekor sebuah cincin diikuti kepala cincin berikutnya. Akibatnya, simpul-simpul dikeluarkan dalam urutan d yang tidak menurun.

Langkah 3: $d[v]$ tidak pernah terlalu besar. Misalkan ada simpul yang dapat dicapai tetapi mendapat $d[v] > \delta(s, v)$, dan ambillah v semacam itu yang $\delta(s, v)$-nya terkecil. Misalkan u adalah simpul tepat sebelum v pada sebuah lintasan terpendek, sehingga $\delta(s, u) = \delta(s, v) - 1$, dan karena cara v dipilih, $d[u] = \delta(s, u)$. Ketika u dikeluarkan, BFS memeriksa sisi ke v. Bila v belum ditemukan, ia mendapat $d[v] = d[u] + 1 = \delta(s, v)$, kontradiksi. Bila v sudah ditemukan dari suatu x yang lebih awal, maka menurut Langkah 2 x dikeluarkan tidak lebih lambat daripada u, sehingga $d[x] \leq d[u]$, dan $d[v] = d[x] + 1 \leq d[u] + 1 = \delta(s, v)$, lagi-lagi kontradiksi. Jadi v semacam itu tidak ada. Karena itu, setiap sisi (u, v) dalam graf dengan u dapat dicapai memenuhi

$$d[v] \leq d[u] + 1 \tag{13.7}$$

yang tidak lain adalah hukum "tidak ada sisi yang melompati cincin" yang kita lihat pada **Gambar 13.6**. Langkah 1 dan 3 bersama-sama memberikan $d[v] = \delta(s, v)$, dan menelusuri orang tua dari v kembali ke s menghasilkan lintasan dengan tepat sebanyak itu sisi, yaitu lintasan terpendek.

> **DENGAN BAHASA SEHARI-HARI.** *Queue* membuat BFS menuntaskan setiap cincin sebelum memulai cincin berikutnya, sehingga saat pertama sebuah simpul tercapai adalah melalui rute dengan sisi sesedikit mungkin. Ingatlah siapa yang pertama kali mencapai setiap simpul, maka Anda dapat membaca rutenya secara mundur.

13.6.5 Waktu jalannya

Hitunglah kerja **Pseudokode 13.1** pada graf yang disimpan sebagai list ketetanggaan. Baris 1 sampai 3 menyentuh setiap simpul satu kali: $\Theta(V)$. Setiap simpul dimasukkan ke *queue* paling banyak satu kali (karena ditandai saat masuk) dan dengan demikian dikeluarkan paling banyak satu kali, dan setiap operasi *queue* berongkos $O(1)$ (Subbab 5.13): total $O(V)$. Ketika simpul u dikeluarkan, gelung baris 9 berjalan $\deg(u)$ kali dalam graf tak berarah (atau $\deg^+(u)$ kali dalam graf berarah), masing-masing $O(1)$. Dijumlahkan atas semua simpul, menurut Persamaan (13.2),

$$T_{\text{BFS}} = \Theta(V) + \sum_{u \in V} O\left(1 + \deg(u)\right) = O(V + E) \tag{13.8}$$

Ini disebut *waktu linear* untuk graf: sebanding dengan ukuran masukan, yaitu simpul ditambah sisi, dan tidak ada algoritma yang memeriksa seluruh graf dapat berbuat lebih baik. Dengan matriks ketetanggaan, baris 9 akan memindai seluruh baris berisi $|V|$ sel untuk setiap simpul yang dikeluarkan, dan BFS akan berongkos $\Theta(V^2)$, dan karena itulah **Tabel 13.1** menganjurkan list. Memori tambahannya adalah larik d dan π serta *queue* berisi paling banyak $|V|$ simpul: $\Theta(V)$.

13.6.6 Programnya dalam C++

Kode 13.4 mengimplementasikan **Pseudokode 13.1** dengan `std::queue`, adaptor *queue* dari Subbab 5.16.1. Fungsi `bfs` mengisi `dist` dan `parent` lalu mengembalikan urutan kunjungan; bila `trace` dinyalakan, ia mencetak satu baris untuk setiap simpul yang dikeluarkan, sehingga mereproduksi **Tabel 13.2**. Fungsi `printPath` merekonstruksi rute dengan menelusuri orang tua dari simpul tujuan kembali ke sumber dan mendorong setiap simpul ke *depan* sebuah `std::deque`, sehingga rutenya keluar dalam arah maju.

Kode 13.4 `bfs.cpp`: penelusuran melebar pada peta kampus, dengan jejak langkah, jarak langkah, orang tua, dan rute dengan langkah paling sedikit.

```cpp
// bfs.cpp -- breadth-first search on the Pustaka Ceria campus map:
// visiting order, hop distances, parents, and fewest-hop routes.
#include <cstdio>
#include <deque>
#include <queue>
#include <string>
#include <vector>

const int V = 10;
const char *CODE[V] = {"LIB", "CAF", "ADM", "ENG", "DRM",
                       "MED", "AUD", "PRK", "ANX", "WHS"};
const int PATHS[][3] = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                        {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                        {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};

std::vector<std::vector<int>> buildCampus() {   // weights are not needed by BFS
    std::vector<std::vector<int>> adj(V);
    for (const auto &p : PATHS) {
        adj[p[0]].push_back(p[1]);
        adj[p[1]].push_back(p[0]);
    }
    return adj;
}

std::string show(const std::queue<int> &q) {    // print a copy, front first
    std::queue<int> c = q;
    std::string s = "[";
    while (!c.empty()) {
        s += CODE[c.front()];
        c.pop();
        if (!c.empty()) s += " ";
    }
    return s + "]";
}

// Fills dist (-1 = not reached) and parent (-1 = none); returns visiting order.
std::vector<int> bfs(const std::vector<std::vector<int>> &adj, int s,
                     std::vector<int> &dist, std::vector<int> &parent, bool trace) {
    dist.assign(V, -1);
    parent.assign(V, -1);
    std::vector<int> order;
    std::queue<int> q;
    dist[s] = 0;                // mark s when it ENTERS the queue
    q.push(s);
    int step = 0;
    while (!q.empty()) {
```

Kode 13.4 (Lanjutan) *bfs.cpp: penelusuran melebar pada peta kampus, dengan jejak langkah, jarak langkah, orang tua, dan rute dengan langkah paling sedikit.*

```
        int u = q.front();
        q.pop();
        order.push_back(u);
        std::string added;
        for (int v : adj[u]) {
            if (dist[v] == -1) { // first time we see v
                dist[v] = dist[u] + 1;
                parent[v] = u;
                q.push(v);
                added += std::string(" ") + CODE[v];
            }
        }
        if (trace)
            std::printf("%4d  %s(d=%d)  enqueued:%-12s queue now %s\n", ++step,
                        CODE[u], dist[u], added.empty() ? " -" : added.c_str(),
                        show(q).c_str());
    }
    return order;
}

void printPath(const std::vector<int> &parent, const std::vector<int> &dist,
               int t) {
    if (dist[t] == -1) {
        std::printf("  to %s: no route\n", CODE[t]);
        return;
    }
    std::deque<int> path;        // walk the parents back from t, then read forward
    for (int v = t; v != -1; v = parent[v]) path.push_front(v);
    std::printf("  to %s (%d hops):", CODE[t], dist[t]);
    for (int v : path) std::printf(" %s", CODE[v]);
    std::printf("\n");
}

int main() {
    auto adj = buildCampus();
    std::vector<int> dist, parent;
    std::printf("BFS from LIB\nstep  dequeued\n");
    std::vector<int> order = bfs(adj, 0, dist, parent, true);
```

Kode 13.4 (Lanjutan) *bfs.cpp: penelusuran melebar pada peta kampus, dengan jejak langkah, jarak langkah, orang tua, dan rute dengan langkah paling sedikit.*

```
    std::printf("\nVisiting order:");
    for (int v : order) std::printf(" %s", CODE[v]);
    std::printf("\n\nvertex  dist  parent\n");
    for (int v = 0; v < V; ++v) {
        if (dist[v] == -1) std::printf("  %s     -     -\n", CODE[v]);
        else std::printf("  %s  %4d  %5s\n", CODE[v], dist[v],
                         parent[v] == -1 ? "-" : CODE[parent[v]]);
    }
    std::printf("\nFewest-hop routes from LIB\n");
    printPath(parent, dist, 7);  // PRK
    printPath(parent, dist, 4);  // DRM
    printPath(parent, dist, 9);  // WHS
    return 0;
}
```

```
BFS from LIB
step  dequeued
   1  LIB(d=0)  enqueued: CAF ADM ENG queue now [CAF ADM ENG]
   2  CAF(d=1)  enqueued: DRM MED     queue now [ADM ENG DRM MED]
   3  ADM(d=1)  enqueued: -           queue now [ENG DRM MED]
   4  ENG(d=1)  enqueued: AUD         queue now [DRM MED AUD]
   5  DRM(d=2)  enqueued: -           queue now [MED AUD]
   6  MED(d=2)  enqueued: -           queue now [AUD]
   7  AUD(d=2)  enqueued: PRK         queue now [PRK]
   8  PRK(d=3)  enqueued: -           queue now []

Visiting order: LIB CAF ADM ENG DRM MED AUD PRK

vertex  dist  parent
  LIB     0      -
  CAF     1    LIB
  ADM     1    LIB
  ENG     1    LIB
  DRM     2    CAF
  MED     2    CAF
  AUD     2    ENG
  PRK     3    AUD
  ANX     -     -
  WHS     -     -

Fewest-hop routes from LIB
  to PRK (3 hops): LIB ENG AUD PRK
  to DRM (2 hops): LIB CAF DRM
  to WHS: no route
```

Jejak langkahnya adalah **Tabel 13.2**, baris demi baris, dan tabel jarak serta orang tuanya adalah **Gambar 13.6** dalam bentuk angka. Rute ke PRK, LIB, ENG, AUD, PRK, terdiri atas 3 langkah, sesedikit mungkin; Subbab 13.10 akan menemukan rute ke PRK yang 10 meter lebih pendek tetapi memakai 4 langkah. Rute ke WHS tidak ada, dan program mengatakannya alih-alih mencetak hal yang tak bermakna: `dist[WHS]` masih bernilai -1.

Fungsi `show` menyalin *queue* sebelum mencetaknya, karena `std::queue` tidak menyediakan cara untuk melihat elemen apa pun selain elemen terdepan; mencetak berarti menghabiskan *queue*, sehingga yang dicetak adalah salinannya. Inilah harga yang jujur dari memakai adaptor yang hanya menjanjikan perilaku *queue*.

13.6.7 Program yang sama dalam Java dan Python

Di Java, `ArrayDeque` berfungsi sebagai *queue*, dengan `addLast` untuk memasukkan dan `pollFirst` untuk mengeluarkan (Subbab 5.16.2), seperti diperlihatkan **Kode 13.5**; di Python, `collections.deque` dengan `append` dan `popleft` (Subbab 5.16.3). `list` Python dengan `pop(0)` juga dapat bekerja, tetapi setiap `pop(0)` menggeser seluruh list dan berongkos $\Theta(n)$, yang akan mengubah BFS berwaktu linear pada Persamaan (13.8) menjadi kuadratik.

Kode 13.5 `Bfs.java`: penelusuran melebar dalam Java, dengan `ArrayDeque` sebagai *queue*.

```java
// Bfs.java -- breadth-first search on the Pustaka Ceria campus map:
// visiting order, hop distances, parents, and fewest-hop routes.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Bfs {
    static final int V = 10;
    static final String[] CODE = {"LIB", "CAF", "ADM", "ENG", "DRM",
                                  "MED", "AUD", "PRK", "ANX", "WHS"};
    static final int[][] PATHS = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                                  {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                                  {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};

    static List<List<Integer>> buildCampus() {
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
        for (int[] p : PATHS) {
            adj.get(p[0]).add(p[1]);
            adj.get(p[1]).add(p[0]);
        }
        return adj;
    }

    static String show(ArrayDeque<Integer> q) {       // front first
        StringBuilder sb = new StringBuilder("[");
        for (int v : q) {
            if (sb.length() > 1) sb.append(' ');
            sb.append(CODE[v]);
        }
        return sb.append(']').toString();
    }

    static List<Integer> bfs(List<List<Integer>> adj, int s, int[] dist,
                             int[] parent, boolean trace) {
        Arrays.fill(dist, -1);
        Arrays.fill(parent, -1);
        List<Integer> order = new ArrayList<>();
        ArrayDeque<Integer> q = new ArrayDeque<>();
        dist[s] = 0;                          // mark s when it ENTERS the queue
        q.addLast(s);
        int step = 0;
        while (!q.isEmpty()) {
            int u = q.pollFirst();
            order.add(u);
```

Kode 13.5 (Lanjutan) *Bfs.java: penelusuran melebar dalam Java, dengan ArrayDeque sebagai queue.*

```
            StringBuilder added = new StringBuilder();
            for (int v : adj.get(u)) {
                if (dist[v] == -1) {        // first time we see v
                    dist[v] = dist[u] + 1;
                    parent[v] = u;
                    q.addLast(v);
                    added.append(' ').append(CODE[v]);
                }
            }
            if (trace)
                System.out.printf("%4d  %s(d=%d)  enqueued:%-12s queue now %s%n",
                        ++step, CODE[u], dist[u],
                        added.length() == 0 ? " -" : added, show(q));
        }
        return order;
    }

    static void printPath(int[] parent, int[] dist, int t) {
        if (dist[t] == -1) {
            System.out.printf("  to %s: no route%n", CODE[t]);
            return;
        }
        ArrayDeque<Integer> path = new ArrayDeque<>();
        for (int v = t; v != -1; v = parent[v]) path.addFirst(v);
        System.out.printf("  to %s (%d hops):", CODE[t], dist[t]);
        for (int v : path) System.out.print(" " + CODE[v]);
        System.out.println();
    }

    public static void main(String[] args) {
        List<List<Integer>> adj = buildCampus();
        int[] dist = new int[V], parent = new int[V];
        System.out.println("BFS from LIB\nstep  dequeued");
        List<Integer> order = bfs(adj, 0, dist, parent, true);

        System.out.print("\nVisiting order:");
        for (int v : order) System.out.print(" " + CODE[v]);
        System.out.println("\n\nvertex  dist  parent");
        for (int v = 0; v < V; v++) {
            if (dist[v] == -1) System.out.printf("  %s     -     -%n", CODE[v]);
            else System.out.printf("  %s  %4d  %5s%n", CODE[v], dist[v],
                    parent[v] == -1 ? "-" : CODE[parent[v]]);
        }
        System.out.println("\nFewest-hop routes from LIB");
        printPath(parent, dist, 7);
        printPath(parent, dist, 4);
```

Kode 13.5 (Lanjutan) *Bfs.java: penelusuran melebar dalam Java, dengan ArrayDeque sebagai queue.*

```
        printPath(parent, dist, 9);
    }
}
```

Kode 13.6 `bfs.py`: penelusuran melebar dalam Python, dengan `collections.deque` sebagai *queue*.

```python
# bfs.py -- breadth-first search on the Pustaka Ceria campus map:
# visiting order, hop distances, parents, and fewest-hop routes.
from collections import deque

V = 10
CODE = ["LIB", "CAF", "ADM", "ENG", "DRM", "MED", "AUD", "PRK", "ANX", "WHS"]
PATHS = [(0, 1, 120), (0, 2, 80), (0, 3, 150), (1, 4, 200), (1, 5, 180),
         (2, 3, 60), (3, 6, 100), (4, 5, 90), (5, 6, 160), (6, 7, 70),
         (8, 9, 50)]


def build_campus():                          # weights are not needed by BFS
    adj = [[] for _ in range(V)]
    for u, v, _ in PATHS:
        adj[u].append(v)
        adj[v].append(u)
    return adj


def bfs(adj, s, trace=False):
    """Return (order, dist, parent); dist is -1 for vertices not reached."""
    dist = [-1] * V
    parent = [-1] * V
    order = []
    q = deque()
    dist[s] = 0                              # mark s when it ENTERS the queue
    q.append(s)
    step = 0
    while q:
        u = q.popleft()
        order.append(u)
        added = ""
        for v in adj[u]:
            if dist[v] == -1:                # first time we see v
                dist[v] = dist[u] + 1
                parent[v] = u
                q.append(v)
                added += " " + CODE[v]
        if trace:
            step += 1
            shown = "[" + " ".join(CODE[x] for x in q) + "]"
            print(f"{step:4}  {CODE[u]}(d={dist[u]})  "
                  f"enqueued:{added or ' -':<12} queue now {shown}")
    return order, dist, parent
```

Kode 13.6 (Lanjutan) *bfs.py: penelusuran melebar dalam Python, dengan collections.deque sebagai queue.*

```
def print_path(parent, dist, t):
    if dist[t] == -1:
        print(f"  to {CODE[t]}: no route")
        return
    path = deque()                          # walk the parents back from t
    v = t
    while v != -1:
        path.appendleft(v)
        v = parent[v]
    print(f"  to {CODE[t]} ({dist[t]} hops): " + " ".join(CODE[x] for x in path))


def main():
    adj = build_campus()
    print("BFS from LIB\nstep  dequeued")
    order, dist, parent = bfs(adj, 0, trace=True)
    print("\nVisiting order: " + " ".join(CODE[v] for v in order))
    print("\nvertex  dist  parent")
    for v in range(V):
        if dist[v] == -1:
            print(f"  {CODE[v]}     -     -")
        else:
            p = "-" if parent[v] == -1 else CODE[parent[v]]
            print(f"  {CODE[v]}  {dist[v]:4}  {p:>5}")
    print("\nFewest-hop routes from LIB")
    for t in (7, 4, 9):                     # PRK, DRM, WHS
        print_path(parent, dist, t)


if __name__ == "__main__":
    main()
```

Keduanya mencetak keluaran yang persis sama dengan **Kode 13.4**. Mengiterasi `ArrayDeque` di Java, atau `deque` di Python, mengunjungi elemen-elemennya dari depan ke belakang tanpa mengeluarkannya, sehingga keduanya tidak memerlukan salinan yang harus dibuat oleh `show` di C++.

13.7 Penelusuran Mendalam (Depth-First Search)

13.7.1 Gagasannya: menjelajahi labirin dengan segulung benang

Dalam legenda Yunani, Theseus memasuki Labirin Minotaurus sambil mengurai segulung benang pemberian Ariadne, agar ia selalu dapat menemukan jalan pulang. Benang itulah gagasan *penelusuran mendalam* (*depth-first search*, DFS). Dari sebuah ruangan, Anda mengambil lorong mana saja yang belum pernah dicoba dan mengikutinya ke ruangan berikutnya, dan dari sana lagi-lagi lorong yang belum dicoba, makin lama makin dalam. Bila Anda tiba di ruangan yang semua lorongnya menuju ruangan yang sudah pernah dilihat, Anda mengikuti benang mundur satu ruangan dan mencoba lorong berikutnya yang belum dicoba di ruangan itu. Bila Anda telah mundur sampai ke pintu masuk dan tidak ada lagi lorong yang belum dicoba, Anda telah menjelajahi segala sesuatu yang dapat dicapai.

BFS menjelajah dengan hati-hati, cincin demi cincin; DFS menjelajah dengan berani, sejauh mungkin di sepanjang satu rute sebelum mundur. Benang itu adalah *stack*: ujungnya adalah ruangan tempat Anda berada, masuk lebih dalam berarti *push* sebuah ruangan, dan mundur berarti *pop*. Cara paling alami memprogramnya adalah dengan rekursi (Bab 10), karena *call stack* menyimpan benang itu untuk kita: setiap panggilan rekursif adalah satu langkah lebih dalam, dan setiap *return* adalah satu langkah mundur. Bentuk modern DFS, beserta waktu penemuan dan

waktu selesai yang dijelaskan berikut ini, dirumuskan oleh Robert Tarjan pada tahun 1972 [5], berdasarkan karyanya bersama John Hopcroft [6]; gagasan menjelajahi labirin itu sendiri sudah ada sejak abad kesembilan belas.

13.7.2 DFS rekursif dengan waktu penemuan dan waktu selesai

DFS memberi setiap simpul sebuah warna yang mencatat kedudukannya dalam penelusuran: *putih* (belum ditemukan), *abu-abu* (sudah ditemukan, tetapi penelusuran masih menjelajah di bawahnya, jadi ia sedang berada pada benang), dan *hitam* (selesai: semua yang dapat dicapai darinya sudah dijelajahi). DFS juga menyimpan sebuah jam global, `time`, dan mencap setiap simpul dua kali: *waktu penemuan* $v.d$ ketika simpul itu menjadi abu-abu dan *waktu selesai* $v.f$ ketika ia menjadi hitam. (Perhatikan bahwa kini d berarti waktu penemuan, bukan jarak; huruf-hurufnya mengikuti Cormen dkk. [2].) **Pseudokode 13.2** memberikan prosedurnya. Prosedur luar `DFS` memulai penelusuran baru dari setiap simpul yang masih putih, sehingga seluruh graf dijelajahi walaupun grafnya tidak terhubung.

Pseudokode 13.2 `DFS` dan `DFS-VISIT`: penelusuran mendalam dengan waktu penemuan dan waktu selesai.

```
DFS(G):
 1.  for each vertex u in G.V
 2.      u.color ← WHITE
 3.      u.π ← NIL
 4.  time ← 0
 5.  for each vertex u in G.V
 6.      if u.color = WHITE
 7.          DFS-VISIT(G, u)

DFS-VISIT(G, u):
 1.  time ← time + 1
 2.  u.d ← time                    // u ditemukan
 3.  u.color ← GRAY
 4.  for each v in G.adj[u]
 5.      if v.color = WHITE          // lorong yang belum dijelajahi
 6.          v.π ← u
 7.          DFS-VISIT(G, v)         // masuk lebih dalam
 8.  u.color ← BLACK
 9.  time ← time + 1
10.  u.f ← time                    // u selesai
```

Rekursi ini tidak mempunyai baris kasus dasar yang eksplisit; ia berhenti dengan sendirinya. Panggilan pada simpul yang semua tetangganya tidak putih hanya menjalankan gelungnya tanpa berekursi, lalu kembali. Karena setiap panggilan mengubah simpulnya menjadi abu-abu sebelum melihat tetangga mana pun, dan hanya simpul putih yang dikunjungi, `DFS-VISIT` dipanggil tepat satu kali untuk setiap simpul, sekalipun grafnya penuh siklus.

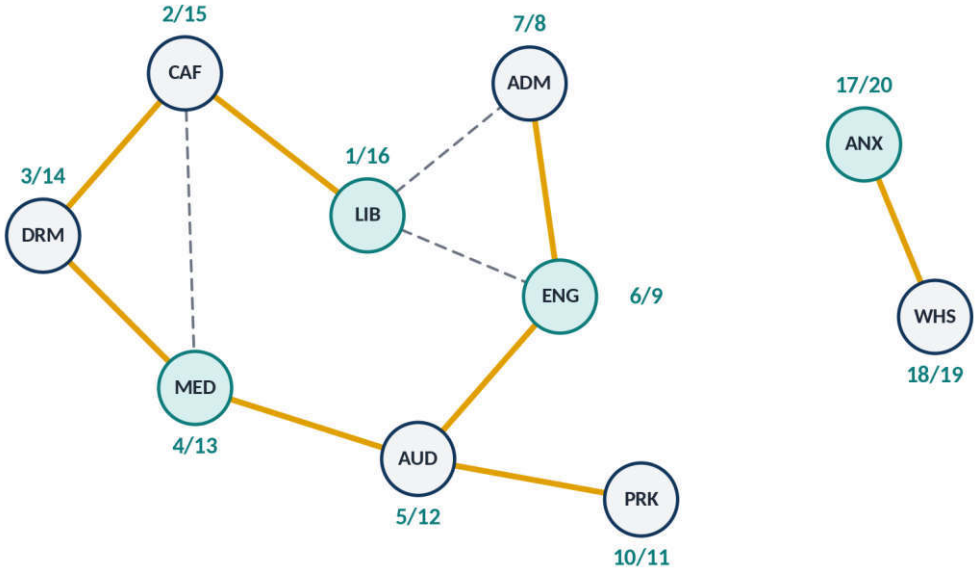
Waktu jalannya dihitung seperti untuk BFS. Baris 1 sampai 4 dan gelung baris 5 sampai 7 pada `DFS` berongkos $\Theta(V)$ ditambah panggilan-panggilannya. Ada satu panggilan `DFS-VISIT` untuk setiap simpul, dan panggilan pada u menghabiskan $O(1 + \deg(u))$ di dalam gelungnya sendiri, tidak termasuk panggilan rekursif yang dibuatnya. Dijumlahkan, DFS berongkos $\Theta(V + E)$ dengan list ketetanggaan, sama dengan Persamaan (13.8). Memori tambahannya $\Theta(V)$ untuk warna, orang tua, dan waktu, ditambah tumpukan rekursi yang kedalamannya dapat mencapai $|V|$ panggilan, hal yang dibahas lagi di Subbab 13.7.5.

13.7.3 Menelusuri DFS pada peta kampus

Jalankan `DFS` pada peta kampus, mencoba simpul awal dalam urutan LIB, CAF, ..., WHS dan tetangga menurut urutan list ketetanggaan.

- Waktu 1: LIB ditemukan. Tetangga pertamanya, CAF, putih: masuk lebih dalam.
- Waktu 2: CAF ditemukan (orang tua LIB). Tetangga pertamanya, LIB, abu-abu (sedang pada benang): lewati. Berikutnya, DRM putih: masuk lebih dalam.
- Waktu 3: DRM ditemukan (orang tua CAF). CAF abu-abu: lewati. MED putih: masuk lebih dalam.
- Waktu 4: MED ditemukan (orang tua DRM). CAF dan DRM abu-abu: lewati keduanya. AUD putih: masuk lebih dalam.
- Waktu 5: AUD ditemukan (orang tua MED). Tetangga pertamanya, ENG, putih: masuk lebih dalam.
- Waktu 6: ENG ditemukan (orang tua AUD). LIB abu-abu: lewati. ADM putih: masuk lebih dalam.
- Waktu 7: ADM ditemukan (orang tua ENG). LIB dan ENG abu-abu. ADM tidak punya apa-apa lagi: pada waktu 8 ia selesai.
- Kembali di ENG: AUD, tetangga terakhirnya, abu-abu. ENG selesai pada waktu 9.
- Kembali di AUD: MED abu-abu; PRK putih. PRK ditemukan pada waktu 10; satu-satunya tetangganya, AUD, abu-abu; PRK selesai pada waktu 11. AUD tidak punya tetangga lagi dan selesai pada waktu 12.
- Benang tergulung kembali: MED selesai pada 13, DRM pada 14, CAF pada 15 (tetangga terakhirnya, MED, hitam), LIB pada 16 (ADM dan ENG hitam).
- Gelung luar `DFS` menemukan simpul putih berikutnya, ANX: ditemukan pada 17, WHS pada 18, selesai pada 19 dan 20.

Urutan kunjungannya adalah LIB, CAF, DRM, MED, AUD, ENG, ADM, PRK, lalu ANX, WHS. Bandingkan dengan urutan BFS, LIB, CAF, ADM, ENG, DRM, MED, AUD, PRK: DFS langsung berlari mengelilingi putaran panjang pada peta, sedalam enam gedung, sebelum kembali untuk PRK, sedangkan BFS tidak pernah pergi lebih dari tiga langkah dari LIB. **Gambar 13.7** menggambar hasilnya. Label di samping setiap simpul adalah d/f; sisi-sisi kuning tua adalah *sisi pohon* (*tree edges*), yaitu sisi-sisi yang dilalui DFS ketika menemukan simpul baru, dan semuanya membentuk *hutan DFS* (*DFS forest*), di sini dua pohon, satu berakar di LIB dan satu di ANX. Sisi-sisi putus-putus adalah *sisi balik* (*back edges*): sisi dari sebuah simpul ke simpul abu-abu, yaitu leluhur yang masih berada pada benang.

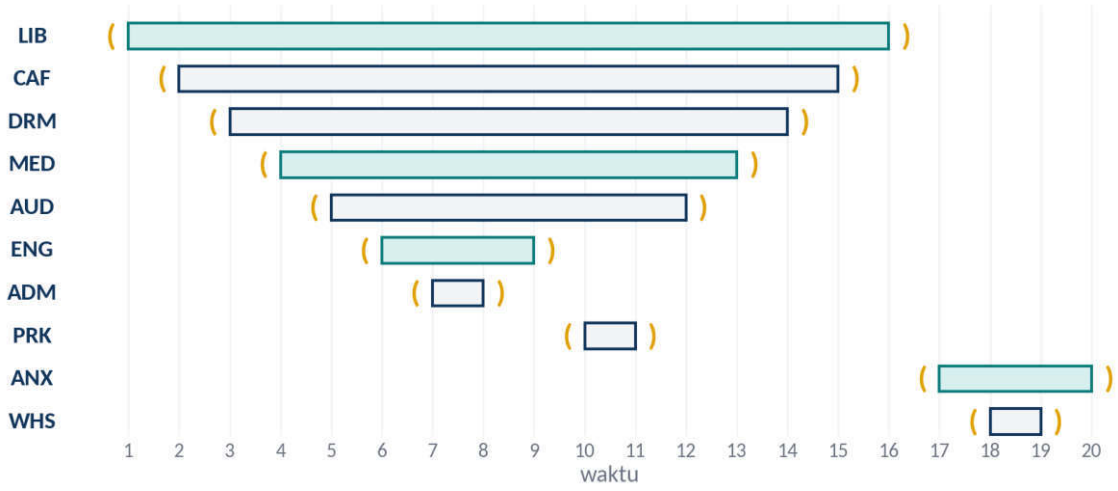


Gambar 13.7 Penelusuran mendalam pada peta kampus. Setiap simpul diberi label waktu penemuan dan waktu selesainya d/f; sisi kuning tua adalah sisi pohon dan sisi putus-putus adalah sisi balik.

Setiap sisi bukan-pohon pada **Gambar 13.7** adalah sisi balik, dan setiap sisi balik menutup sebuah siklus: CAF-MED menutup CAF, DRM, MED, CAF; LIB-ENG menutup LIB, CAF, DRM, MED, AUD, ENG, LIB; LIB-ADM menutup LIB, CAF, DRM, MED, AUD, ENG, ADM, LIB. Dalam graf tak berarah hal ini selalu terjadi. Ketika DFS memeriksa sisi $\{u, v\}$ dan v tidak putih, v adalah orang tua u (sisi pohon yang dilihat dari ujung lainnya) atau leluhur abu-abu dari u; v tidak mungkin simpul hitam di cabang lain, karena bila v sudah selesai sebelum u ditemukan, sisi $\{v, u\}$ sudah diperiksa dari v ketika u masih putih, dan u akan menjadi anak v. Jadi dalam graf tak berarah hanya ada sisi pohon dan sisi balik, dan graf itu mempunyai siklus tepat bila DFS menemukan sisi balik. Subbab 13.7.6 mengubah fakta ini menjadi pendeteksi siklus.

13.7.4 Struktur tanda kurung

Tulislah penemuan setiap simpul sebagai kurung buka dan selesainya sebagai kurung tutup, pada sebuah garis waktu. **Gambar 13.8** melakukannya untuk peta kampus: setiap batang membentang dari d sampai f.



Gambar 13.8 Selang dari penemuan sampai selesai untuk setiap simpul pada DFS di **Gambar 13.7**: setiap dua selang bersarang atau saling lepas, seperti tanda kurung yang berpasangan dengan benar.

Membaca batang-batang itu dari atas ke bawah menghasilkan "(LIB (CAF (DRM (MED (AUD (ENG (ADM ADM) ENG) (PRK PRK) AUD) MED) DRM) CAF) LIB) (ANX (WHS WHS) ANX)", sebuah ekspresi yang tanda kurungnya berpasangan sempurna, seperti kurung-kurung di Subbab 5.6. Inilah *teorema tanda kurung* (*parenthesis theorem*) [2]: untuk setiap dua simpul u dan v, tepat satu dari keadaan berikut berlaku.

$$\begin{aligned}[u.d, u.f] \cap [v.d, v.f] = \emptyset \quad &\text{atau} \quad [u.d, u.f] \\ \subset [v.d, v.f] \quad &\text{atau} \quad [v.d, v.f] \subset [u.d, u.f]\end{aligned} \tag{13.9}$$

Alasannya adalah disiplin *stack* pada benang itu. Bila v ditemukan ketika u abu-abu, maka v di-*push* di atas u, sehingga ia harus di-*pop* (selesai) sebelum u: selang v terletak di dalam selang u, dan v adalah keturunan u dalam hutan DFS. Bila v ditemukan sesudah u selesai, kedua selang saling lepas. Tidak ada kemungkinan lain. Pada **Gambar 13.8**, batang PRK (10 sampai 11) terletak di dalam batang AUD (5 sampai 12) karena PRK keturunan AUD, dan saling lepas dengan batang ENG (6 sampai 9) karena keduanya berada di cabang yang berbeda di bawah AUD. Waktu selesai akan menjadi kunci pengurutan topologis di Subbab 13.9.3.

13.7.5 DFS tanpa rekursi

Rekursi itu anggun, tetapi setiap panggilan menempati sebuah *frame* pada *call stack*, dan *call stack* itu kecil: biasanya 8 MB di Linux dan 1 MB untuk *thread* utama di Windows (Subbab 10.12 mengukur batas-batas ini dalam ketiga bahasa). DFS di sepanjang lintasan sejuta simpul, yang mudah terjadi pada jaringan jalan, membuat sejuta panggilan bersarang dan meluapkan *stack*. Obatnya, seperti yang diperlihatkan Subbab 10.13 untuk rekursi pada umumnya, adalah menyimpan benang itu dalam *stack* eksplisit di memori biasa.

Bagian yang halus adalah membuat *stack* eksplisit itu mengingat apa yang diingat oleh sebuah panggilan rekursif. Ketika `DFS-VISIT(u)` memanggil dirinya sendiri untuk seorang tetangga lalu kembali, ia melanjutkan gelungnya pada *tetangga berikutnya* dari u. Jadi setiap entri *stack* harus menyimpan bukan hanya sebuah simpul, tetapi juga sebuah posisi dalam list ketetanggaannya: pasangan (u, i) yang berarti "kita berada di simpul u dan tetangga berikutnya yang akan dicoba adalah `adj[u][i]`". **Pseudokode 13.3** adalah hasilnya.

Pseudokode 13.3 `DFS-ITERATIVE`: penelusuran yang sama dengan *stack* eksplisit berisi pasangan (simpul, indeks tetangga berikutnya), menghasilkan waktu penemuan dan waktu selesai yang sama dengan versi rekursif.

```
DFS-ITERATIVE(G, s):
 1.  time ← 1;  s.d ← time
 2.  S ← an empty stack;  PUSH(S, (s, 0))
 3.  while S is not empty
 4.      (u, i) ← TOP(S)
 5.      if i < length of G.adj[u]
 6.          v ← G.adj[u][i]
 7.          TOP(S).i ← i + 1          // lain kali, coba tetangga sesudah v
 8.          if v.d = 0               // putih: "panggil" DFS-VISIT(v)
 9.              time ← time + 1;  v.d ← time
10.              PUSH(S, (v, 0))
11.      else                         // tak ada tetangga tersisa: "kembali" dari u
12.          time ← time + 1;  u.f ← time
13.          POP(S)
```

Setiap putaran gelung melakukan salah satu dari dua hal. Bila simpul teratas masih mempunyai tetangga yang belum dicoba, ia mencobanya dan, bila tetangga itu putih, melakukan *push* (masuk

lebih dalam). Bila tidak, ia menyelesaikan simpul teratas dan melakukan *pop* (mundur). Karena *stack* memuat tepat rantai simpul abu-abu, dengan tepat posisi gelung yang akan disimpan oleh panggilan-panggilan rekursif, waktu penemuan dan waktu selesainya identik dengan **Pseudokode 13.2**, klaim yang diperiksa oleh program di bawah. *Stack* itu tidak pernah memuat lebih dari $|V|$ entri, karena setiap simpul di-*push* satu kali, ketika ia menjadi abu-abu.

DFS iteratif yang lebih pendek sering dijumpai: *push* sumber; berulang-ulang *pop* sebuah simpul, lewati bila sudah dikunjungi, dan bila belum, tandai lalu *push* semua tetangganya yang belum dikunjungi (dalam urutan terbalik, agar tetangga pertama berada di puncak). Versi ini mengunjungi simpul-simpul dalam urutan mendalam, tetapi *stack*-nya dapat memuat sampai $|E|$ entri, sebagian di antaranya ganda, dan ia tidak menghasilkan waktu selesai yang benar, karena tidak ada saat tertentu ketika sebuah simpul dapat dikatakan "selesai". Untuk urutan penemuan saja versi ini baik; untuk apa pun yang memerlukan waktu selesai, seperti pengurutan topologis di Subbab 13.9.3, pakailah **Pseudokode 13.3**. Soal Latihan 13.6 membandingkan keduanya.

13.7.6 Menemukan siklus dalam graf tak berarah

Subbab 13.7.3 memperlihatkan bahwa graf tak berarah mempunyai siklus tepat bila DFS bertemu sisi balik: sisi dari simpul saat ini u ke simpul v yang sudah dikunjungi dan *bukan* orang tua u sendiri. (Orang tua dikecualikan karena sisi pohon $\{\pi[u], u\}$ muncul lagi di `adj[u]`, dilihat dari ujung lainnya, dan berjalan di sepanjang satu sisi lalu langsung kembali bukanlah siklus.) Ketika sisi balik ditemukan, siklusnya sendiri dapat dibaca dari orang tua: mulai di u, ikuti π naik sampai v, dan tutup putarannya dengan sisi $\{u, v\}$. Pada peta kampus, sisi balik pertama yang dijumpai DFS adalah MED-CAF, ditemukan pada waktu 4 ketika MED sedang memeriksa tetangga-tetangganya, dan orang tua menghasilkan MED, DRM, CAF, lalu kembali ke MED, yaitu segitiga di sebelah kiri peta.

Fungsi `findCycle` pada listing berikut melakukan persis hal itu. Ia meneruskan simpul asal kedatangannya sebagai `from`, alih-alih memeriksa larik orang tua, yang merupakan cara umum menuliskannya. (Uji ini mengasumsikan graf sederhana. Dalam multigraf, dua sisi ganda antara u dan v sudah membentuk siklus, dan uji itu harus mengecualikan sisi tertentu yang dilalui untuk datang, bukan simpulnya.)

13.7.7 Programnya

Kode 13.7 memuat `dfsVisit` rekursif, yang mencetak setiap penemuan dengan indentasi sesuai kedalamannya dalam hutan DFS; `dfsIterative` dari **Pseudokode 13.3**, dengan `std::stack` berisi pasangan simpul dan indeks tetangga; pemeriksaan bahwa keduanya menghasilkan waktu penemuan dan waktu selesai yang sama; dan `findCycle`. Referensi `std::size_t &i = st.top().second` patut dicermati: menaikkan `i` mengubah indeks yang tersimpan *di dalam* entri teratas, dan dengan cara itulah baris 7 pada **Pseudokode 13.3** memperbarui puncak *stack* di tempat.

Kode 13.7 `dfs.cpp`: penelusuran mendalam rekursif dan iteratif dengan waktu penemuan dan waktu selesai, serta pendeteksian siklus dalam graf tak berarah.

```cpp
// dfs.cpp -- depth-first search on the Pustaka Ceria campus map:
// recursive with discovery/finish times, iterative with an explicit stack,
// and cycle detection in an undirected graph.
#include <cstdio>
#include <stack>
#include <utility>
#include <vector>

const int V = 10;
const char *CODE[V] = {"LIB", "CAF", "ADM", "ENG", "DRM",
                       "MED", "AUD", "PRK", "ANX", "WHS"};
const int PATHS[][3] = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                        {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                        {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};

std::vector<std::vector<int>> adj(V);
std::vector<int> d(V), f(V), parent(V);   // discovery, finish, DFS-tree parent
int timer = 0;

void dfsVisit(int u, int depth) {
    d[u] = ++timer;                       // discovered
    std::printf("%*s%s d=%d\n", 2 * depth, "", CODE[u], d[u]);
    for (int v : adj[u]) {
        if (d[v] == 0) {                  // unvisited neighbor: go deeper
            parent[v] = u;
            dfsVisit(v, depth + 1);
        }
    }
    f[u] = ++timer;                       // finished: every neighbor is done
}

// The same search without recursion. Each stack entry is a vertex plus the
// index of the next neighbor to try, exactly what a recursive call remembers.
std::vector<int> dfsIterative(int s, std::vector<int> &dd, std::vector<int> &ff) {
    std::vector<int> order;
    int t = 0;
    dd.assign(V, 0);
    ff.assign(V, 0);
    std::stack<std::pair<int, std::size_t>> st;
    dd[s] = ++t;
    order.push_back(s);
    st.push({s, 0});
    while (!st.empty()) {
        int u = st.top().first;
        std::size_t &i = st.top().second;  // reference: advancing i updates the top
        if (i < adj[u].size()) {
```

Kode 13.7 (Lanjutan) *dfs.cpp: penelusuran mendalam rekursif dan iteratif dengan waktu penemuan dan waktu selesai, serta pendeteksian siklus dalam graf tak berarah.*

```
            int v = adj[u][i++];
            if (dd[v] == 0) {               // "call" dfsVisit(v)
                dd[v] = ++t;
                order.push_back(v);
                st.push({v, 0});
            }
        } else {                            // "return" from u
            ff[u] = ++t;
            st.pop();
        }
    }
    return order;
}

// Undirected cycle: a visited neighbor that is not the vertex we came from.
bool findCycle(int u, int from, std::vector<int> &par, std::vector<bool> &seen) {
    seen[u] = true;
    for (int v : adj[u]) {
        if (!seen[v]) {
            par[v] = u;
            if (findCycle(v, u, par, seen)) return true;
        } else if (v != from) {             // an edge back to an ancestor
            std::printf("Cycle found at edge %s-%s:", CODE[u], CODE[v]);
            for (int x = u; x != v; x = par[x]) std::printf(" %s", CODE[x]);
            std::printf(" %s %s\n", CODE[v], CODE[u]);
            return true;
        }
    }
    return false;
}

int main() {
    for (const auto &p : PATHS) {
        adj[p[0]].push_back(p[1]);
        adj[p[1]].push_back(p[0]);
    }
    std::printf("Recursive DFS over all vertices (indentation = depth)\n");
    parent.assign(V, -1);
    for (int s = 0; s < V; ++s)
        if (d[s] == 0) dfsVisit(s, 0);
    std::printf("\nvertex   d   f  parent\n");
    for (int v = 0; v < V; ++v)
        std::printf("  %s  %3d %3d  %s\n", CODE[v], d[v], f[v],
                    parent[v] == -1 ? "-" : CODE[parent[v]]);
```

Kode 13.7 (Lanjutan) *dfs.cpp: penelusuran mendalam rekursif dan iteratif dengan waktu penemuan dan waktu selesai, serta pendeteksian siklus dalam graf tak berarah.*

```
    std::vector<int> dd, ff;
    std::vector<int> order = dfsIterative(0, dd, ff);
    std::printf("\nIterative DFS from LIB:");
    for (int v : order) std::printf(" %s", CODE[v]);
    bool same = true;
    for (int v : order) same = same && dd[v] == d[v] && ff[v] == f[v];
    std::printf("\nSame d and f as the recursive search: %s\n",
                same ? "yes" : "no");

    std::vector<int> par(V, -1);
    std::vector<bool> seen(V, false);
    if (!findCycle(0, -1, par, seen)) std::printf("No cycle reachable from LIB\n");

    std::vector<bool> seen2(V, false);   // the ANX-WHS piece is a single edge
    if (!findCycle(8, -1, par, seen2)) std::printf("No cycle reachable from ANX\n");
    return 0;
}
```

```
Recursive DFS over all vertices (indentation = depth)
LIB d=1
  CAF d=2
    DRM d=3
      MED d=4
        AUD d=5
          ENG d=6
            ADM d=7
          PRK d=10
ANX d=17
  WHS d=18

vertex   d   f  parent
  LIB    1  16  -
  CAF    2  15  LIB
  ADM    7   8  ENG
  ENG    6   9  AUD
  DRM    3  14  CAF
  MED    4  13  DRM
  AUD    5  12  MED
  PRK   10  11  AUD
  ANX   17  20  -
  WHS   18  19  ANX

Iterative DFS from LIB: LIB CAF DRM MED AUD ENG ADM PRK
Same d and f as the recursive search: yes
Cycle found at edge MED-CAF: MED DRM CAF MED
No cycle reachable from ANX
```

Daftar berindentasi itu adalah hutan DFS pada **Gambar 13.7** yang dibaringkan: setiap simpul muncul di bawah simpul yang menemukannya. Indentasinya bertambah sampai enam tingkat dalam perjalanan ke ADM, sedangkan BFS tidak pernah melampaui jarak 3. Tabel d, f, dan orang tua sesuai dengan **Gambar 13.7** dan **Gambar 13.8**, dan penelusuran iteratif mereproduksi setiap cap waktu. Pendeteksi siklus melaporkan segitiga MED, DRM, CAF, dan dengan benar tidak menemukan siklus pada komponen ANX-WHS, yang hanya berupa satu sisi.

Kode 13.8 dan **Kode 13.9** adalah program yang sama dalam Java dan Python. Java tidak mempunyai pasangan berisi simpul dan indeks yang dapat diubah, sehingga setiap entri *stack* adalah larik `int` berisi dua elemen yang elemen keduanya dinaikkan di tempat; `ArrayDeque` dengan `push`, `peek`, dan `pop` berfungsi sebagai *stack* (Subbab 5.16.2). Di Python, list berisi dua elemen memainkan peran yang sama, dan *stack*-nya adalah `list` biasa dengan `append` dan `pop`.

Kode 13.8 `Dfs.java`: penelusuran mendalam, rekursif dan iteratif, dalam Java.

```java
// Dfs.java -- depth-first search on the Pustaka Ceria campus map:
// recursive with discovery/finish times, iterative with an explicit stack,
// and cycle detection in an undirected graph.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Dfs {
    static final int V = 10;
    static final String[] CODE = {"LIB", "CAF", "ADM", "ENG", "DRM",
                                  "MED", "AUD", "PRK", "ANX", "WHS"};
    static final int[][] PATHS = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                                  {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                                  {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};

    static List<List<Integer>> adj = new ArrayList<>();
    static int[] d = new int[V], f = new int[V], parent = new int[V];
    static int timer = 0;

    static void dfsVisit(int u, int depth) {
        d[u] = ++timer;                                  // discovered
        System.out.printf("%s%s d=%d%n", "  ".repeat(depth), CODE[u], d[u]);
        for (int v : adj.get(u)) {
            if (d[v] == 0) {                             // go deeper
                parent[v] = u;
                dfsVisit(v, depth + 1);
            }
        }
        f[u] = ++timer;                                  // finished
    }

    // Each stack entry is {vertex, index of the next neighbor to try}.
    static List<Integer> dfsIterative(int s, int[] dd, int[] ff) {
        List<Integer> order = new ArrayList<>();
        int t = 0;
        Arrays.fill(dd, 0);
        Arrays.fill(ff, 0);
        ArrayDeque<int[]> st = new ArrayDeque<>();
        dd[s] = ++t;
        order.add(s);
        st.push(new int[] {s, 0});
        while (!st.isEmpty()) {
            int[] top = st.peek();
            int u = top[0];
            if (top[1] < adj.get(u).size()) {
```

Kode 13.8 (Lanjutan) *Dfs.java: penelusuran mendalam, rekursif dan iteratif, dalam Java.*

```
                int v = adj.get(u).get(top[1]++);
                if (dd[v] == 0) {                       // "call" dfsVisit(v)
                    dd[v] = ++t;
                    order.add(v);
                    st.push(new int[] {v, 0});
                }
            } else {                                    // "return" from u
                ff[u] = ++t;
                st.pop();
            }
        }
        return order;
    }

    static boolean findCycle(int u, int from, int[] par, boolean[] seen) {
        seen[u] = true;
        for (int v : adj.get(u)) {
            if (!seen[v]) {
                par[v] = u;
                if (findCycle(v, u, par, seen)) return true;
            } else if (v != from) {                     // an edge back to an ancestor
                System.out.printf("Cycle found at edge %s-%s:", CODE[u], CODE[v]);
                for (int x = u; x != v; x = par[x]) System.out.print(" " + CODE[x]);
                System.out.printf(" %s %s%n", CODE[v], CODE[u]);
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
        for (int[] p : PATHS) {
            adj.get(p[0]).add(p[1]);
            adj.get(p[1]).add(p[0]);
        }
        System.out.println("Recursive DFS over all vertices (indentation = depth)");
        Arrays.fill(parent, -1);
        for (int s = 0; s < V; s++)
            if (d[s] == 0) dfsVisit(s, 0);
        System.out.println("\nvertex   d   f  parent");
        for (int v = 0; v < V; v++)
            System.out.printf("  %s  %3d %3d  %s%n", CODE[v], d[v], f[v],
                    parent[v] == -1 ? "-" : CODE[parent[v]]);
```

Kode 13.8 (Lanjutan) *Dfs.java: penelusuran mendalam, rekursif dan iteratif, dalam Java.*

```
        int[] dd = new int[V], ff = new int[V];
        List<Integer> order = dfsIterative(0, dd, ff);
        System.out.print("\nIterative DFS from LIB:");
        for (int v : order) System.out.print(" " + CODE[v]);
        boolean same = true;
        for (int v : order) same = same && dd[v] == d[v] && ff[v] == f[v];
        System.out.printf("%nSame d and f as the recursive search: %s%n",
                same ? "yes" : "no");

        int[] par = new int[V];
        Arrays.fill(par, -1);
        if (!findCycle(0, -1, par, new boolean[V]))
            System.out.println("No cycle reachable from LIB");
        if (!findCycle(8, -1, par, new boolean[V]))
            System.out.println("No cycle reachable from ANX");
    }
}
```

Kode 13.9 `dfs.py`: penelusuran mendalam, rekursif dan iteratif, dalam Python.

```
# dfs.py -- depth-first search on the Pustaka Ceria campus map:
# recursive with discovery/finish times, iterative with an explicit stack,
# and cycle detection in an undirected graph.

V = 10
CODE = ["LIB", "CAF", "ADM", "ENG", "DRM", "MED", "AUD", "PRK", "ANX", "WHS"]
PATHS = [(0, 1, 120), (0, 2, 80), (0, 3, 150), (1, 4, 200), (1, 5, 180),
         (2, 3, 60), (3, 6, 100), (4, 5, 90), (5, 6, 160), (6, 7, 70),
         (8, 9, 50)]

adj = [[] for _ in range(V)]
d = [0] * V                                     # discovery time, 0 = unvisited
f = [0] * V                                     # finish time
parent = [-1] * V
timer = 0


def dfs_visit(u, depth):
    global timer
    timer += 1
    d[u] = timer                                # discovered
    print(f"{'  ' * depth}{CODE[u]} d={d[u]}")
    for v in adj[u]:
        if d[v] == 0:                           # unvisited neighbor: go deeper
            parent[v] = u
            dfs_visit(v, depth + 1)
    timer += 1
    f[u] = timer                                # finished


def dfs_iterative(s):
    """Each stack entry is [vertex, index of the next neighbor to try]."""
    dd, ff, order, t = [0] * V, [0] * V, [], 1
    dd[s] = t
    order.append(s)
    stack = [[s, 0]]
    while stack:
        top = stack[-1]
        u = top[0]
        if top[1] < len(adj[u]):
            v = adj[u][top[1]]
            top[1] += 1
            if dd[v] == 0:                      # "call" dfs_visit(v)
                t += 1
                dd[v] = t
                order.append(v)
```

Kode 13.9 (Lanjutan) *dfs.py: penelusuran mendalam, rekursif dan iteratif, dalam Python.*

```
                stack.append([v, 0])
        else:                                   # "return" from u
            t += 1
            ff[u] = t
            stack.pop()
    return order, dd, ff


def find_cycle(u, came_from, par, seen):
    seen[u] = True
    for v in adj[u]:
        if not seen[v]:
            par[v] = u
            if find_cycle(v, u, par, seen):
                return True
        elif v != came_from:                    # an edge back to an ancestor
            walk, x = [], u
            while x != v:
                walk.append(CODE[x])
                x = par[x]
            print(f"Cycle found at edge {CODE[u]}-{CODE[v]}: "
                  + " ".join(walk + [CODE[v], CODE[u]]))
            return True
    return False


def main():
    for u, v, _ in PATHS:
        adj[u].append(v)
        adj[v].append(u)
    print("Recursive DFS over all vertices (indentation = depth)")
    for s in range(V):
        if d[s] == 0:
            dfs_visit(s, 0)
    print("\nvertex   d   f  parent")
    for v in range(V):
        p = "-" if parent[v] == -1 else CODE[parent[v]]
        print(f"  {CODE[v]}  {d[v]:3} {f[v]:3}  {p}")

    order, dd, ff = dfs_iterative(0)
    print("\nIterative DFS from LIB: " + " ".join(CODE[v] for v in order))
    same = all(dd[v] == d[v] and ff[v] == f[v] for v in order)
    print(f"Same d and f as the recursive search: {'yes' if same else 'no'}")
```

Kode 13.9 (Lanjutan) *dfs.py: penelusuran mendalam, rekursif dan iteratif, dalam Python.*

```
    par = [-1] * V
    if not find_cycle(0, -1, par, [False] * V):
        print("No cycle reachable from LIB")
    if not find_cycle(8, -1, par, [False] * V):   # ANX-WHS is a single edge
        print("No cycle reachable from ANX")


if __name__ == "__main__":
    main()
```

Keduanya mencetak keluaran yang persis sama dengan **Kode 13.7**. Satu peringatan untuk Python: CPython secara bawaan membatasi kedalaman rekursi sampai 1000 panggilan, sehingga `dfs_visit` rekursif gagal dengan `RecursionError` pada graf apa pun yang mempunyai lintasan lebih dari kira-kira seribu simpul. Menaikkan batas itu dengan `sys.setrecursionlimit` hanya memindahkan masalahnya ke arah kerusakan interpreter yang sesungguhnya; untuk graf besar di Python, versi iteratif adalah satu-satunya yang aman.

13.8 Keterhubungan dan Komponen Terhubung

13.8.1 Satu penelusuran untuk setiap komponen

BFS maupun DFS yang dimulai dari simpul s mengunjungi tepat simpul-simpul yang dapat dicapai dari s, yaitu tepat komponen terhubung tempat s berada dalam graf tak berarah. Jadi komponen-komponen dapat ditemukan dengan gelung yang paling sederhana: telusuri simpul-simpul secara berurutan; setiap kali sebuah simpul belum berlabel, ia memulai komponen baru, dan satu penelusuran dari simpul itu melabeli seluruh komponennya. **Pseudokode 13.4** menuliskannya dengan BFS; DFS akan sama baiknya, dan gelung luar `DFS` pada **Pseudokode 13.2** sebenarnya sudah merupakan algoritma ini.

Pseudokode 13.4 `CONNECTED-COMPONENTS`: melabeli setiap simpul dengan nomor komponennya, satu penelusuran melebar untuk setiap komponen.

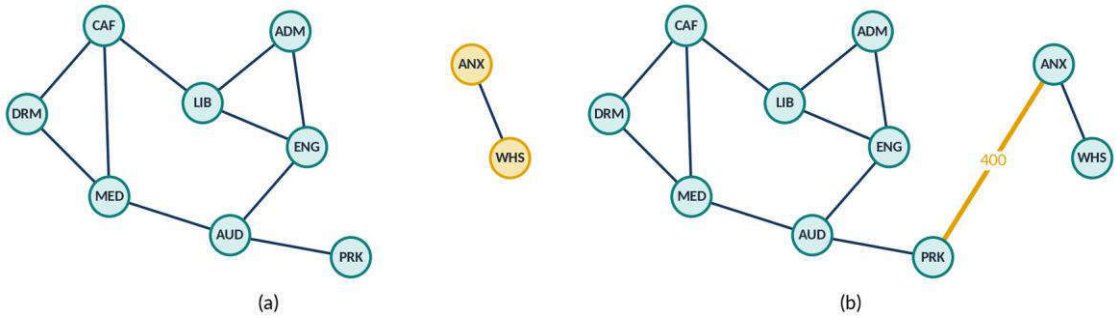
```
CONNECTED-COMPONENTS(G):
 1.  for each vertex v in G.V
 2.      v.comp ← -1                 // belum masuk komponen mana pun
 3.  count ← 0
 4.  for each vertex s in G.V
 5.      if s.comp = -1              // s memulai komponen baru
 6.          s.comp ← count
 7.          Q ← an empty queue;  ENQUEUE(Q, s)
 8.          while Q is not empty    // satu BFS melabeli seluruh komponen
 9.              u ← DEQUEUE(Q)
10.              for each v in G.adj[u]
11.                  if v.comp = -1
12.                      v.comp ← count
13.                      ENQUEUE(Q, v)
14.          count ← count + 1
15.  return count
```

Mudah tergoda untuk mengira bahwa gelung bersarang ini membuatnya kuadratik, padahal tidak. Setiap simpul dilabeli satu kali dan dimasukkan ke *queue* satu kali selama seluruh eksekusi, bukan satu kali per komponen, dan setiap list ketetanggaan dipindai satu kali, ketika simpulnya dikeluarkan. Penelusuran-penelusuran untuk komponen yang berbeda menyentuh bagian graf yang saling lepas, sehingga ongkos-ongkosnya berjumlah satu kali lintasan atas semuanya: $\Theta(V + E)$, sama dengan satu BFS. Hopcroft dan Tarjan menjadikan pengamatan ini dasar sebuah keluarga algoritma graf berwaktu linear [6].

Begitu setiap simpul membawa nomor komponennya, beberapa pertanyaan menjadi pencarian $O(1)$. Apakah u dan v terhubung? Tepat bila `comp[u] = comp[v]`. Apakah seluruh graf terhubung? Tepat bila `count` bernilai 1. Seberapa besar komponen tempat v berada? Hitunglah simpul-simpul yang berlabel sama, atau lebih baik, catat ukurannya selama penelusuran.

13.8.2 Jalan shuttle

Pada peta kampus, **Pseudokode 13.4** mulai di LIB, yang BFS-nya melabeli kedelapan gedung kampus dengan komponen 0; gelungnya lalu melewati CAF, ADM, dan yang lain, yang sudah berlabel, sampai di ANX, dan melabeli ANX serta WHS dengan komponen 1. Peta ini mempunyai dua komponen, seperti diperlihatkan **Gambar 13.9**(a) dengan dua warna. Pengelola perpustakaan kini membuka jalan *shuttle* sepanjang 400 meter dari tempat parkir ke annex kota. Satu sisi baru, PRK-ANX, ditambahkan ke list ketetanggaan dan algoritma dijalankan lagi: BFS-nya dari LIB kini melintasi jalan baru itu, dan kesepuluh gedung masuk ke komponen 0 (**Gambar 13.9**(b)).



Gambar 13.9 Komponen terhubung peta kampus: (a) sebagaimana dibangun, dua komponen (hijau toska dan kuning tua); (b) sesudah jalan *shuttle* PRK-ANX sepanjang 400 meter ditambahkan, satu komponen.

Satu sisi dapat mengubah banyaknya komponen paling banyak satu: menambahkan sisi di antara dua komponen yang berbeda menggabungkan keduanya, dan menambahkan sisi di dalam satu komponen tidak mengubah apa pun. Menghapus sisi adalah bayangan cerminnya. Sisi yang penghapusannya akan memutuskan komponennya, seperti AUD-PRK atau PRK-ANX yang baru, disebut *jembatan* (*bridge*), untuk mengenang Königsberg; LIB-CAF bukan jembatan, karena siklus-siklus di sekitarnya menyediakan rute lain. Menemukan semua jembatan secara efisien adalah penerapan klasik waktu penemuan dan waktu selesai DFS [5], dan jaringan yang andal, baik jaringan jalan maupun jaringan komputer, adalah jaringan tanpa jembatan.

13.8.3 Programnya

Kode 13.10 mengimplementasikan **Pseudokode 13.4** beserta laporan yang mencetak komponen-komponen, apakah peta itu terhubung, dan apakah LIB serta WHS berada dalam komponen yang sama; kemudian program menambahkan jalan *shuttle* dan melapor lagi.

Kode 13.10 `components.cpp`: komponen terhubung peta kampus, sebelum dan sesudah jalan *shuttle*.

```cpp
// components.cpp -- connected components of the Pustaka Ceria campus map,
// before and after a shuttle road joins the city annex to the campus.
#include <cstdio>
#include <queue>
#include <vector>

const int V = 10;
const char *CODE[V] = {"LIB", "CAF", "ADM", "ENG", "DRM",
                       "MED", "AUD", "PRK", "ANX", "WHS"};
const int PATHS[][3] = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                        {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                        {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};

// comp[v] = number of v's component (0, 1, 2, ...); returns how many there are.
int components(const std::vector<std::vector<int>> &adj, std::vector<int> &comp) {
    comp.assign(adj.size(), -1);
    int count = 0;
    for (int s = 0; s < static_cast<int>(adj.size()); ++s) {
        if (comp[s] != -1) continue;       // already inside an earlier component
        std::queue<int> q;                 // one BFS sweeps out one component
        comp[s] = count;
        q.push(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int v : adj[u])
                if (comp[v] == -1) {
                    comp[v] = count;
                    q.push(v);
                }
        }
        ++count;
    }
    return count;
}
```

Kode 13.10 (Lanjutan) *components.cpp: komponen terhubung peta kampus, sebelum dan sesudah jalan shuttle.*

```cpp
void report(const char *title, const std::vector<std::vector<int>> &adj) {
    std::vector<int> comp;
    int k = components(adj, comp);
    std::printf("%s: %d component(s)\n", title, k);
    for (int c = 0; c < k; ++c) {
        std::printf("  component %d:", c);
        for (int v = 0; v < V; ++v)
            if (comp[v] == c) std::printf(" %s", CODE[v]);
        std::printf("\n");
    }
    std::printf("  connected? %s\n", k == 1 ? "yes" : "no");
    std::printf("  same component LIB and WHS? %s\n",
                comp[0] == comp[9] ? "yes" : "no");
}

int main() {
    std::vector<std::vector<int>> adj(V);
    for (const auto &p : PATHS) {
        adj[p[0]].push_back(p[1]);
        adj[p[1]].push_back(p[0]);
    }
    report("Campus map as built", adj);

    adj[7].push_back(8);                    // new shuttle road PRK-ANX, 400 m
    adj[8].push_back(7);
    std::printf("\nAdding the shuttle road PRK-ANX\n\n");
    report("After the shuttle road", adj);
    return 0;
}
```

```
Campus map as built: 2 component(s)
  component 0: LIB CAF ADM ENG DRM MED AUD PRK
  component 1: ANX WHS
  connected? no
  same component LIB and WHS? no

Adding the shuttle road PRK-ANX

After the shuttle road: 1 component(s)
  component 0: LIB CAF ADM ENG DRM MED AUD PRK ANX WHS
  connected? yes
  same component LIB and WHS? yes
```

Program yang sama dalam Java (**Kode 13.11**) dan Python (**Kode 13.12**) menyusul. Seperti sebelumnya, hanya kelas *queue*-nya yang berubah.

Kode 13.11 `Components.java`: komponen terhubung dalam Java.

```java
// Components.java -- connected components of the Pustaka Ceria campus map,
// before and after a shuttle road joins the city annex to the campus.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Components {
    static final int V = 10;
    static final String[] CODE = {"LIB", "CAF", "ADM", "ENG", "DRM",
                                  "MED", "AUD", "PRK", "ANX", "WHS"};
    static final int[][] PATHS = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                                  {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                                  {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};

    // comp[v] = number of v's component; returns how many components there are.
    static int components(List<List<Integer>> adj, int[] comp) {
        Arrays.fill(comp, -1);
        int count = 0;
        for (int s = 0; s < adj.size(); s++) {
            if (comp[s] != -1) continue;             // inside an earlier component
            ArrayDeque<Integer> q = new ArrayDeque<>();   // one BFS, one component
            comp[s] = count;
            q.addLast(s);
            while (!q.isEmpty()) {
                int u = q.pollFirst();
                for (int v : adj.get(u))
                    if (comp[v] == -1) {
                        comp[v] = count;
                        q.addLast(v);
                    }
            }
            count++;
        }
        return count;
    }
```

Kode 13.11 (Lanjutan) *Components.java: komponen terhubung dalam Java.*

```
    static void report(String title, List<List<Integer>> adj) {
        int[] comp = new int[V];
        int k = components(adj, comp);
        System.out.printf("%s: %d component(s)%n", title, k);
        for (int c = 0; c < k; c++) {
            System.out.printf("  component %d:", c);
            for (int v = 0; v < V; v++)
                if (comp[v] == c) System.out.print(" " + CODE[v]);
            System.out.println();
        }
        System.out.printf("  connected? %s%n", k == 1 ? "yes" : "no");
        System.out.printf("  same component LIB and WHS? %s%n",
                comp[0] == comp[9] ? "yes" : "no");
    }

    public static void main(String[] args) {
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
        for (int[] p : PATHS) {
            adj.get(p[0]).add(p[1]);
            adj.get(p[1]).add(p[0]);
        }
        report("Campus map as built", adj);

        adj.get(7).add(8);                          // shuttle road PRK-ANX, 400 m
        adj.get(8).add(7);
        System.out.println("\nAdding the shuttle road PRK-ANX\n");
        report("After the shuttle road", adj);
    }
}
```

Kode 13.12 `components.py`: komponen terhubung dalam Python.

```python
# components.py -- connected components of the Pustaka Ceria campus map,
# before and after a shuttle road joins the city annex to the campus.
from collections import deque

V = 10
CODE = ["LIB", "CAF", "ADM", "ENG", "DRM", "MED", "AUD", "PRK", "ANX", "WHS"]
PATHS = [(0, 1, 120), (0, 2, 80), (0, 3, 150), (1, 4, 200), (1, 5, 180),
         (2, 3, 60), (3, 6, 100), (4, 5, 90), (5, 6, 160), (6, 7, 70),
         (8, 9, 50)]


def components(adj):
    """Return (count, comp) where comp[v] is the number of v's component."""
    comp = [-1] * len(adj)
    count = 0
    for s in range(len(adj)):
        if comp[s] != -1:                    # already inside an earlier component
            continue
        q = deque([s])                       # one BFS sweeps out one component
        comp[s] = count
        while q:
            u = q.popleft()
            for v in adj[u]:
                if comp[v] == -1:
                    comp[v] = count
                    q.append(v)
        count += 1
    return count, comp


def report(title, adj):
    k, comp = components(adj)
    print(f"{title}: {k} component(s)")
    for c in range(k):
        print(f"  component {c}: "
              + " ".join(CODE[v] for v in range(V) if comp[v] == c))
    print(f"  connected? {'yes' if k == 1 else 'no'}")
    print(f"  same component LIB and WHS? {'yes' if comp[0] == comp[9] else 'no'}")
```

Kode 13.12 (Lanjutan) *components.py: komponen terhubung dalam Python.*

```python
def main():
    adj = [[] for _ in range(V)]
    for u, v, _ in PATHS:
        adj[u].append(v)
        adj[v].append(u)
    report("Campus map as built", adj)

    adj[7].append(8)                         # new shuttle road PRK-ANX, 400 m
    adj[8].append(7)
    print("\nAdding the shuttle road PRK-ANX\n")
    report("After the shuttle road", adj)


if __name__ == "__main__":
    main()
```

Keduanya mencetak keluaran yang persis sama dengan **Kode 13.10**. Program-program ini menjalankan ulang seluruh pelabelan sesudah menambahkan satu sisi, yang berongkos $\Theta(V + E)$ setiap kali. Bila sisi ditambahkan satu per satu dan pertanyaan “apakah u dan v terhubung?” diajukan di sela-selanya, seperti pada jaringan yang terus tumbuh, struktur *disjoint-set* atau

union-find menjawab setiap operasi dalam waktu teramortisasi yang hampir konstan [2]; struktur itu termasuk materi tingkat lanjut, tetapi patut diketahui keberadaannya.

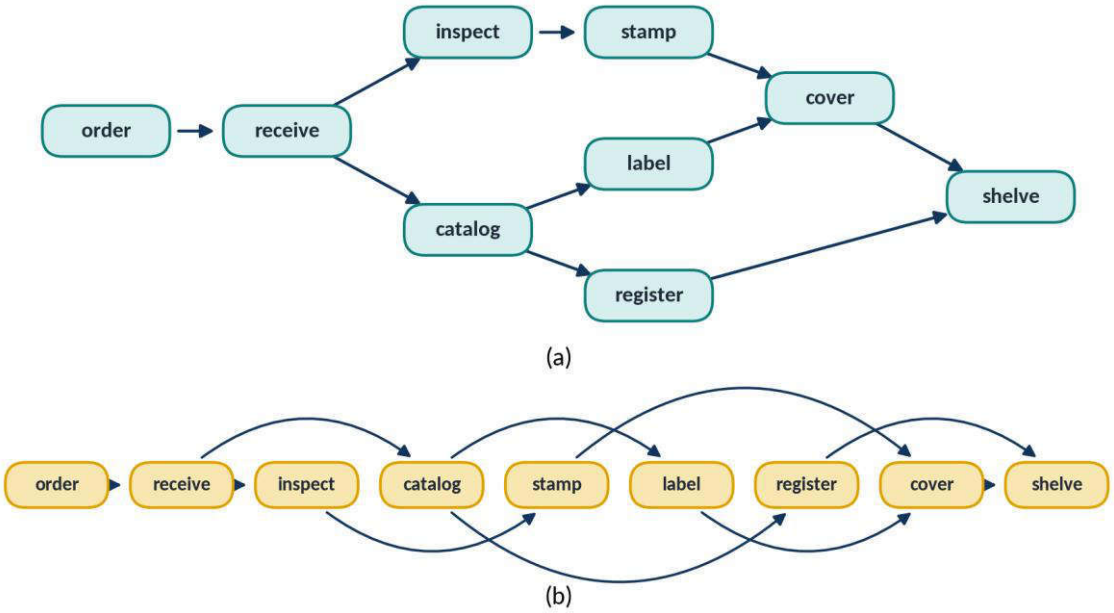
13.8.4 Graf berarah: catatan tentang keterhubungan kuat

Dalam graf berarah, "dapat dicapai" tidak lagi simetris: jalan satu arah dapat membawa Anda dari u ke v tanpa jalan kembali. Graf berarah disebut *terhubung kuat* (*strongly connected*) bila setiap simpul dapat mencapai setiap simpul lain *dengan mengikuti arah panah*, dan *komponen terhubung kuat*-nya adalah himpunan-himpunan simpul maksimal yang semua anggotanya dapat saling mencapai. Penelusuran dari s dalam graf berarah menemukan segala sesuatu yang dapat dicapai *dari* s, yang pada umumnya lebih banyak daripada komponen terhubung kuat tempat s berada. Algoritma baku untuk komponen terhubung kuat, oleh Tarjan [5] serta oleh Kosaraju dan Sharir [2], dibangun di atas dua kali DFS dan waktu selesainya, dan berjalan dalam $\Theta(V + E)$; keduanya di luar cakupan buku ini. Perayap web di Subbab 13.13 menjumpai pertanyaan berarah yang lebih sederhana, "halaman mana yang dapat dicapai dari halaman beranda?", dan BFS menjawabnya secara langsung.

13.9 Graf Asiklik Berarah dan Pengurutan Topologis

13.9.1 Alur kerja buku baru

Sebelum buku yang baru tiba sampai di rak Pustaka Ceria, sembilan tugas harus dikerjakan: buku *dipesan* dari penerbit (*order*), *diterima* di gudang (*receive*), *diperiksa* dari kerusakan (*inspect*), *dikatalogkan* (*catalog*, diberi nomor klasifikasi dan rekaman katalog), *dicap* dengan cap perpustakaan (*stamp*), *diberi label* punggung (*label*), *disampul* plastik pelindung (*cover*), *didaftarkan* dalam katalog daring (*register*), dan akhirnya *dijajarkan di rak* (*shelve*). Nama-nama tugas dalam bahasa Inggris itu dipakai apa adanya dalam gambar dan program. Tugas-tugas itu tidak dapat dikerjakan dalam sembarang urutan. Buku tidak dapat diperiksa sebelum diterima, dan tidak dapat diberi label sebelum dikatalogkan, karena labelnya memuat nomor klasifikasi. Ketergantungan-ketergantungan itu membentuk graf berarah pada **Gambar 13.10**(a), dengan panah (u, v) berarti "tugas u harus selesai sebelum tugas v dimulai".



Gambar 13.10 Alur kerja buku baru: (a) DAG ketergantungannya, dengan panah dari setiap tugas ke tugas yang bergantung padanya; (b) DAG yang sama dalam urutan topologis hasil algoritma Kahn, dengan semua panah menunjuk ke kanan.

Graf ini adalah DAG: dengan mengikuti panah, Anda tidak pernah dapat kembali ke tugas yang telah Anda tinggalkan. Seorang petugas perpustakaan yang memproses satu buku pada satu waktu memerlukan urutan kesembilan tugas yang setiap panahnya menunjuk *ke depan*, sehingga setiap tugas datang sesudah semua prasyaratnya.

> **DEFINISI.** *Urutan topologis* (*topological order* atau *topological sort*) sebuah graf berarah adalah daftar semua simpulnya sedemikian rupa sehingga untuk setiap panah (u, v), u muncul sebelum v.

Gambar 13.10(b) memperlihatkan satu urutan topologis: dijajarkan dalam satu garis, kesepuluh panah semuanya menunjuk ke kanan. Urutan topologis ada jika dan hanya jika grafnya DAG. Satu arah mudah: bila ada siklus $v_1 \to v_2 \to \cdots \to v_k \to v_1$, maka v_1 harus mendahului v_2, yang harus mendahului v_3, dan seterusnya, sampai v_k harus mendahului v_1, sehingga v_1 harus mendahului dirinya sendiri, sesuatu yang tidak dapat dicapai daftar mana pun. Arah lainnya, bahwa setiap DAG mempunyai urutan topologis, dibuktikan oleh kedua algoritma berikut, yang masing-masing membangun satu urutan. Urutan topologis biasanya tidak tunggal: dalam alur kerja ini, *inspect* dan *catalog* dapat dikerjakan dalam urutan mana pun begitu buku diterima.

13.9.2 Algoritma Kahn

Algoritma pertama, yang diterbitkan Arthur B. Kahn pada tahun 1962 untuk menjadwalkan tugas-tugas proyek rekayasa yang besar [7], melakukan apa yang dilakukan orang yang bijak dengan daftar pekerjaannya: berulang-ulang memilih tugas yang tidak mempunyai prasyarat yang belum selesai, mengerjakannya, lalu mencoretnya. “Tidak mempunyai prasyarat yang belum selesai” berarti “berderajat-masuk 0 di antara tugas-tugas yang tersisa”. Algoritma ini menyimpan derajat-masuk saat ini untuk setiap simpul dan sebuah *queue* berisi simpul-simpul yang derajat-masuknya sudah turun menjadi 0.

Pseudokode 13.5 KAHN: pengurutan topologis dengan berulang-ulang membuang simpul berderajat-masuk 0.

```
KAHN(G):
 1.  for each vertex v in G.V
 2.      v.indeg ← 0
 3.  for each vertex u in G.V
 4.      for each v in G.adj[u]
 5.          v.indeg ← v.indeg + 1
 6.  Q ← a queue holding every vertex with indeg = 0
 7.  L ← an empty list
 8.  while Q is not empty
 9.      u ← DEQUEUE(Q)
10.      append u to L                 // u dapat dikerjakan sekarang
11.      for each v in G.adj[u]        // coret u: v kehilangan satu prasyarat
12.          v.indeg ← v.indeg - 1
13.          if v.indeg = 0
14.              ENQUEUE(Q, v)
15.  if L has fewer than |G.V| vertices
16.      report "the graph has a cycle"
17.  return L
```

Mengapa algoritma ini benar? Sebuah simpul masuk ke L hanya ketika derajat-masuknya sudah turun menjadi 0, yang berarti semua prasyaratnya sudah ditambahkan ke L sebelumnya: setiap panah menunjuk ke depan dalam L. Dan mengapa ia tidak macet pada DAG? Selama masih ada simpul tersisa, bagian DAG yang tersisa juga merupakan DAG, dan setiap DAG dengan sedikitnya satu simpul mempunyai simpul berderajat-masuk 0 (bila tidak, mulailah dari mana saja dan terus berjalan mundur melalui panah-panah yang masuk; karena simpulnya berhingga, jalan itu pada akhirnya mengulang sebuah simpul, dan itu berarti siklus). Jadi *queue* tidak pernah kering selama masih ada simpul, dan semua $|V|$ simpul dikeluarkan. Sebaliknya, pada graf yang mengandung siklus, tidak ada simpul siklus yang pernah mencapai derajat-masuk 0, karena masing-masing mempunyai prasyarat pada siklus itu yang tidak pernah dikeluarkan; algoritma berhenti lebih awal, dan baris 15 mendeteksinya.

Waktu jalannya $\Theta(V + E)$: baris 1 sampai 5 melihat setiap simpul dan panah satu kali, setiap simpul dimasukkan dan dikeluarkan dari *queue* satu kali, dan baris 11 sampai 14 melihat setiap panah satu kali lagi.

Tabel 13.3 menelusuri algoritma Kahn pada alur kerja tersebut. Derajat-masuk awalnya 0 untuk *order*, 2 untuk *cover* dan *shelve* (masing-masing menunggu dua tugas), dan 1 untuk setiap tugas lainnya.

Tabel 13.3 Algoritma Kahn pada alur kerja buku baru.

Langkah	Keluaran *u*	Derajat-masuk yang turun	Menjadi 0 (dimasukkan)	*Queue* sesudahnya
0			order	order
1	order	receive 1 → 0	receive	receive
2	receive	inspect 1 → 0, catalog 1 → 0	inspect, catalog	inspect catalog
3	inspect	stamp 1 → 0	stamp	catalog stamp
4	catalog	label 1 → 0, register 1 → 0	label, register	stamp label register
5	stamp	cover 2 → 1	tidak ada	label register
6	label	cover 1 → 0	cover	register cover
7	register	shelve 2 → 1	tidak ada	cover
8	cover	shelve 1 → 0	shelve	shelve
9	shelve	tidak ada	tidak ada	(kosong)

Keluarannya adalah order, receive, inspect, catalog, stamp, label, register, cover, shelve, yaitu urutan yang dijajarkan pada **Gambar 13.10**(b). Langkah 5 dan 7 memperlihatkan derajat-masuk menjalankan tugasnya: *stamp* sudah selesai, tetapi *cover* masih menunggu *label*, dan baru ketika *label* selesai pada langkah 6, *cover* menjadi tersedia.

13.9.3 Pengurutan topologis dengan DFS

Algoritma kedua memakai waktu selesai DFS, dan ia luar biasa singkat: jalankan DFS pada seluruh graf dan daftarlah simpul-simpulnya dalam *urutan waktu selesai yang menurun*. Dengan kata lain, setiap kali sebuah simpul selesai, letakkan ia di depan daftar [2], [5].

Pseudokode 13.6 `TOPOLOGICAL-SORT`: urutan topologis menurut waktu selesai DFS yang menurun.

```
TOPOLOGICAL-SORT(G):
 1.  L ← an empty list
 2.  run DFS(G); each time a vertex u finishes (line 10 of DFS-VISIT),
 3.      insert u at the front of L
 4.  return L
```

Mengapa ini benar? Ambillah sembarang panah (u, v) dan lihatlah saat DFS memeriksanya, dari u, ketika u abu-abu. Ada tiga kemungkinan untuk v. Bila v putih, ia menjadi keturunan u dan, menurut struktur tanda kurung Persamaan (13.9), selesai sebelum u. Bila v hitam, ia sudah selesai, sehingga lagi-lagi $v.f < u.f$. Dan v tidak mungkin abu-abu, karena v yang abu-abu adalah leluhur u pada benang, dan panah (u, v) bersama lintasan pohon dari v turun ke u akan membentuk siklus berarah, yang tidak dimiliki DAG. Jadi dalam setiap kasus $v.f < u.f$: setiap panah berjalan dari simpul yang selesai lebih akhir ke simpul yang selesai lebih awal, dan mendaftar menurut waktu selesai yang menurun membuat setiap panah menunjuk ke depan.

Pada alur kerja itu, DFS yang dimulai dari *order* berjalan melalui order, receive, inspect, stamp, cover, shelve sebelum ada yang selesai. *shelve* selesai pertama kali, lalu *cover*, *stamp*, dan *inspect*; kembali di *receive*, penelusuran menuju *catalog* dan *label* (yang tetangganya, *cover*, sudah hitam), lalu *register* (yang tetangganya, *shelve*, hitam). Urutan selesai: shelve, cover, stamp, inspect, label, register, catalog, receive, order. Dibalik: order, receive, catalog, register, label, inspect, stamp, cover, shelve. Urutan ini berbeda dari urutan Kahn, dan sama sahnya: periksalah bahwa setiap panah pada **Gambar 13.10**(a) menunjuk ke depan di dalamnya.

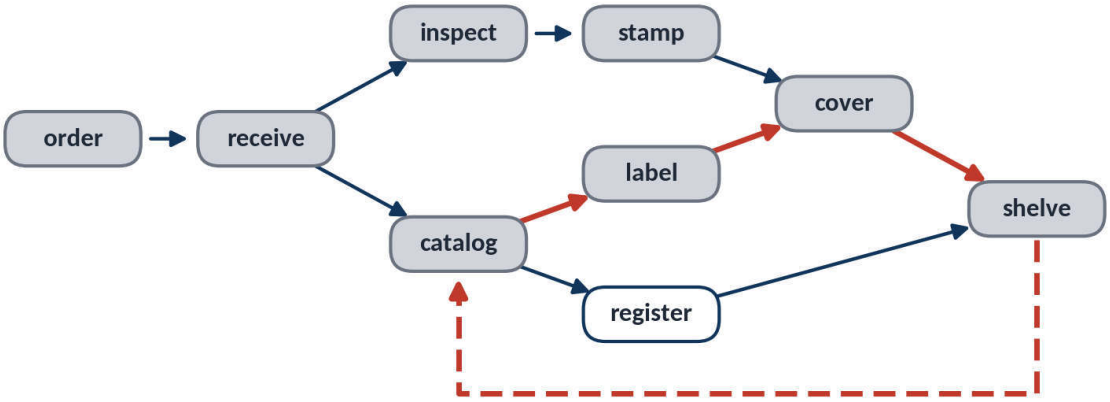
13.9.4 Mendeteksi siklus dalam graf berarah

Bukti di atas mengandung sebuah pendeteksi siklus. Dalam graf berarah, panah dari simpul saat ini u ke simpul *abu-abu* v adalah sisi balik, dan sisi balik berarti siklus berarah: v mencapai u melalui sisi-sisi pohon, dan u mencapai v melalui sisi balik itu. Sebaliknya, bila graf mempunyai siklus, DFS menemukan sisi balik: misalkan v adalah simpul siklus yang pertama kali ditemukan; semua simpul lain pada siklus itu masih putih saat itu dan dapat dicapai dari v, sehingga semuanya menjadi keturunan v, dan panah dari simpul terakhir di antara mereka menuju v diperiksa ketika v masih abu-abu. Jadi *graf berarah asiklik tepat bila DFS tidak menemukan panah menuju simpul abu-abu*. Perhatikan bahwa panah menuju simpul *hitam* tidak berbahaya dalam graf berarah: pada alur kerja itu, *label* ke *cover* tiba di *cover* yang sudah hitam pada DFS di atas, dan tidak ada siklus.

Karena itulah DFS memerlukan tiga warna dalam graf berarah, bukan sekadar “sudah dikunjungi atau belum”: “sudah dikunjungi” mencampuradukkan abu-abu dan hitam, padahal perbedaan

keduanya adalah perbedaan antara siklus dan rute kedua yang tidak berbahaya menuju tugas yang sama.

Misalkan seorang pustakawan, yang bermaksud menyatakan bahwa penjajaran di rak disusul pemeriksaan katalog terakhir, memasukkan panah keliru dari *shelve* ke *catalog*. Kini catalog, label, cover, shelve, catalog adalah siklus, dan tidak ada urutan yang dapat memenuhi semua ketergantungan. Kedua algoritma menyadarinya. Algoritma Kahn mengeluarkan order, receive, inspect, dan stamp, lalu berhenti dengan *queue* kosong: *catalog* masih menunggu *shelve*, yang menunggu *cover*, yang menunggu *label*, yang menunggu *catalog*. Hanya 4 dari 9 tugas yang dikeluarkan, dan baris 15 pada **Pseudokode 13.5** melaporkan siklus itu. DFS berjalan melalui order, receive, inspect, stamp, cover, shelve, lalu melalui panah keliru ke catalog dan terus ke label, yang panahnya menuju *cover*, yang abu-abu. **Gambar 13.11** memperlihatkan saat itu: setiap tugas pada lintasan saat ini berwarna abu-abu, hanya *register* yang masih putih, dan panah-panah merah adalah siklusnya, yang ditutup oleh panah keliru yang putus-putus.



Gambar 13.11 Alur kerja dengan panah keliru dari shelve ke catalog (putus-putus). Ketika DFS memeriksa panah dari label ke cover, setiap tugas pada lintasan saat ini berwarna abu-abu dan cover salah satunya: sebuah sisi balik, yang membuktikan adanya siklus merah.

Sisi balik yang dilaporkan DFS, label ke cover, bukanlah panah keliru itu sendiri, melainkan panah lain pada siklus yang sama; panah mana dari sebuah siklus yang tertangkap bergantung pada tempat penelusuran memasuki siklus itu. Pelajaran praktisnya: pendeteksi siklus membuktikan bahwa siklus ada dan dapat mencetaknya (dengan mengikuti orang tua dari u naik ke v, seperti di Subbab 13.7.6), tetapi keputusan tentang ketergantungan mana yang salah ada di tangan pustakawan.

13.9.5 Programnya

Kode 13.13 menjalankan algoritma Kahn dengan jejak langkah, lalu versi DFS dengan tiga warna, kemudian menambahkan panah keliru itu dan menjalankan keduanya lagi.

Kode 13.13 `topo.cpp`: pengurutan topologis dengan algoritma Kahn dan dengan penelusuran mendalam, beserta pendeteksian siklus.

```cpp
// topo.cpp -- topological sort of the Pustaka Ceria new-book workflow,
// by Kahn's algorithm and by depth-first search, with cycle detection.
#include <algorithm>
#include <cstdio>
#include <queue>
#include <vector>

const int V = 9;
const char *TASK[V] = {"order", "receive", "inspect", "catalog", "stamp",
                       "label", "cover", "register", "shelve"};
// Arc u -> v means "task u must be finished before task v starts".
const int ARCS[][2] = {{0, 1}, {1, 2}, {1, 3}, {2, 4}, {3, 5},
                       {3, 7}, {4, 6}, {5, 6}, {6, 8}, {7, 8}};

using Graph = std::vector<std::vector<int>>;

// Kahn: repeatedly output a task with no unfinished prerequisite.
std::vector<int> kahn(const Graph &g, bool trace) {
    std::vector<int> indeg(V, 0), order;
    for (int u = 0; u < V; ++u)
        for (int v : g[u]) ++indeg[v];
    std::queue<int> q;
    for (int u = 0; u < V; ++u)
        if (indeg[u] == 0) q.push(u);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        if (trace) std::printf("  output %-8s frees:", TASK[u]);
        for (int v : g[u]) {
            if (--indeg[v] == 0) {          // v's last prerequisite is done
                q.push(v);
                if (trace) std::printf(" %s", TASK[v]);
            }
        }
        if (trace) std::printf("\n");
    }
    return order;                           // shorter than V <=> there is a cycle
}

// DFS with three colors: WHITE unvisited, GRAY on the current path, BLACK done.
enum Color { WHITE, GRAY, BLACK };
```

Kode 13.13 (Lanjutan) *topo.cpp: pengurutan topologis dengan algoritma Kahn dan dengan penelusuran mendalam, beserta pendeteksian siklus.*

```
bool dfsTopo(const Graph &g, int u, std::vector<Color> &c, std::vector<int> &out) {
    c[u] = GRAY;
    for (int v : g[u]) {
        if (c[v] == GRAY) {                // back edge: v is an ancestor of u
            std::printf("  back edge %s -> %s closes a cycle\n", TASK[u], TASK[v]);
            return false;
        }
        if (c[v] == WHITE && !dfsTopo(g, v, c, out)) return false;
    }
    c[u] = BLACK;
    out.push_back(u);                      // u finishes after all it leads to
    return true;
}

bool dfsOrder(const Graph &g, std::vector<int> &order) {
    std::vector<Color> c(V, WHITE);
    order.clear();
    for (int u = 0; u < V; ++u)
        if (c[u] == WHITE && !dfsTopo(g, u, c, order)) return false;
    std::reverse(order.begin(), order.end());   // decreasing finish time
    return true;
}

void print(const char *label, const std::vector<int> &order) {
    std::printf("%s", label);
    for (int u : order) std::printf(" %s", TASK[u]);
    std::printf("\n");
}

int main() {
    Graph g(V);
    for (const auto &a : ARCS) g[a[0]].push_back(a[1]);

    std::printf("Kahn's algorithm\n");
    std::vector<int> k = kahn(g, true);
    print("Kahn order:", k);

    std::vector<int> d;
    if (dfsOrder(g, d)) print("DFS order: ", d);
```

Kode 13.13 (Lanjutan) *topo.cpp: pengurutan topologis dengan algoritma Kahn dan dengan penelusuran mendalam, beserta pendeteksian siklus.*

```
    std::printf("\nA mistaken arc shelve -> catalog is added\n");
    g[8].push_back(3);
    k = kahn(g, false);
    print("Kahn outputs only:", k);
    std::printf("  %zu of %d tasks: the rest wait on each other in a cycle\n",
                k.size(), V);
    if (!dfsOrder(g, d)) std::printf("DFS: no topological order exists\n");
    return 0;
}
```

```
Kahn's algorithm
  output order    frees: receive
  output receive  frees: inspect catalog
  output inspect  frees: stamp
  output catalog  frees: label register
  output stamp    frees:
  output label    frees: cover
  output register frees:
  output cover    frees: shelve
  output shelve   frees:
Kahn order: order receive inspect catalog stamp label register cover shelve
DFS order:  order receive catalog register label inspect stamp cover shelve

A mistaken arc shelve -> catalog is added
Kahn outputs only: order receive inspect stamp
  4 of 9 tasks: the rest wait on each other in a cycle
  back edge label -> cover closes a cycle
DFS: no topological order exists
```

Jejak langkahnya adalah **Tabel 13.3**, urutan DFS-nya adalah urutan yang diturunkan di Subbab 13.9.3, dan sesudah panah keliru ditambahkan, kedua algoritma menolak, persis seperti yang diramalkan **Gambar 13.11**. Dalam `dfsTopo`, simpul yang selesai ditambahkan di *akhir* `out` dan seluruh vektor dibalik sekali di ujung; penambahan di akhir berongkos $O(1)$ teramortisasi, sedangkan penyisipan di depan `std::vector` akan berongkos $\Theta(n)$ setiap kali (Subbab 7.3). Versi Java dan Python, **Kode 13.14** dan **Kode 13.15**, menyusul.

Kode 13.14 `Topo.java`: pengurutan topologis dalam Java.

```java
// Topo.java -- topological sort of the Pustaka Ceria new-book workflow,
// by Kahn's algorithm and by depth-first search, with cycle detection.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Topo {
    static final int V = 9;
    static final String[] TASK = {"order", "receive", "inspect", "catalog", "stamp",
                                  "label", "cover", "register", "shelve"};
    // Arc u -> v means "task u must be finished before task v starts".
    static final int[][] ARCS = {{0, 1}, {1, 2}, {1, 3}, {2, 4}, {3, 5},
                                 {3, 7}, {4, 6}, {5, 6}, {6, 8}, {7, 8}};

    static List<Integer> kahn(List<List<Integer>> g, boolean trace) {
        int[] indeg = new int[V];
        List<Integer> order = new ArrayList<>();
        for (int u = 0; u < V; u++)
            for (int v : g.get(u)) indeg[v]++;
        ArrayDeque<Integer> q = new ArrayDeque<>();
        for (int u = 0; u < V; u++)
            if (indeg[u] == 0) q.addLast(u);
        while (!q.isEmpty()) {
            int u = q.pollFirst();
            order.add(u);
            if (trace) System.out.printf("  output %-8s frees:", TASK[u]);
            for (int v : g.get(u)) {
                if (--indeg[v] == 0) {              // last prerequisite done
                    q.addLast(v);
                    if (trace) System.out.print(" " + TASK[v]);
                }
            }
            if (trace) System.out.println();
        }
        return order;                                // shorter than V: a cycle
    }

    static final int WHITE = 0, GRAY = 1, BLACK = 2;
```

Kode 13.14 (Lanjutan) *Topo.java: pengurutan topologis dalam Java.*

```
    static boolean dfsTopo(List<List<Integer>> g, int u, int[] c,
                           List<Integer> out) {
        c[u] = GRAY;
        for (int v : g.get(u)) {
            if (c[v] == GRAY) {                       // back edge
                System.out.printf("  back edge %s -> %s closes a cycle%n",
                        TASK[u], TASK[v]);
                return false;
            }
            if (c[v] == WHITE && !dfsTopo(g, v, c, out)) return false;
        }
        c[u] = BLACK;
        out.add(u);                                   // u finishes
        return true;
    }

    static boolean dfsOrder(List<List<Integer>> g, List<Integer> order) {
        int[] c = new int[V];                         // all WHITE
        order.clear();
        for (int u = 0; u < V; u++)
            if (c[u] == WHITE && !dfsTopo(g, u, c, order)) return false;
        Collections.reverse(order);                   // decreasing finish time
        return true;
    }

    static void print(String label, List<Integer> order) {
        System.out.print(label);
        for (int u : order) System.out.print(" " + TASK[u]);
        System.out.println();
    }

    public static void main(String[] args) {
        List<List<Integer>> g = new ArrayList<>();
        for (int i = 0; i < V; i++) g.add(new ArrayList<>());
        for (int[] a : ARCS) g.get(a[0]).add(a[1]);

        System.out.println("Kahn's algorithm");
        List<Integer> k = kahn(g, true);
        print("Kahn order:", k);

        List<Integer> d = new ArrayList<>();
        if (dfsOrder(g, d)) print("DFS order: ", d);
```

Kode 13.14 (Lanjutan) *Topo.java: pengurutan topologis dalam Java.*

```
        System.out.println("\nA mistaken arc shelve -> catalog is added");
        g.get(8).add(3);
        k = kahn(g, false);
        print("Kahn outputs only:", k);
        System.out.printf(
                "  %d of %d tasks: the rest wait on each other in a cycle%n",
                k.size(), V);
        if (!dfsOrder(g, d)) System.out.println("DFS: no topological order exists");
    }
}
```

Kode 13.15 `topo.py`: pengurutan topologis dalam Python.

```
# topo.py -- topological sort of the Pustaka Ceria new-book workflow,
# by Kahn's algorithm and by depth-first search, with cycle detection.
from collections import deque

V = 9
TASK = ["order", "receive", "inspect", "catalog", "stamp",
        "label", "cover", "register", "shelve"]
# Arc (u, v) means "task u must be finished before task v starts".
ARCS = [(0, 1), (1, 2), (1, 3), (2, 4), (3, 5),
        (3, 7), (4, 6), (5, 6), (6, 8), (7, 8)]


def kahn(g, trace=False):
    indeg = [0] * V
    for u in range(V):
        for v in g[u]:
            indeg[v] += 1
    q = deque(u for u in range(V) if indeg[u] == 0)
    order = []
    while q:
        u = q.popleft()
        order.append(u)
        freed = []
        for v in g[u]:
            indeg[v] -= 1
            if indeg[v] == 0:               # v's last prerequisite is done
                q.append(v)
                freed.append(TASK[v])
        if trace:
            print(f"  output {TASK[u]:<8} frees:" + "".join(" " + t for t in freed))
    return order                            # shorter than V <=> a cycle


WHITE, GRAY, BLACK = 0, 1, 2
```

Kode 13.15 (Lanjutan) *topo.py: pengurutan topologis dalam Python.*

```
def dfs_topo(g, u, color, out):
    color[u] = GRAY
    for v in g[u]:
        if color[v] == GRAY:                # back edge: v is an ancestor of u
            print(f"  back edge {TASK[u]} -> {TASK[v]} closes a cycle")
            return False
        if color[v] == WHITE and not dfs_topo(g, v, color, out):
            return False
    color[u] = BLACK
    out.append(u)                           # u finishes after all it leads to
    return True


def dfs_order(g):
    """Return a topological order, or None if the graph has a cycle."""
    color = [WHITE] * V
    out = []
    for u in range(V):
        if color[u] == WHITE and not dfs_topo(g, u, color, out):
            return None
    return out[::-1]                        # decreasing finish time


def show(label, order):
    print(label + "".join(" " + TASK[u] for u in order))


def main():
    g = [[] for _ in range(V)]
    for u, v in ARCS:
        g[u].append(v)

    print("Kahn's algorithm")
    show("Kahn order:", kahn(g, trace=True))
    d = dfs_order(g)
    if d is not None:
        show("DFS order: ", d)

    print("\nA mistaken arc shelve -> catalog is added")
    g[8].append(3)
    k = kahn(g)
    show("Kahn outputs only:", k)
    print(f"  {len(k)} of {V} tasks: the rest wait on each other in a cycle")
    if dfs_order(g) is None:
        print("DFS: no topological order exists")
```

Kode 13.15 (Lanjutan) *topo.py: pengurutan topologis dalam Python.*

```
if __name__ == "__main__":
    main()
```

Keduanya mencetak keluaran yang persis sama dengan **Kode 13.13**. Pengurutan topologis ada di mana-mana dalam perangkat lunak. Perkakas *build* `make` dan `ninja` mengompilasi berkas-berkas program dalam urutan topologis ketergantungannya; pengelola paket seperti `apt` dan `pip` memasang pustaka dalam urutan topologis kebutuhannya, dan melaporkan "*dependency cycle*" bila urutan itu tidak ada; program lembar kerja menghitung ulang sel-sel dalam urutan topologis rumus-rumus yang saling merujuk, dan melaporkan "*circular reference*" tepat dalam keadaan **Gambar 13.11**. Python bahkan menyertakan algoritma ini dalam pustaka bakunya, sebagai kelas `graphlib.TopologicalSorter`.

13.10 Lintasan Terpendek Berbobot: Dijkstra dengan Perlahan

13.10.1 Mengapa BFS tidak cukup

BFS menjawab "rute mana yang memakai jalan setapak paling sedikit?". Pustakawan yang mendorong troli berisi buku dari perpustakaan pusat ke tempat parkir memedulikan pertanyaan lain: rute mana yang *terpendek dalam meter*? Kedua pertanyaan itu mempunyai jawaban berbeda pada peta kampus. Rute BFS, LIB, ENG, AUD, PRK, terdiri atas 3 jalan setapak dan $150 + 100 + 70 = 320$ meter; rute LIB, ADM, ENG, AUD, PRK terdiri atas 4 jalan setapak tetapi hanya $80 + 60 + 100 + 70 = 310$ meter. BFS tidak dapat menemukan rute kedua, karena ia hanya melihat banyaknya sisi: ia mencapai ENG langsung dari LIB dalam satu langkah dan tidak pernah mempertimbangkan bahwa jalan memutar dua langkah lewat ADM (140 meter) lebih pendek daripada jalan langsungnya (150 meter).

Bila bobotnya berbeda-beda, simpul yang harus dituntaskan berikutnya bukanlah simpul yang ditemukan pertama kali, melainkan simpul yang saat ini *paling dekat*. Mengganti *queue* pertama-masuk-pertama-keluar pada BFS dengan antrean yang selalu memberikan simpul terdekat menghasilkan algoritma yang diterbitkan Edsger W. Dijkstra pada tahun 1959 [8], salah satu algoritma yang paling banyak dipakai dalam komputasi. Algoritma ini mensyaratkan semua bobot tidak negatif, sesuatu yang secara alami dipenuhi oleh panjang, waktu, dan biaya.

13.10.2 Relaksasi

Algoritma Dijkstra menyimpan, untuk setiap simpul v, sebuah *jarak sementara* $d[v]$: panjang rute terpendek dari sumber s ke v yang ditemukan sejauh ini, ∞ bila belum ada yang ditemukan. Ia mulai dengan $d[s] = 0$ dan $d[v] = \infty$ untuk setiap v lainnya, lalu memperbaiki taksiran-taksiran itu dengan satu langkah dasar yang disebut *merelaksasi* (*relaxing*) sisi (u, v):

$$\text{jika } d[u] + w(u,v) < d[v]: \qquad d[v] \leftarrow d[u] + w(u,v), \quad \pi[v] \leftarrow u \tag{13.10}$$

Dalam kata-kata: "Saya tahu rute ke u sepanjang $d[u]$. Melanjutkan dari u ke v menambah $w(u,v)$. Bila itu mengalahkan rute terbaik ke v yang saya ketahui, ingatlah rute baru itu, dan ingatlah bahwa ia datang dari u." Relaksasi tidak pernah membuat taksiran menjadi salah: $d[v]$ selalu merupakan panjang suatu rute sungguhan (atau ∞), sehingga ia hanya dapat terlalu besar, tidak pernah terlalu kecil. Seluruh seninya adalah merelaksasi sisi-sisi dalam urutan yang membuat setiap taksiran tepat secepat mungkin.

13.10.3 Algoritma Dijkstra

Urutan Dijkstra bersifat rakus (*greedy*): berulang-ulang ambil simpul yang belum final dengan jarak sementara terkecil, nyatakan jarak itu *final*, lalu relaksasi setiap sisi yang keluar darinya. *Priority queue* (Subbab 5.15) adalah struktur yang memberikan "entri dengan kunci terkecil" dengan cepat: ia mendukung *insert* dan *extract-min*, masing-masing dalam waktu $O(\log n)$ bila dibangun sebagai *binary heap*, yang menjadi pokok bahasan Bab 14. Bab ini memakai *priority queue* yang sudah disediakan ketiga bahasa: `std::priority_queue` di C++ [9], `java.util.PriorityQueue` di Java [10], dan modul `heapq` di Python [11].

Cara benar yang paling sederhana untuk memakai antrean pustaka semacam itu adalah versi *malas* (*lazy*) pada **Pseudokode 13.7**. Ketika relaksasi menurunkan $d[v]$, kita tidak berusaha mencari dan memperbarui entri lama v di dalam antrean (antrean pustaka tidak dapat melakukannya); kita cukup menyisipkan entri baru $(d[v], v)$. Entri lama yang lebih besar tetap

tertinggal, dan ketika akhirnya keluar dari antrean, v sudah final sehingga entri itu dikenali sebagai *basi* (*stale*) dan dilewati.

Pseudokode 13.7 `DIJKSTRA`: lintasan terpendek dari satu sumber dengan bobot tak negatif, memakai *min-priority queue* dengan penghapusan malas atas entri basi.

```
DIJKSTRA(G, w, s):
 1.  for each vertex v in G.V
 2.      v.d ← ∞;  v.π ← NIL;  v.done ← FALSE
 3.  s.d ← 0
 4.  PQ ← an empty min-priority queue;  INSERT(PQ, (0, s))
 5.  while PQ is not empty
 6.      (du, u) ← EXTRACT-MIN(PQ)
 7.      if u.done                        // entri basi: lewati
 8.          continue
 9.      u.done ← TRUE                    // u.d kini final
10.      for each v in G.adj[u]
11.          if not v.done and u.d + w(u, v) < v.d
12.              v.d ← u.d + w(u, v)      // relaksasi (u, v)
13.              v.π ← u
14.              INSERT(PQ, (v.d, v))
```

Letakkan **Pseudokode 13.7** di samping **Pseudokode 13.1**, maka kemiripannya tampak jelas. Keduanya mengambil simpul dari sebuah antrean lalu melihat tetangga-tetangganya; BFS memakai *queue* FIFO dan menambahkan 1 per sisi, Dijkstra memakai *priority queue* dan menambahkan $w(u, v)$. Pada graf yang semua bobotnya 1, algoritma Dijkstra mengunjungi simpul-simpul tepat dalam urutan BFS dan menghitung jarak yang sama.

13.10.4 Menelusuri Dijkstra dari perpustakaan pusat

Tabel 13.4 menjalankan **Pseudokode 13.7** pada peta kampus dari LIB. Entri *priority queue* ditulis (jarak, simpul); jarak yang sama diputuskan berdasarkan nomor simpul yang lebih kecil, seperti yang dilakukan program.

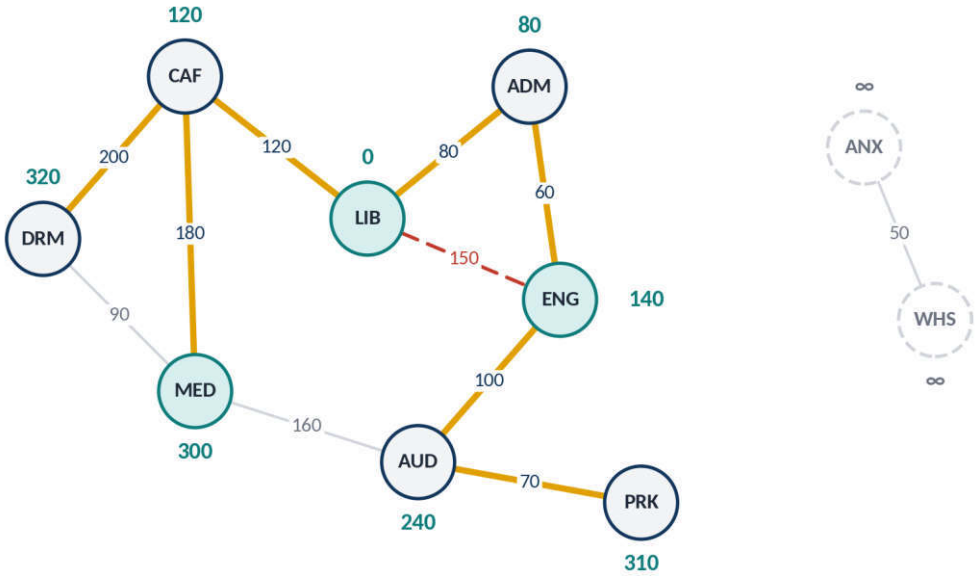
Tabel 13.4 Algoritma Dijkstra dari LIB pada peta kampus. Isi *priority queue* didaftar menurut jarak yang menaik.

Langkah	Diekstrak	Status	Relaksasi yang memperbaiki d	*Priority queue* sesudahnya
0			d[LIB] = 0	(0, LIB)
1	(0, LIB)	final	CAF 120, ADM 80, ENG 150	(80, ADM) (120, CAF) (150, ENG)
2	(80, ADM)	final	ENG 150 → 140	(120, CAF) (140, ENG) (150, ENG)
3	(120, CAF)	final	DRM 320, MED 300	(140, ENG) (150, ENG) (300, MED) (320, DRM)
4	(140, ENG)	final	AUD 240	(150, ENG) (240, AUD) (300, MED) (320, DRM)
5	(150, ENG)	basi, dilewati		(240, AUD) (300, MED) (320, DRM)
6	(240, AUD)	final	PRK 310 (MED: 400 > 300, tidak)	(300, MED) (310, PRK) (320, DRM)
7	(300, MED)	final	tidak ada (DRM: 390 > 320)	(310, PRK) (320, DRM)
8	(310, PRK)	final	tidak ada	(320, DRM)
9	(320, DRM)	final	tidak ada	(kosong)

Langkah 2 adalah saat Dijkstra mengalahkan BFS. ADM diekstrak dengan jarak 80, dan merelaksasi ADM-ENG menemukan rute ke ENG sepanjang $80 + 60 = 140$ meter, lebih baik

daripada 150 meter jalan langsung yang ditemukan pada langkah 1. Taksiran ENG turun dan entri baru (140, ENG) disisipkan; entri lama (150, ENG) tetap berada di antrean. Pada langkah 4 entri baru itu keluar lebih dahulu dan membuat ENG final pada 140; pada langkah 5 entri lama keluar dan dilewati karena basi. Setiap rute berikutnya lewat ENG (ke AUD, dan terus ke PRK) mewarisi angka 140 yang lebih baik itu. Pada langkah 6 relaksasi juga *gagal* untuk sekali ini: dari AUD, MED akan berongkos $240 + 160 = 400$, lebih buruk daripada 300 yang sudah diketahui lewat CAF.

Gambar 13.12 memperlihatkan hasilnya. Setiap simpul membawa jarak finalnya dari LIB; sisi-sisi kuning tua adalah *pohon lintasan terpendek* (*shortest-path tree*), yang dibentuk oleh sisi-sisi $\{\pi[v], v\}$. Jalan setapak merah putus-putus LIB-ENG adalah sisi yang dipakai pohon BFS dan dihindari Dijkstra. ANX dan WHS tetap berjarak ∞: tidak ada rute yang mencapainya.



Gambar 13.12 Jarak berjalan terpendek dalam meter dari LIB. Sisi kuning tua membentuk pohon lintasan terpendek; sisi merah putus-putus LIB-ENG berada pada rute dengan langkah paling sedikit ke PRK, tetapi tidak pada rute terpendek.

13.10.5 Mengapa algoritma ini benar, dan kapan tidak

Klaim kuncinya adalah bahwa ketika sebuah simpul u diekstrak untuk pertama kali, jaraknya $d[u]$ sudah final, yaitu sama dengan jarak terpendek sesungguhnya $\delta(s, u)$. Berikut gagasan buktinya; Cormen dkk. memberikannya secara lengkap [2]. Misalkan, sebaliknya, pada saat itu ada rute P yang lebih pendek dari s ke u. P berawal di s, yang sudah final, dan berakhir di u, yang belum final, sehingga di suatu tempat di sepanjang P ada simpul pertama y yang belum final, yang dicapai dari simpul final x. Ketika x menjadi final, sisi (x, y) direlaksasi, sehingga $d[y] \le \delta(s, x) + w(x, y)$, yaitu panjang bagian P sampai y. Karena semua bobot tidak negatif, sisa P dari y ke u tidak mungkin berpanjang negatif, sehingga $d[y] \le \text{panjang}(P) < d[u]$. Namun, dengan begitu *priority queue* akan memberikan y sebelum u: kontradiksi. Jadi $d[u] = \delta(s, u)$.

Langkah "sisa P tidak mungkin berpanjang negatif" adalah tempat asumsi $w \ge 0$ dipakai, dan asumsi itu tidak dapat dibuang. Tinjaulah tiga simpul s, a, b dengan $w(s, a) = 2$, $w(s, b) = 3$, dan $w(b, a) = -2$. Algoritma Dijkstra mengekstrak s, lalu a dengan jarak 2, dan menyatakannya final; kemudian ia mengekstrak b dengan jarak 3, tetapi a sudah selesai, sehingga rute s, b, a sepanjang $3 - 2 = 1$ tidak pernah tercatat. Bobot negatif memang muncul, misalnya sebagai potongan harga dalam model biaya, dan graf semacam itu memerlukan algoritma lain, seperti Bellman-Ford [2], yang lebih lambat tetapi benar.

13.10.6 Waktu jalan, dan sekilas ke Bab 14

Hitunglah operasi *priority queue* pada **Pseudokode 13.7**. Setiap relaksasi yang berhasil menyisipkan satu entri, dan setiap sisi paling banyak direlaksasi dengan berhasil satu kali dari setiap ujungnya, sehingga ada paling banyak $|E|$ penyisipan untuk graf berarah (atau $2|E|$ untuk graf tak berarah) ditambah satu untuk s, dan ekstraksi dalam jumlah yang sama. Dengan *binary heap*, masing-masing berongkos $O(\log E) = O(\log V)$, karena $|E| \le |V|^2$ berarti $\log|E| \le 2\log|V|$. Sisa pekerjaannya $\Theta(V + E)$ seperti pada BFS. Secara keseluruhan,

$$T_{\text{Dijkstra}} = O((V + E)\log V) \tag{13.11}$$

Untuk peta kampus itu hanya segelintir operasi; untuk 58 juta busur jaringan jalan sebuah negara, waktunya masih jauh di bawah satu detik pada laptop. Dua penyempurnaan patut diketahui namanya. Rumusan asli Dijkstra tahun 1959 sama sekali tidak memakai *priority queue*, hanya larik yang dipindai untuk mencari minimum, yang menghasilkan $\Theta(V^2)$ dan sebenarnya merupakan pilihan yang lebih baik untuk graf padat, ketika E mendekati V^2. Dan dengan *heap* yang lebih rumit, *Fibonacci heap* dari Fredman dan Tarjan [12], batasnya membaik menjadi $O(E + V\log V)$.

> **SEKILAS KE DEPAN.** Segala sesuatu pada Persamaan (13.11) yang tidak linear berasal dari *priority queue*. Bab 14 membuka kotaknya: ia membangun *binary heap* yang berdiri di balik `std::priority_queue`, `PriorityQueue`, dan `heapq`, memperlihatkan mengapa *insert* dan *extract-min* berongkos $O(\log n)$, dan menambahkan operasi *decrease-key* yang memungkinkan algoritma Dijkstra memperbarui sebuah entri alih-alih menyisipkan entri baru.

13.10.7 Programnya

Kode 13.16 mengimplementasikan **Pseudokode 13.7**. `std::priority_queue` pada C++ secara bawaan adalah *max-heap*: `top`-nya adalah elemen terbesar. Mendeklarasikannya dengan pembanding `std::greater<Item>` mengubahnya menjadi *min-heap* atas pasangan (jarak, simpul), dan karena pasangan dibandingkan lebih dahulu menurut komponen pertamanya, `top` adalah entri dengan jarak terkecil, dengan jarak yang sama dimenangkan nomor simpul yang lebih kecil. Baris `auto [du, u] = pq.top();` memakai *structured binding* (C++17) untuk membongkar pasangan itu.

Kode 13.16 `dijkstra.cpp`: algoritma Dijkstra pada peta kampus dengan `std::priority_queue` sebagai *min-heap*.

```cpp
// dijkstra.cpp -- shortest walking distances from the main library,
// with a priority queue (std::priority_queue as a min-heap).
#include <climits>
#include <cstdio>
#include <deque>
#include <functional>
#include <queue>
#include <utility>
#include <vector>

const int V = 10;
const char *CODE[V] = {"LIB", "CAF", "ADM", "ENG", "DRM",
                       "MED", "AUD", "PRK", "ANX", "WHS"};
const int PATHS[][3] = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                        {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                        {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};
const int INF = INT_MAX;

struct Nbr {
    int to, w;
};
using Item = std::pair<int, int>;          // (tentative distance, vertex)

void dijkstra(const std::vector<std::vector<Nbr>> &adj, int s,
              std::vector<int> &dist, std::vector<int> &parent) {
    dist.assign(V, INF);
    parent.assign(V, -1);
    std::vector<bool> done(V, false);
    // greater<Item> turns the default max-heap into a min-heap
    std::priority_queue<Item, std::vector<Item>, std::greater<Item>> pq;
    dist[s] = 0;
    pq.push({0, s});
    while (!pq.empty()) {
        auto [du, u] = pq.top();
        pq.pop();
        if (done[u]) {                     // an old, longer entry: ignore it
            std::printf("  pop (%3d, %s)  stale, skipped\n", du, CODE[u]);
            continue;
        }
        done[u] = true;                    // du is now final
        std::printf("  pop (%3d, %s)  final.", du, CODE[u]);
        for (const Nbr &e : adj[u]) {
            if (!done[e.to] && du + e.w < dist[e.to]) {   // relax u -> e.to
                dist[e.to] = du + e.w;
                parent[e.to] = u;
                pq.push({dist[e.to], e.to});
```

Kode 13.16 (Lanjutan) *dijkstra.cpp: algoritma Dijkstra pada peta kampus dengan std::priority_queue sebagai min-heap.*

```
                std::printf(" %s=%d", CODE[e.to], dist[e.to]);
            }
        }
        std::printf("\n");
    }
}

int main() {
    std::vector<std::vector<Nbr>> adj(V);
    for (const auto &p : PATHS) {
        adj[p[0]].push_back({p[1], p[2]});
        adj[p[1]].push_back({p[0], p[2]});
    }
    std::vector<int> dist, parent;
    std::printf("Dijkstra from LIB (relaxed distances after each pop)\n");
    dijkstra(adj, 0, dist, parent);

    std::printf("\nvertex  dist(m)  route\n");
    for (int v = 0; v < V; ++v) {
        if (dist[v] == INF) {
            std::printf("  %s   unreachable\n", CODE[v]);
            continue;
        }
        std::deque<int> path;
        for (int x = v; x != -1; x = parent[x]) path.push_front(x);
        std::printf("  %s  %7d  ", CODE[v], dist[v]);
        for (std::size_t i = 0; i < path.size(); ++i)
            std::printf("%s%s", i ? "-" : "", CODE[path[i]]);
        std::printf("\n");
    }
    return 0;
}
```

```
Dijkstra from LIB (relaxed distances after each pop)
  pop (  0, LIB)  final. CAF=120 ADM=80 ENG=150
  pop ( 80, ADM)  final. ENG=140
  pop (120, CAF)  final. DRM=320 MED=300
  pop (140, ENG)  final. AUD=240
  pop (150, ENG)  stale, skipped
  pop (240, AUD)  final. PRK=310
  pop (300, MED)  final.
  pop (310, PRK)  final.
  pop (320, DRM)  final.

vertex  dist(m)  route
  LIB        0  LIB
  CAF      120  LIB-CAF
  ADM       80  LIB-ADM
  ENG      140  LIB-ADM-ENG
  DRM      320  LIB-CAF-DRM
  MED      300  LIB-CAF-MED
  AUD      240  LIB-ADM-ENG-AUD
  PRK      310  LIB-ADM-ENG-AUD-PRK
  ANX   unreachable
  WHS   unreachable
```

Jejak langkahnya adalah **Tabel 13.4**, termasuk entri basi untuk ENG pada langkah 5, dan tabel rutenya adalah **Gambar 13.12**. Di Java (**Kode 13.17**), `PriorityQueue` sudah merupakan *min-heap*; ia diberi pembanding atas *record* `Item` yang membandingkan jarak lebih dahulu dan nomor simpul kemudian. Di Python, `heapq` menjaga list biasa dalam urutan *heap*, dan *tuple* `(distance, vertex)` dibandingkan persis seperti pasangan di C++.

Kode 13.17 `Dijkstra.java`: algoritma Dijkstra dengan `java.util.PriorityQueue`.

```
// Dijkstra.java -- shortest walking distances from the main library,
// with a priority queue (java.util.PriorityQueue is a min-heap).
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.PriorityQueue;

public class Dijkstra {
    static final int V = 10;
    static final String[] CODE = {"LIB", "CAF", "ADM", "ENG", "DRM",
                                  "MED", "AUD", "PRK", "ANX", "WHS"};
    static final int[][] PATHS = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                                  {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                                  {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};
    static final int INF = Integer.MAX_VALUE;

    record Nbr(int to, int w) {}
    record Item(int dist, int v) {}                 // (tentative distance, vertex)

    static void dijkstra(List<List<Nbr>> adj, int s, int[] dist, int[] parent) {
        Arrays.fill(dist, INF);
        Arrays.fill(parent, -1);
        boolean[] done = new boolean[V];
        PriorityQueue<Item> pq = new PriorityQueue<>((a, b) ->
                a.dist() != b.dist() ? Integer.compare(a.dist(), b.dist())
                                     : Integer.compare(a.v(), b.v()));
        dist[s] = 0;
        pq.add(new Item(0, s));
        while (!pq.isEmpty()) {
            Item it = pq.poll();
            int du = it.dist(), u = it.v();
            if (done[u]) {                          // an old, longer entry
                System.out.printf("  pop (%3d, %s)  stale, skipped%n", du, CODE[u]);
                continue;
            }
            done[u] = true;                         // du is now final
            System.out.printf("  pop (%3d, %s)  final.", du, CODE[u]);
            for (Nbr e : adj.get(u)) {
                if (!done[e.to()] && du + e.w() < dist[e.to()]) {   // relax
                    dist[e.to()] = du + e.w();
                    parent[e.to()] = u;
                    pq.add(new Item(dist[e.to()], e.to()));
                    System.out.printf(" %s=%d", CODE[e.to()], dist[e.to()]);
                }
            }
```

Kode 13.17 (Lanjutan) *Dijkstra.java: algoritma Dijkstra dengan java.util.PriorityQueue.*

```
            System.out.println();
        }
    }

    public static void main(String[] args) {
        List<List<Nbr>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
        for (int[] p : PATHS) {
            adj.get(p[0]).add(new Nbr(p[1], p[2]));
            adj.get(p[1]).add(new Nbr(p[0], p[2]));
        }
        int[] dist = new int[V], parent = new int[V];
        System.out.println("Dijkstra from LIB (relaxed distances after each pop)");
        dijkstra(adj, 0, dist, parent);

        System.out.println("\nvertex  dist(m)  route");
        for (int v = 0; v < V; v++) {
            if (dist[v] == INF) {
                System.out.printf("  %s   unreachable%n", CODE[v]);
                continue;
            }
            ArrayDeque<Integer> path = new ArrayDeque<>();
            for (int x = v; x != -1; x = parent[x]) path.addFirst(x);
            StringBuilder sb = new StringBuilder();
            for (int x : path)
                sb.append(sb.length() > 0 ? "-" : "").append(CODE[x]);
            System.out.printf("  %s  %7d  %s%n", CODE[v], dist[v], sb);
        }
    }
}
```

Kode 13.18 `dijkstra.py`: algoritma Dijkstra dengan modul `heapq`.

```
# dijkstra.py -- shortest walking distances from the main library,
# with a priority queue (the heapq module keeps a list as a min-heap).
import heapq
from collections import deque

V = 10
CODE = ["LIB", "CAF", "ADM", "ENG", "DRM", "MED", "AUD", "PRK", "ANX", "WHS"]
PATHS = [(0, 1, 120), (0, 2, 80), (0, 3, 150), (1, 4, 200), (1, 5, 180),
         (2, 3, 60), (3, 6, 100), (4, 5, 90), (5, 6, 160), (6, 7, 70),
         (8, 9, 50)]
INF = float("inf")


def dijkstra(adj, s):
    dist = [INF] * V
    parent = [-1] * V
    done = [False] * V
    dist[s] = 0
    pq = [(0, s)]                           # (tentative distance, vertex)
    while pq:
        du, u = heapq.heappop(pq)
        if done[u]:                         # an old, longer entry: ignore it
            print(f"  pop ({du:3}, {CODE[u]})  stale, skipped")
            continue
        done[u] = True                      # du is now final
        line = f"  pop ({du:3}, {CODE[u]})  final."
        for v, w in adj[u]:
            if not done[v] and du + w < dist[v]:     # relax u -> v
                dist[v] = du + w
                parent[v] = u
                heapq.heappush(pq, (dist[v], v))
                line += f" {CODE[v]}={dist[v]}"
        print(line)
    return dist, parent


def main():
    adj = [[] for _ in range(V)]
    for u, v, w in PATHS:
        adj[u].append((v, w))
        adj[v].append((u, w))
    print("Dijkstra from LIB (relaxed distances after each pop)")
    dist, parent = dijkstra(adj, 0)
```

Kode 13.18 (Lanjutan) *dijkstra.py: algoritma Dijkstra dengan modul heapq.*

```
    print("\nvertex  dist(m)  route")
    for v in range(V):
        if dist[v] == INF:
            print(f"  {CODE[v]}   unreachable")
            continue
        path, x = deque(), v
        while x != -1:
            path.appendleft(CODE[x])
            x = parent[x]
        print(f"  {CODE[v]}  {dist[v]:7}  " + "-".join(path))


if __name__ == "__main__":
    main()
```

Keduanya mencetak keluaran yang persis sama dengan **Kode 13.16**. Ada satu perbedaan yang tak kasatmata: C++ dan Java memakai `INT_MAX` dan `Integer.MAX_VALUE` untuk ∞, yang aman di sini hanya karena baris 11 pada **Pseudokode 13.7** tidak pernah menambahkan bobot pada jarak

simpul yang belum tercapai (simpul yang diekstrak selalu berjarak berhingga). Python memakai `float("inf")`, yang $\infty + w$-nya tetap ∞ sehingga luapan tidak mungkin terjadi.

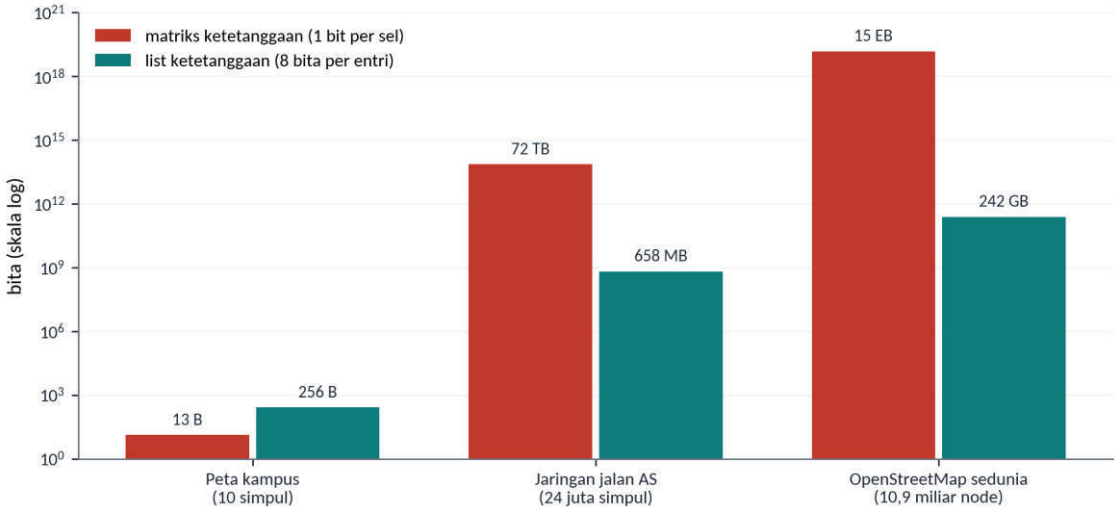
13.11 Graf pada Skala Dunia Nyata

Peta kampus mempunyai sepuluh simpul. Algoritma-algoritma bab ini dipakai setiap hari pada graf yang semiliar kali lebih besar, dan ada baiknya melihat apa yang berubah dan apa yang tidak.

13.11.1 Jaringan jalan

OpenStreetMap, peta dunia yang disunting secara kolaboratif, menyimpan geografi sebagai graf, hanya saja tidak menyebutnya demikian: *node* adalah titik bertitik koordinat, dan *way* adalah barisan *node* yang membentuk jalan, jalan setapak, sungai, dan garis tepi bangunan. Pada 23 September 2026 basis datanya memuat 10.882.374.688 *node*, 1.219.683.308 *way*, dan 14.759.380 *relation* [13]. Sebuah *way* yang melalui k *node* menyumbang $k - 1$ ruas, sehingga berkas seluruh planet itu melukiskan kira-kira $10.9 - 1.2 \approx 9.7$ miliar ruas antara *node* yang berurutan, tidak semuanya jalan raya. Penelitian perencanaan rute telah lama memakai tolok ukur yang lebih kecil dan tetap, yaitu jaringan jalan Amerika Serikat yang disiapkan untuk *9th DIMACS Implementation Challenge* pada tahun 2006: 23.947.347 simpul (persimpangan jalan) dan 58.333.344 busur [14]. Tolok ukur itu bersifat historis, tetapi karena tetap, ia memungkinkan algoritma-algoritma yang terbit terpaut dua puluh tahun dibandingkan pada graf yang persis sama.

Kedua graf itu sangat jarang. Dalam jaringan DIMACS, rata-rata derajat-keluarnya adalah 58.333.344 dibagi 23.947.347, kira-kira 2,4, dan menurut Persamaan (13.5) kerapatannya sekitar $2 \times 29.2 \times 10^6/(23.9 \times 10^6)^2 \approx 10^{-7}$. Akibatnya bagi representasi sangat dramatis, dan **Gambar 13.13** memplotnya pada skala logaritmik, dengan asumsi yang paling murah hati bagi matriks (satu bit per sel, tanpa bobot) dan list ketetanggaan ringkas yang sederhana (8 bita per entri list dan per simpul).



Gambar 13.13 Memori untuk matriks ketetanggaan (satu bit per sel) dan list ketetanggaan (8 bita per entri) tiga graf jalan: peta kampus, jaringan DIMACS AS (tolok ukur 2006), dan OpenStreetMap sedunia (*node* dan ruas per 23 September 2026). Skala logaritmik.

Untuk jaringan AS, menurut Persamaan (13.6), matriksnya memerlukan $(2.39 \times 10^7)^2 \approx 5.7 \times 10^{14}$ sel, sekitar 72 terabita bahkan dengan satu bit per sel, sedangkan list ketetanggaannya memerlukan sekitar 658 megabita, yang muat di memori sebuah ponsel. Untuk OpenStreetMap sedunia, matriksnya akan memerlukan sekitar 1.2×10^{20} sel, kira-kira 15 eksabita, melebihi

seluruh penyimpanan sebuah pusat data besar, sedangkan list-nya memerlukan sekitar 240 gigabita, sebuah server besar yang biasa saja. Pada skala ini matriks bukan sekadar lebih lambat; ia tidak mungkin ada. **Tabel 13.5** mengumpulkan angka-angkanya.

Tabel 13.5 Graf-graf nyata dan memori yang diperlukan dua representasi utamanya. Banyaknya ruas OpenStreetMap ditaksir sebagai banyaknya *node* dikurangi banyaknya *way*.

Graf	Simpul	Sisi	Rata-rata derajat	Matriks (1 bit per sel)	List ketetanggaan
Peta kampus (Subbab 13.4)	10	11	2,2	13 B	256 B
Jalan AS DIMACS, 2006 [14]	23.947.347	58.333.344 busur	2,4 (keluar)	72 TB	658 MB
OpenStreetMap sedunia, Sep. 2026 [13]	10.882.374.688 *node*	≈ 9,66 miliar ruas	≈ 1,8	15 EB	242 GB
Keluarga aplikasi Meta, Jun. 2026 [15]	3,60 miliar orang aktif harian	tidak dipublikasikan	tidak dipublikasikan	≈ 1,6 EB	bergantung pada banyaknya sisi

Bagaimana dengan waktu? BFS, DFS, dan algoritma berwaktu linear lainnya dalam bab ini menyentuh setiap simpul dan sisi sebanyak konstanta kali, sehingga pada jaringan AS mereka memerlukan sekitar 10^8 langkah dasar, jauh di bawah satu detik pada laptop masa kini. Algoritma Dijkstra, menurut Persamaan (13.11), hanya lebih lambat sebesar faktor logaritmik: satu kueri dari satu persimpangan ke semua persimpangan lain memakan beberapa detik. Itu cepat untuk satu pertanyaan, tetapi layanan navigasi menjawab jutaan pertanyaan per menit dan harus membalas dalam hitungan milidetik. Karena itu perencana rute sungguhan *memproses lebih dahulu* jaringan jalannya satu kali, misalnya dengan *contraction hierarchies*, yang menambahkan sisi-sisi pintas sehingga Dijkstra yang sedikit dimodifikasi dapat melompati sebagian besar graf, lalu menjawab setiap kueri ribuan kali lebih cepat daripada Dijkstra biasa [16]. Namun, di bawah jalan-jalan pintas itu, algoritmanya tetaplah Dijkstra dari Subbab 13.10, dan grafnya tetaplah list ketetanggaan.

13.11.2 Graf sosial

Jejaring sosial adalah graf yang simpulnya orang dan sisinya pertemanan (tak berarah) atau "mengikuti" (berarah). Yang terbesar dioperasikan oleh Meta, yang keluarga aplikasinya (Facebook, Instagram, WhatsApp, dan Messenger) mempunyai rata-rata 3,60 miliar *orang aktif harian* (*daily active people*) pada Juni 2026, menurut laporannya untuk kuartal kedua 2026 [15]. Graf pertemanan di balik angka-angka itu tidak dipublikasikan, tetapi sebuah pengukuran historis yang terkenal memberi gambaran tentang bentuknya: pada Februari 2016, para peneliti Facebook menghitung bahwa jarak rata-rata antara dua dari 1,6 miliar penggunanya saat itu adalah 4,57 langkah, yaitu 3,57 orang perantara, "tiga setengah derajat keterpisahan" [17]. Menghitung jarak rata-rata atas miliaran orang pada prinsipnya memerlukan BFS dari setiap simpul, dan para peneliti itu memakai teknik penghitungan hampiran, justru karena waktu linear per sumber pun sudah terlalu mahal bila sumbernya miliaran.

Graf sosial berbeda dari jaringan jalan dalam satu hal penting: derajatnya sangat tidak merata. Kebanyakan orang mempunyai beberapa ratus koneksi, tetapi segelintir akun mempunyai ratusan juta pengikut. Untuk simpul semacam itu, uji sisi $O(\deg(u))$ pada list ketetanggaan di **Tabel 13.1** menjadi mahal, dan sistem sungguhan menyimpan list tetangga simpul-simpul besar dalam keadaan terurut atau ter-*hash* (Bab 15) agar pertanyaan "apakah u mengikuti v?" terjawab

tanpa pemindaian. Segala hal lain dalam bab ini berlaku tanpa perubahan: saran "orang yang mungkin Anda kenal" berawal dari simpul-simpul berjarak 2 dalam sebuah BFS, dan apakah dua akun terhubung sama sekali adalah pertanyaan komponen.

13.11.3 Apa yang berubah pada skala besar, dan apa yang tidak

Tiga pelajaran dari angka-angka ini berlaku untuk setiap program graf yang Anda tulis.

- **Representasi dipilih berdasarkan E, bukan V.** Graf nyata itu jarang, sehingga list ketetanggaan (atau bentuk lariknya yang ringkas, yang menyambung semua list menjadi satu larik beserta indeks tempat setiap list dimulai) adalah pilihan bawaan.
- **Waktu linear adalah sasarannya.** Algoritma $\Theta(V + E)$ dapat berjalan pada sebuah planet; algoritma $\Theta(V^2)$ tidak, berapa pun konstantanya.
- **Kedalaman rekursi menjadi batas yang nyata.** DFS pada jaringan jalan dapat masuk sedalam jutaan simpul, jauh melampaui *call stack* mana pun, sehingga kode produksi memakai *stack* eksplisit dari Subbab 13.7.5.

13.12 Memilih Algoritma: Ongkos Sekilas

Tabel 13.6 merangkum algoritma-algoritma bab ini. Semua ongkos mengasumsikan list ketetanggaan; dengan matriks ketetanggaan, gantilah setiap $V + E$ dengan V^2.

Tabel 13.6 Algoritma-algoritma graf dalam bab ini, beserta struktur yang diandalkan masing-masing dan ongkosnya pada list ketetanggaan.

Pertanyaan	Algoritma	Struktur yang diandalkan	Waktu	Memori tambahan
Apa saja yang dapat dicapai dari s?	BFS atau DFS	*queue* atau *stack*	$\Theta(V + E)$	$\Theta(V)$
Sisi paling sedikit dari s ke setiap simpul	BFS (**Pseudokode 13.1**)	*queue* FIFO	$\Theta(V + E)$	$\Theta(V)$
Waktu penemuan dan selesai, hutan DFS	DFS (**Pseudokode 13.2**, **Pseudokode 13.3**)	rekursi atau *stack* eksplisit	$\Theta(V + E)$	$\Theta(V)$
Apakah graf tak berarah mempunyai siklus?	DFS, sisi balik ke bukan-orang-tua	*stack*	$\Theta(V + E)$	$\Theta(V)$
Komponen terhubung	satu BFS atau DFS per komponen (**Pseudokode 13.4**)	*queue* atau *stack*	$\Theta(V + E)$	$\Theta(V)$
Urutan yang menghormati semua ketergantungan	Kahn (**Pseudokode 13.5**) atau waktu selesai DFS (**Pseudokode 13.6**)	*queue*, atau *stack*	$\Theta(V + E)$	$\Theta(V)$
Apakah graf berarah mempunyai siklus?	Kahn mengeluarkan kurang dari V, atau DFS menemukan simpul abu-abu	*queue*, atau tiga warna	$\Theta(V + E)$	$\Theta(V)$
Rute berbobot terpendek, bobot ≥ 0	Dijkstra (**Pseudokode 13.7**)	*min-priority queue*	$O((V + E)\log V)$	$O(V + E)$

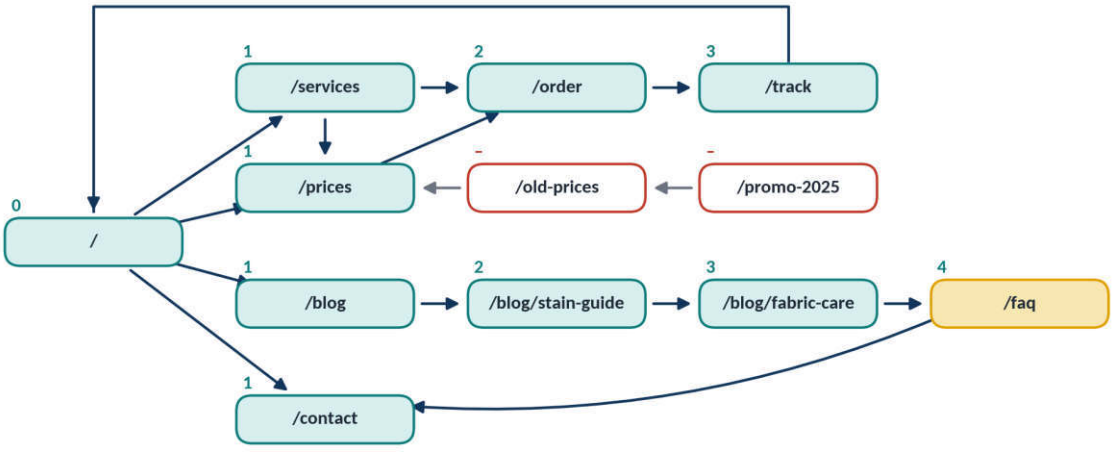
Kolom tengah itu menceritakan kisah seluruh buku dalam bentuk mini. Gelung yang sama, "ambil sebuah simpul dari sebuah wadah lalu lihat tetangga-tetangganya", menjadi penelusuran melebar dengan *queue* dari Bab 5, penelusuran mendalam dengan *stack* dari Bab 5 atau rekursi dari Bab 10, dan algoritma Dijkstra dengan *priority queue* dari Bab 14. Wadahnya menentukan urutan penelusuran, dan urutan itu menentukan apa yang dihitung oleh penelusuran.

13.13 Profitina dalam Praktik

Profitina adalah platform visibilitas AI (GEO): ia memindai situs web sebuah merek dan mengukur seberapa siap situs itu ditemukan dan dikutip oleh mesin penjawab AI, lalu membantu memperbaiki apa yang ditemukannya [18]. Sebuah situs web adalah graf berarah: setiap halaman adalah simpul, dan setiap *hyperlink* dari halaman u ke halaman v adalah panah (u, v). Perayap (*crawler*), baik jenis mesin pencari klasik maupun yang memasok mesin penjawab AI, menemukan sebuah situs dengan mengikuti panah-panah itu dari halaman-halaman yang sudah dikenalnya; sejak masa awal pencarian web, hal ini berarti struktur tautan sebuah situs menentukan apa yang dapat ditemukan sama sekali [19]. Karena itu, beberapa pertanyaan yang diajukan pemindai situs persis merupakan pertanyaan-pertanyaan bab ini; kita melukiskan bentuk masing-masing dan kelas algoritma yang cocok, bukan implementasi internal produknya.

- **Dapatkah setiap halaman dicapai dari halaman beranda?** Ini adalah keterjangkauan dalam graf berarah, dijawab oleh satu BFS dari halaman beranda. Halaman yang tercantum dalam *sitemap* situs tetapi tidak dapat dicapai lewat tautan adalah *halaman yatim* (*orphan page*): pengunjung tidak dapat mengekliknya untuk sampai ke sana, dan perayap yang mengikuti tautan pun tidak akan menemukannya.
- **Berapa klik kedalaman setiap halaman?** Jarak langkah dari halaman beranda, yang dihitung BFS sebagai hasil sampingan. Halaman yang terkubur banyak klik di dalam cenderung lebih jarang dikunjungi, dan lebih jarang dirayapi, sehingga kedalamannya layak dilaporkan.
- **Halaman mana yang paling banyak ditautkan?** Derajat-masuk setiap simpul, dihitung dalam satu lintasan atas semua panah. Halaman berderajat-masuk 0, selain halaman beranda, menurut definisinya adalah halaman yatim.
- **Apakah tautan-tautannya membentuk putaran?** Tentu saja, dan itu tidak apa-apa: situs web bukan DAG, dan perayap harus menandai halaman yang sudah dikunjungi persis seperti BFS, atau ia akan mengikuti tautan menu kembali ke halaman beranda selamanya.

Versi yang dapat dijalankan dari skenario ini memakai situs web Profitina Laundry, studi kasus yang dipakai di seluruh buku ini, dengan dua belas halaman yang diambil dari *sitemap*-nya dan lima belas tautan. **Gambar 13.14** menggambar graf situs itu beserta kedalaman klik setiap halaman, dan **Kode 13.19** menghitungnya.



Gambar 13.14 Situs web Profitina Laundry sebagai graf berarah. Angka di atas setiap halaman adalah kedalaman kliknya dari halaman beranda; /faq (kuning tua) berada empat klik di dalam, dan dua halaman bergaris tepi merah adalah halaman yatim yang tidak dicapai lintasan tautan mana pun.

Kode 13.19 `laundry_site.cpp`: kedalaman klik, tautan masuk, dan halaman yatim situs web Profitina Laundry, dengan penelusuran melebar.

```cpp
// laundry_site.cpp -- the Profitina Laundry website as a directed graph:
// click depth from the home page (BFS), orphan pages and inbound links.
#include <cstdio>
#include <queue>
#include <string>
#include <vector>

const std::vector<std::string> PAGE = {
    "/",          "/services",   "/prices",           "/order",
    "/track",     "/blog",       "/blog/stain-guide", "/blog/fabric-care",
    "/faq",       "/contact",    "/promo-2025",       "/old-prices"};
// Link u -> v: page u contains a hyperlink to page v.
const int LINKS[][2] = {{0, 1}, {0, 2}, {0, 5},  {0, 9},  {1, 2}, {1, 3},
                        {2, 3}, {3, 4}, {4, 0},  {5, 6},  {6, 7}, {7, 8},
                        {8, 9}, {10, 11}, {11, 2}};
const int MAX_DEPTH = 3;   // a page more than 3 clicks from home is hard to find

int main() {
    const int n = static_cast<int>(PAGE.size());
    std::vector<std::vector<int>> out(n);
    std::vector<int> inbound(n, 0);
    for (const auto &l : LINKS) {
        out[l[0]].push_back(l[1]);
        ++inbound[l[1]];
    }

    std::vector<int> depth(n, -1);          // BFS from the home page
    std::queue<int> q;
    depth[0] = 0;
    q.push(0);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v : out[u])
            if (depth[v] == -1) {
                depth[v] = depth[u] + 1;
                q.push(v);
            }
    }
```

Kode 13.19 (Lanjutan) *laundry_site.cpp: kedalaman klik, tautan masuk, dan halaman yatim situs web Profitina Laundry, dengan penelusuran melebar.*

```cpp
    std::printf("%-20s %6s %8s\n", "page", "clicks", "inbound");
    for (int v = 0; v < n; ++v) {
        if (depth[v] == -1) std::printf("%-20s %6s %8d\n", PAGE[v].c_str(), "-",
                                        inbound[v]);
        else std::printf("%-20s %6d %8d\n", PAGE[v].c_str(), depth[v], inbound[v]);
    }
    std::printf("\nOrphan pages (in the sitemap, unreachable from /):");
    for (int v = 0; v < n; ++v)
        if (depth[v] == -1) std::printf(" %s", PAGE[v].c_str());
    std::printf("\nPages deeper than %d clicks:", MAX_DEPTH);
    for (int v = 0; v < n; ++v)
        if (depth[v] > MAX_DEPTH) std::printf(" %s", PAGE[v].c_str());
    std::printf("\n");
    return 0;
}
```

```
page                 clicks  inbound
/                         0        1
/services                 1        1
/prices                   1        3
/order                    2        2
/track                    3        1
/blog                     1        1
/blog/stain-guide         2        1
/blog/fabric-care         3        1
/faq                      4        1
/contact                  1        2
/promo-2025               -        0
/old-prices               -        1

Orphan pages (in the sitemap, unreachable from /): /promo-2025 /old-prices
Pages deeper than 3 clicks: /faq
```

Hasilnya terbaca seperti audit situs. Halaman `/prices` mempunyai tiga tautan masuk, terbanyak di antara semua halaman, karena ia dicapai dari halaman beranda, dari `/services`, dan dari halaman harga lama. Halaman FAQ berada empat klik di dalam, satu lebih dalam daripada ambang tiga yang dipakai program, karena satu-satunya lintasan ke sana melewati seluruh blog; satu tautan dari halaman beranda atau dari `/services` akan membawanya ke kedalaman 1 atau 2. Halaman `/promo-2025` dan `/old-prices` adalah halaman yatim: promosi lama masih bertaut ke daftar harga lama, dan daftar harga lama bahkan bertaut ke harga yang berlaku, tetapi tidak ada yang bertaut *ke* promosi itu, sehingga BFS dari `/` tidak pernah sampai ke sana. Perbaikannya adalah keputusan pemilik situs, bukan keputusan program: tautkan halaman-halaman itu dari suatu tempat, atau pensiunkan. Program hanya membuat strukturnya terlihat, dan itulah pembagian kerja yang tepat antara sebuah alat dan penggunanya. Versi Java dan Python, **Kode 13.20** dan **Kode 13.21**, mencetak audit yang sama.

Kode 13.20 `LaundrySite.java`: audit situs web dalam Java.

```java
// LaundrySite.java -- the Profitina Laundry website as a directed graph:
// click depth from the home page (BFS), orphan pages and inbound links.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class LaundrySite {
    static final String[] PAGE = {
        "/",      "/services", "/prices",           "/order",
        "/track", "/blog",     "/blog/stain-guide", "/blog/fabric-care",
        "/faq",   "/contact",  "/promo-2025",       "/old-prices"};
    // Link u -> v: page u contains a hyperlink to page v.
    static final int[][] LINKS = {{0, 1}, {0, 2}, {0, 5}, {0, 9}, {1, 2}, {1, 3},
                                  {2, 3}, {3, 4}, {4, 0}, {5, 6}, {6, 7}, {7, 8},
                                  {8, 9}, {10, 11}, {11, 2}};
    static final int MAX_DEPTH = 3;

    public static void main(String[] args) {
        int n = PAGE.length;
        List<List<Integer>> out = new ArrayList<>();
        for (int i = 0; i < n; i++) out.add(new ArrayList<>());
        int[] inbound = new int[n];
        for (int[] l : LINKS) {
            out.get(l[0]).add(l[1]);
            inbound[l[1]]++;
        }

        int[] depth = new int[n];                    // BFS from the home page
        Arrays.fill(depth, -1);
        ArrayDeque<Integer> q = new ArrayDeque<>();
        depth[0] = 0;
        q.addLast(0);
        while (!q.isEmpty()) {
            int u = q.pollFirst();
            for (int v : out.get(u))
                if (depth[v] == -1) {
                    depth[v] = depth[u] + 1;
                    q.addLast(v);
                }
        }
```

Kode 13.20 (Lanjutan) *LaundrySite.java: audit situs web dalam Java.*

```java
        System.out.printf("%-20s %6s %8s%n", "page", "clicks", "inbound");
        for (int v = 0; v < n; v++)
            System.out.printf("%-20s %6s %8d%n", PAGE[v],
                    depth[v] == -1 ? "-" : String.valueOf(depth[v]), inbound[v]);
        System.out.print("\nOrphan pages (in the sitemap, unreachable from /):");
        for (int v = 0; v < n; v++)
            if (depth[v] == -1) System.out.print(" " + PAGE[v]);
        System.out.printf("%nPages deeper than %d clicks:", MAX_DEPTH);
        for (int v = 0; v < n; v++)
            if (depth[v] > MAX_DEPTH) System.out.print(" " + PAGE[v]);
        System.out.println();
    }
}
```

Kode 13.21 `laundry_site.py`: audit situs web dalam Python.

```
# laundry_site.py -- the Profitina Laundry website as a directed graph:
# click depth from the home page (BFS), orphan pages and inbound links.
from collections import deque

PAGE = ["/", "/services", "/prices", "/order",
        "/track", "/blog", "/blog/stain-guide", "/blog/fabric-care",
        "/faq", "/contact", "/promo-2025", "/old-prices"]
# Link (u, v): page u contains a hyperlink to page v.
LINKS = [(0, 1), (0, 2), (0, 5), (0, 9), (1, 2), (1, 3),
         (2, 3), (3, 4), (4, 0), (5, 6), (6, 7), (7, 8),
         (8, 9), (10, 11), (11, 2)]
MAX_DEPTH = 3        # a page more than 3 clicks from home is hard to find


def main():
    n = len(PAGE)
    out = [[] for _ in range(n)]
    inbound = [0] * n
    for u, v in LINKS:
        out[u].append(v)
        inbound[v] += 1

    depth = [-1] * n                          # BFS from the home page
    depth[0] = 0
    q = deque([0])
    while q:
        u = q.popleft()
        for v in out[u]:
            if depth[v] == -1:
                depth[v] = depth[u] + 1
                q.append(v)

    print(f"{'page':<20} {'clicks':>6} {'inbound':>8}")
    for v in range(n):
        shown = "-" if depth[v] == -1 else str(depth[v])
        print(f"{PAGE[v]:<20} {shown:>6} {inbound[v]:>8}")
    print("\nOrphan pages (in the sitemap, unreachable from /):"
          + "".join(" " + PAGE[v] for v in range(n) if depth[v] == -1))
    print(f"Pages deeper than {MAX_DEPTH} clicks:"
          + "".join(" " + PAGE[v] for v in range(n) if depth[v] > MAX_DEPTH))


if __name__ == "__main__":
    main()
```

Graf juga muncul di bagian lain produk itu. Komunitas *real-time* Profitina memungkinkan pengguna mengikuti kanal, yang merupakan graf berarah dari pengguna ke kanal, dan penghitung "N baru" di dasbor dihitung tepat di sepanjang panah-panah itu. Dan saran Content Engine tentang tautan internal pada hakikatnya adalah saran tentang derajat-masuk dan kedalaman yang dihitung di atas.

13.14 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Lupa menandai simpul yang sudah dikunjungi. Penelusuran tanpa penanda "sudah dikunjungi" berputar mengelilingi siklus pertama selamanya. Pohon tidak pernah memerlukan penanda (Bab 11); graf selalu memerlukannya.

Menandai simpul pada BFS ketika keluar dari *queue* alih-alih ketika masuk. Hasilnya tetap benar, tetapi sebuah simpul dapat dimasukkan sekali untuk setiap tetangganya, dan *queue* dapat membengkak sampai $|E|$ entri. Tandai saat masuk, seperti baris 11 pada **Pseudokode 13.1** (Subbab 13.6.2).

Menyimpan sisi tak berarah hanya ke satu arah. `adj[u].push_back(v)` tanpa `adj[v].push_back(u)` diam-diam mengubah graf menjadi graf berarah, dan penelusuran lalu melewatkan simpul-simpul. Pemeriksaan jabat tangan di Subbab 13.5.5, jumlah panjang list sama dengan $2|E|$, menangkapnya.

Memakai matriks untuk graf besar yang jarang. Ia memboroskan memori $\Theta(V^2)$ dan mengubah setiap penelusuran menjadi berwaktu $\Theta(V^2)$ (**Tabel 13.1**, **Gambar 13.13**).

Membangun matriks Python sebagai `[[0] * n] * n`. Semua barisnya adalah objek list yang sama; menulis satu sel mengubah satu kolom penuh. Pakailah `[[0] * n for _ in range(n)]` (Subbab 13.5.6).

Memakai `list.pop(0)` sebagai *queue* di Python. Setiap panggilan berongkos $\Theta(n)$, sehingga BFS menjadi kuadratik. Pakailah `collections.deque` (Subbab 13.6.7).

Mengharapkan BFS menemukan rute terpendek dalam graf berbobot. BFS meminimumkan banyaknya sisi, bukan jumlah bobotnya; pada peta kampus ia memilih rute yang 10 meter lebih panjang daripada rute terpendek (Subbab 13.10.1).

Memakai algoritma Dijkstra dengan bobot negatif. Aturan "diekstrak berarti final" menjadi salah, dan algoritma dapat mengembalikan jarak yang salah tanpa peringatan apa pun (Subbab 13.10.5).

Menganggap setiap tetangga yang sudah dikunjungi sebagai siklus dalam graf tak berarah. Orang tua selalu merupakan tetangga yang sudah dikunjungi; kecualikan ia, atau setiap sisi akan tampak sebagai siklus (Subbab 13.7.6).

Memakai dua warna untuk mendeteksi siklus dalam graf berarah. Panah menuju simpul yang sudah selesai (hitam) bukan siklus; hanya panah menuju simpul abu-abu yang merupakan siklus (Subbab 13.9.4).

Berekursi terlalu dalam. DFS rekursif meluapkan *call stack* pada lintasan panjang, dan Python secara bawaan berhenti pada 1000 tingkat. Pakailah *stack* eksplisit **Pseudokode 13.3** untuk graf besar.

Bergantung pada satu urutan BFS atau DFS tertentu. Urutan kunjungan bergantung pada urutan list ketetanggaan, yang bergantung pada masukan. Jarak, komponen, dan ada-tidaknya siklus tidak bergantung padanya, tetapi pohon dan cap waktu yang persis bergantung padanya; uji yang mengharapkan satu urutan yang persis sesungguhnya menguji berkas masukan sama banyaknya dengan menguji algoritmanya.

13.15 Ringkasan Bab

- **Graf** $G = (V, E)$ adalah himpunan simpul dan himpunan sisi (Persamaan (13.1)); sisinya dapat tak berarah atau berarah dan dapat membawa bobot. Graf merampatkan list dan pohon: pohon adalah graf terhubung yang asiklik.
- **Derajat** sebuah simpul menghitung sisi-sisinya; **lema jabat tangan** menyatakan bahwa jumlah derajat sama dengan $2|E|$ (Persamaan (13.2)), dan dalam graf berarah jumlah derajat-masuk serta jumlah derajat-keluar masing-masing sama dengan $|E|$ (Persamaan (13.3)). Graf sederhana mempunyai paling banyak $|V|(|V|-1)/2$ sisi (Persamaan (13.4)); graf nyata itu jarang (Persamaan (13.5)).
- **Lintasan, siklus, komponen.** Keterjangkauan memecah graf tak berarah menjadi **komponen terhubung**; **DAG** adalah graf berarah tanpa siklus berarah.

- **Representasi. Matriks ketetanggaan** menjawab pertanyaan sisi dalam $O(1)$ tetapi memerlukan memori $\Theta(V^2)$ dan waktu $\Theta(V)$ untuk mendaftar tetangga; **list ketetanggaan** memerlukan memori $\Theta(V+E)$ dan mendaftar tetangga u dalam $\Theta(\deg(u))$; **list sisi** adalah format masukan yang alami (Persamaan (13.6), **Tabel 13.1**). Penelusuran menghendaki list.
- **BFS** memakai *queue*, mengunjungi simpul dalam cincin-cincin berjarak langkah sama, dan menghitung jarak serta rute dengan langkah paling sedikit dalam $\Theta(V+E)$ (**Pseudokode 13.1**, Persamaan (13.8)); setiap sisi memenuhi $d[v] \le d[u]+1$ (Persamaan (13.7)).
- **DFS** memakai rekursi atau *stack* eksplisit berisi pasangan (simpul, tetangga berikutnya), mencatat waktu penemuan dan waktu selesai yang selang-selangnya bersarang seperti tanda kurung (Persamaan (13.9)), dan berjalan dalam $\Theta(V+E)$. Dalam graf tak berarah, sisi balik ke simpul yang bukan orang tua berarti siklus.
- **Komponen terhubung** ditemukan dengan satu penelusuran untuk setiap simpul yang belum berlabel, dalam total $\Theta(V+E)$ (**Pseudokode 13.4**).
- **Pengurutan topologis** menyusun DAG sehingga setiap panah menunjuk ke depan: **algoritma Kahn** berulang-ulang membuang simpul berderajat-masuk 0, dan **metode DFS** mendaftar simpul menurut waktu selesai yang menurun. Kahn yang mengeluarkan kurang dari $|V|$ simpul, atau DFS yang bertemu simpul abu-abu, membuktikan adanya siklus berarah.
- **Algoritma Dijkstra** menemukan rute berbobot terpendek bila semua bobot tidak negatif, dengan berulang-ulang memfinalkan simpul terdekat dan **merelaksasi** sisi-sisinya (Persamaan (13.10)); dengan *priority queue* berbasis *binary heap* ia berjalan dalam $O((V+E)\log V)$ (Persamaan (13.11)). Bab 14 membangun *heap* itu.
- **Pada skala besar**, jaringan jalan berisi puluhan juta sampai miliaran simpul dan jejaring sosial berisi miliaran orang ditangani dengan algoritma-algoritma yang persis sama pada list ketetanggaan; matriks dan rekursi yang dalam tidak mungkin dipakai.

13.16 Soal Latihan

13.1 Untuk graf tak berarah dengan $V = \{1,2,3,4,5,6\}$ dan sisi-sisi $\{1,2\}$, $\{1,3\}$, $\{2,3\}$, $\{3,4\}$, $\{5,6\}$: (a) sebutkan derajat setiap simpul dan periksalah lema jabat tangan; (b) sebutkan sebuah siklus; (c) sebutkan komponen-komponen terhubungnya; (d) hitunglah kerapatannya.

13.2 Dapatkah graf tak berarah sederhana mempunyai barisan derajat (a) 3, 3, 3, 3, 3; (b) 3, 3, 3, 3; (c) 5, 1, 1, 1? Untuk masing-masing, gambarlah graf semacam itu atau jelaskan mengapa tidak ada.

13.3 Sebuah graf mempunyai $|V| = 10^5$ simpul dan $|E| = 3 \times 10^5$ sisi tak berarah. Dengan Persamaan (13.6), berapa sel yang diperlukan matriks ketetanggaan, dan berapa entri yang diperlukan list ketetanggaan? Bila satu sel matriks memakan 1 bita dan satu entri list 8 bita, berapa bita yang dipakai masing-masing?

13.4 Jalankan BFS dengan tangan pada peta kampus dari PRK, dengan tetangga menurut urutan list ketetanggaan pada **Gambar 13.4**(b). Sebutkan urutan kunjungan, jarak dan orang tua setiap simpul, serta rute dengan langkah paling sedikit dari PRK ke LIB.

13.5 Ubahlah baris 10 dan 11 pada **Pseudokode 13.1** sehingga sebuah simpul baru ditandai ketika dikeluarkan (dan dilewati bila sudah bertanda). Jalankan algoritma yang telah diubah itu dari LIB pada peta kampus dan daftarlah isi *queue* sesudah setiap langkah. Simpul mana yang dimasukkan lebih dari sekali? Apakah jarak-jarak yang dihasilkan masih benar?

13.6 DFS iteratif "sederhana" melakukan *push* sumber; lalu, selama *stack* tidak kosong, ia melakukan *pop* sebuah simpul, melewatinya bila sudah dikunjungi, dan bila belum, menandainya lalu melakukan *push* tetangga-tetangganya yang belum dikunjungi dalam urutan list yang *terbalik*. Jalankan dari LIB pada peta kampus, dengan mendaftar isi *stack* sesudah setiap *pop*. Apakah ia mengunjungi simpul dalam urutan yang sama dengan **Pseudokode 13.2**? Paling banyak berapa entri yang dimuat *stack*-nya, dan bagaimana bila dibandingkan dengan **Pseudokode 13.3**?

13.7 Jalankan `DFS-VISIT` dari **Pseudokode 13.2** mulai dari ADM (hanya awal itu) pada peta kampus dan sebutkan waktu penemuan dan waktu selesai setiap simpul yang dicapainya. Gambarlah struktur tanda kurungnya.

13.8 Berapa banyak komponen terhubung pada peta kampus (tanpa jalan *shuttle*) sesudah menghapus (a) jalan setapak ENG-AUD; (b) jalan setapak AUD-PRK; (c) LIB-CAF dan MED-AUD sekaligus? Jalan setapak mana pada peta asli yang merupakan jembatan?

13.9 Berikan dua urutan topologis alur kerja buku baru yang berbeda dari kedua urutan di Subbab 13.9. Bagaimana Anda mengubah algoritma Kahn agar, setiap kali beberapa tugas tersedia, ia selalu memilih tugas yang namanya paling awal menurut abjad? Urutan apa yang lalu dihasilkannya?

13.10 Sebuah graf disebut *bipartit* bila simpul-simpulnya dapat diwarnai dengan dua warna sehingga setiap sisi menghubungkan dua simpul berwarna berbeda. Tulislah fungsi yang memutuskannya dengan BFS, dengan memberi setiap simpul yang baru ditemukan warna yang berlawanan dengan warna orang tuanya. Ujilah pada peta kampus dan pada siklus empat simpul. Jelaskan mengapa graf yang mengandung siklus berpanjang ganjil tidak pernah bipartit.

13.11 Ruang baca Pustaka Ceria digambar sebagai kisi dengan `#` adalah rak buku, `S` pintu masuk, dan `T` meja baca:

```
S..#....
.#.#.##.
.#...#..
.####.#.
......#T
```

Seorang pembaca bergerak satu sel ke atas, bawah, kiri, atau kanan setiap langkah dan tidak dapat menembus rak. Dengan memperlakukan setiap sel kosong sebagai simpul, temukan banyaknya langkah paling sedikit dari S ke T dengan BFS, dan tandai satu rute terpendek.

13.12 Tulislah fungsi yang menghitung, untuk setiap simpul sebuah DAG, banyaknya lintasan berarah yang berbeda dari sumber *s* kepadanya, dengan memproses simpul-simpul dalam urutan topologis. Berapa banyak lintasan dari *order* ke *shelve* dalam alur kerja buku baru, dan dari *order* ke *cover*?

13.13 Jalankan algoritma Dijkstra dengan tangan dari PRK pada peta kampus, dengan gaya **Tabel 13.4**. Sebutkan jarak-jarak yang dihasilkan dan rute terpendek dari PRK ke CAF. Adakah entri yang basi?

13.14 Dengan contoh tiga simpul di Subbab 13.10.5, tunjukkan langkah demi langkah bagaimana algoritma Dijkstra menghitung jarak yang salah. Kemudian tunjukkan, dengan contoh kecil, bahwa menambahkan konstanta yang sama pada setiap bobot agar semuanya tidak negatif pada umumnya *tidak* memperbaiki masalahnya.

13.15 Untuk setiap keadaan berikut, pilihlah matriks ketetanggaan, list ketetanggaan, atau list sisi, dan berikan alasannya dengan **Tabel 13.1**: (a) 200 bandara sebuah negara, dengan hampir setiap pasangan mempunyai penerbangan langsung dan program terutama menanyakan apakah

ada penerbangan langsung; (b) jalan setapak sebuah kampus dengan 3.000 gedung, dipakai untuk perencanaan rute; (c) berkas berisi 50 juta ruas jalan yang harus diurutkan menurut panjangnya; (d) graf pengikut sebuah jejaring sosial.

13.16 Euler menunjukkan bahwa graf terhubung mempunyai jalan yang memakai setiap sisi tepat satu kali jika dan hanya jika graf itu mempunyai nol atau dua simpul berderajat ganjil. (a) Daratan Königsberg mana yang memerlukan jembatan tambahan, dan paling sedikit berapa jembatan baru yang membuat jalan semacam itu mungkin? (b) Apakah peta kampus (dibatasi pada komponen utamanya) mempunyai jalan semacam itu?

13.17 Penyelesaian Soal Latihan

13.1 (a) $\deg(1) = 2$, $\deg(2) = 2$, $\deg(3) = 3$, $\deg(4) = 1$, $\deg(5) = 1$, $\deg(6) = 1$; jumlahnya $10 = 2 \times 5$. (b) 1, 2, 3, 1. (c) {1,2,3,4} dan {5,6}. (d) Menurut Persamaan (13.5), $\rho = 5/(6 \times 5/2) = 5/15 = 1/3$.

13.2 (a) Tidak mungkin: jumlah derajatnya, 15, ganjil, bertentangan dengan Persamaan (13.2). (b) Mungkin: graf lengkap dengan empat simpul, yang setiap simpulnya bertetangga dengan ketiga simpul lainnya. (c) Tidak mungkin: dalam graf sederhana dengan empat simpul, derajat paling besar 3, sehingga tidak ada simpul yang dapat berderajat 5. (Perhatikan bahwa jumlahnya, 8, genap; lema jabat tangan adalah syarat perlu, bukan syarat cukup.)

13.3 Matriks: $|V|^2 = 10^{10}$ sel, 10 GB dengan satu bita per sel. List: $|V| + 2|E| = 10^5 + 6 \times 10^5 = 7 \times 10^5$ entri, 5,6 MB dengan 8 bita per entri. Matriksnya kira-kira 1.800 kali lebih besar, dan kerapatannya hanya $3 \times 10^5/(10^5 \times 99999/2) \approx 6 \times 10^{-5}$.

13.4 *Queue* berkembang sebagai berikut: PRK; AUD; ENG, MED; MED, LIB, ADM; LIB, ADM, CAF, DRM; lalu simpul-simpul yang tersisa keluar satu per satu. Urutan kunjungan: PRK, AUD, ENG, MED, LIB, ADM, CAF, DRM. Jarak: PRK 0; AUD 1 (orang tua PRK); ENG 2 dan MED 2 (orang tua AUD); LIB 3 dan ADM 3 (orang tua ENG); CAF 3 dan DRM 3 (orang tua MED); ANX dan WHS tidak tercapai. Rute ke LIB: PRK, AUD, ENG, LIB, 3 langkah. Hasil ini diperiksa dengan program **Kode 13.6**.

13.5 Dengan penandaan saat keluar, *queue* menerima setiap tetangga yang belum *bertanda*, termasuk yang sudah menunggu. Dari LIB: sesudah LIB, *queue* berisi CAF ADM ENG; sesudah CAF, ADM ENG DRM MED; sesudah ADM (yang tetangganya, ENG, sedang menunggu tetapi belum bertanda), ENG DRM MED ENG; sesudah ENG, DRM MED ENG AUD; sesudah DRM (MED sedang menunggu, belum bertanda), MED ENG AUD MED; sesudah MED, ENG AUD MED AUD; ENG yang kedua dilewati; sesudah AUD, MED AUD PRK; lalu MED dan AUD dilewati dan PRK diproses. ENG, MED, dan AUD masing-masing dimasukkan dua kali. Jaraknya tetap benar pada eksekusi ini karena jarak sebuah simpul ditetapkan ketika ia pertama kali ditandai, dan *queue* tetap terurut menurut jarak yang tidak menurun; tetapi 11 simpul dimasukkan alih-alih 8, dan pada graf padat *queue* dapat memuat $\Theta(E)$ entri alih-alih paling banyak $|V|$.

13.6 Isi *stack* sesudah setiap *pop* (puncak di kanan): awal LIB; *pop* LIB, *push* ENG ADM CAF (urutan terbalik, sehingga CAF di puncak): ENG ADM CAF; *pop* CAF, *push* MED DRM: ENG ADM MED DRM; *pop* DRM, *push* MED: ENG ADM MED MED; *pop* MED, *push* AUD: ENG ADM MED AUD; *pop* AUD, *push* PRK dan ENG (ENG belum dikunjungi): ENG ADM MED PRK ENG; *pop* ENG, *push* ADM: ENG ADM MED PRK ADM; *pop* ADM: ENG ADM MED PRK; *pop* PRK; lalu MED, ADM, dan ENG di-*pop* dan dilewati. Urutan kunjungan LIB, CAF, DRM, MED, AUD, ENG, ADM, PRK sama dengan versi rekursif di sini, tetapi *stack*-nya memuat sampai 5 entri termasuk entri ganda (MED

dua kali, ADM dua kali, ENG dua kali), sedangkan **Pseudokode 13.3** hanya memuat rantai simpul abu-abu saat ini, masing-masing satu entri. Pada umumnya versi sederhana dapat memuat $\Theta(E)$ entri dan tidak menghasilkan waktu selesai.

13.7 ADM 1/16, LIB 2/15, CAF 3/14, DRM 4/13, MED 5/12, AUD 6/11, ENG 7/8, PRK 9/10; ANX dan WHS tidak tercapai. Tanda kurung: (ADM (LIB (CAF (DRM (MED (AUD (ENG ENG) (PRK PRK) AUD) MED) DRM) CAF) LIB) ADM). Waktu-waktu ini diperiksa dengan program.

13.8 (a) Tetap 2: AUD tetap terhubung ke kampus lewat MED. (b) 3: PRK menjadi terasing, di samping kampus utama dan ANX-WHS. (c) 3: bila LIB-CAF dan MED-AUD sama-sama hilang, tidak ada jalan setapak yang keluar dari segitiga CAF, DRM, MED, sehingga komponennya adalah {LIB,ADM,ENG,AUD,PRK}, {CAF,DRM,MED}, dan {ANX,WHS}. Jembatan pada peta asli adalah AUD-PRK dan ANX-WHS; setiap jalan setapak lainnya terletak pada sebuah siklus.

13.9 Misalnya, order, receive, catalog, label, register, inspect, stamp, cover, shelve; dan order, receive, inspect, catalog, register, label, stamp, cover, shelve. (Periksalah bahwa setiap panah pada **Gambar 13.10**(a) menunjuk ke depan dalam keduanya.) Untuk selalu memilih tugas tersedia yang paling awal menurut abjad, gantilah *queue* FIFO pada **Pseudokode 13.5** dengan *min-priority queue* berkunci nama tugas. Urutannya menjadi: order, receive, catalog (sebelum inspect), inspect, label (sebelum register dan stamp), register, stamp, cover, shelve.

13.10 Lihat `bipartite` pada **Kode 13.22**. Peta kampus tidak bipartit; siklus empat simpul bipartit, dengan warna 0, 1, 0, 1, seperti diperlihatkan keluarannya. Di sepanjang sebuah siklus warnanya harus berselang-seling, sehingga siklus kembali ke warna awalnya hanya sesudah banyaknya sisi genap; siklus berpanjang ganjil, seperti segitiga CAF, DRM, MED, akan menuntut simpul pertamanya berwarna ganda.

Kode 13.22 `exercises.cpp`: penyelesaian Soal Latihan 13.10 sampai 13.12.

```cpp
// exercises.cpp -- solutions to the programming exercises of Chapter 13:
// two-coloring, a grid maze by BFS, and counting paths in a DAG.
#include <cstdio>
#include <queue>
#include <string>
#include <vector>

using Graph = std::vector<std::vector<int>>;

Graph undirected(int n, const std::vector<std::pair<int, int>> &es) {
    Graph g(n);
    for (auto [u, v] : es) {
        g[u].push_back(v);
        g[v].push_back(u);
    }
    return g;
}

// Exercise 13.10: can the vertices be colored 0/1 so every edge joins two
// different colors? BFS gives each new vertex the color opposite its parent.
bool bipartite(const Graph &g, std::vector<int> &color) {
    color.assign(g.size(), -1);
    for (int s = 0; s < static_cast<int>(g.size()); ++s) {
        if (color[s] != -1) continue;
        std::queue<int> q;
        color[s] = 0;
        q.push(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int v : g[u]) {
                if (color[v] == -1) {
                    color[v] = 1 - color[u];
                    q.push(v);
                } else if (color[v] == color[u]) {
                    return false;          // an edge inside one color: odd cycle
                }
            }
        }
    }
    return true;
}
```

Kode 13.22 (Lanjutan) *exercises.cpp: penyelesaian Soal Latihan 13.10 sampai 13.12.*

```
// Exercise 13.11: fewest steps from S to T in a grid; '#' is a bookshelf.
int gridBfs(std::vector<std::string> &grid) {
    int R = static_cast<int>(grid.size()), C = static_cast<int>(grid[0].size());
    std::vector<std::vector<int>> dist(R, std::vector<int>(C, -1));
    std::vector<std::vector<int>> from(R, std::vector<int>(C, -1));
    std::queue<int> q;                    // a cell (r, c) is stored as r*C + c
    int s = -1, t = -1;
    for (int r = 0; r < R; ++r)
        for (int c = 0; c < C; ++c) {
            if (grid[r][c] == 'S') s = r * C + c;
            if (grid[r][c] == 'T') t = r * C + c;
        }
    dist[s / C][s % C] = 0;
    q.push(s);
    const int dr[4] = {-1, 1, 0, 0}, dc[4] = {0, 0, -1, 1};
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        int r = u / C, c = u % C;
        for (int k = 0; k < 4; ++k) {      // the four neighbors of a cell
            int nr = r + dr[k], nc = c + dc[k];
            if (nr < 0 || nr >= R || nc < 0 || nc >= C) continue;
            if (grid[nr][nc] == '#' || dist[nr][nc] != -1) continue;
            dist[nr][nc] = dist[r][c] + 1;
            from[nr][nc] = u;
            q.push(nr * C + nc);
        }
    }
    int steps = dist[t / C][t % C];
    for (int u = from[t / C][t % C]; u != -1 && u != s; u = from[u / C][u % C])
        grid[u / C][u % C] = '*';          // mark the route
    return steps;
}

// Exercise 13.12: number of different paths from s to every vertex of a DAG,
// processed in a topological order (Kahn's).
std::vector<long long> countPaths(const Graph &g, int s) {
    int n = static_cast<int>(g.size());
    std::vector<int> indeg(n, 0);
    for (int u = 0; u < n; ++u)
        for (int v : g[u]) ++indeg[v];
    std::queue<int> q;
    for (int u = 0; u < n; ++u)
        if (indeg[u] == 0) q.push(u);
    std::vector<long long> ways(n, 0);
    ways[s] = 1;
```

Kode 13.22 (Lanjutan) *exercises.cpp: penyelesaian Soal Latihan 13.10 sampai 13.12.*

```
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v : g[u]) {
            ways[v] += ways[u];            // every path to u extends to v
            if (--indeg[v] == 0) q.push(v);
        }
    }
    return ways;
}

int main() {
    Graph campus = undirected(10, {{0, 1}, {0, 2}, {0, 3}, {1, 4}, {1, 5}, {2, 3},
                                  {3, 6}, {4, 5}, {5, 6}, {6, 7}, {8, 9}});
    Graph square = undirected(4, {{0, 1}, {1, 2}, {2, 3}, {3, 0}});
    std::vector<int> color;
    std::printf("13.10 campus map bipartite? %s\n",
                bipartite(campus, color) ? "yes" : "no");
    bool sq = bipartite(square, color);
    std::printf("13.10 4-cycle bipartite? %s, colors", sq ? "yes" : "no");
    for (int c : color) std::printf(" %d", c);
    std::printf("\n");

    std::vector<std::string> room = {"S..#....",
                                     ".#.#.##.",
                                     ".#...#..",
                                     ".####.#.",
                                     "......#T"};
    int steps = gridBfs(room);
    std::printf("\n13.11 fewest steps from S to T: %d\n", steps);
    for (const std::string &row : room) std::printf("  %s\n", row.c_str());

    Graph work(9);
    const int arcs[][2] = {{0, 1}, {1, 2}, {1, 3}, {2, 4}, {3, 5},
                           {3, 7}, {4, 6}, {5, 6}, {6, 8}, {7, 8}};
    for (const auto &a : arcs) work[a[0]].push_back(a[1]);
    std::vector<long long> ways = countPaths(work, 0);
    std::printf("\n13.12 paths from order to shelve: %lld\n", ways[8]);
    std::printf("      paths from order to cover:  %lld\n", ways[6]);
    return 0;
}
```

```
13.10 campus map bipartite? no
13.10 4-cycle bipartite? yes, colors 0 1 0 1

13.11 fewest steps from S to T: 15
  S**#****
  .#*#*##*
  .#***#.*
  .####.#*
  ......#T

13.12 paths from order to shelve: 3
      paths from order to cover:  2
```

13.11 Lihat `gridBfs` pada **Kode 13.22**. Setiap sel kosong adalah simpul dengan paling banyak empat tetangga, yang dihitung dari baris dan kolomnya alih-alih disimpan, dan BFS menghasilkan 15 langkah. Rute yang ditandai `*` pada keluaran menyusuri baris teratas, turun melalui celah di kolom 2, menyusuri baris 2 sampai kolom 4, naik lagi ke baris teratas, lalu menuruni kolom paling kanan sampai T. Dua rak di baris 3 dan 4 memaksa jalan memutar itu; tidak ada jalan yang lebih pendek.

13.12 Lihat `countPaths` pada **Kode 13.22**. Dalam urutan topologis, setiap lintasan ke v berakhir dengan panah (u, v) dari simpul u yang diproses lebih dahulu, sehingga `ways[v]` adalah jumlah `ways[u]` atas semua panah yang masuk ke v. Ada 3 lintasan dari *order* ke *shelve* (lewat inspect dan stamp, lewat catalog dan label, dan lewat catalog dan register) dan 2 lintasan ke *cover*.

Versi Java dan Python dari program latihan ini, **Kode 13.23** dan **Kode 13.24**, menyusul; keduanya mencetak keluaran yang sama.

Kode 13.23 `Exercises.java`: penyelesaian Soal Latihan 13.10 sampai 13.12 dalam Java.

```java
// Exercises.java -- solutions to the programming exercises of Chapter 13:
// two-coloring, a grid maze by BFS, and counting paths in a DAG.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Exercises {
    static List<List<Integer>> undirected(int n, int[][] es) {
        List<List<Integer>> g = new ArrayList<>();
        for (int i = 0; i < n; i++) g.add(new ArrayList<>());
        for (int[] e : es) {
            g.get(e[0]).add(e[1]);
            g.get(e[1]).add(e[0]);
        }
        return g;
    }

    // Two-coloring by BFS: each new vertex gets the color opposite its parent.
    static boolean bipartite(List<List<Integer>> g, int[] color) {
        Arrays.fill(color, -1);
        for (int s = 0; s < g.size(); s++) {
            if (color[s] != -1) continue;
            ArrayDeque<Integer> q = new ArrayDeque<>();
            color[s] = 0;
            q.addLast(s);
            while (!q.isEmpty()) {
                int u = q.pollFirst();
                for (int v : g.get(u)) {
                    if (color[v] == -1) {
                        color[v] = 1 - color[u];
                        q.addLast(v);
                    } else if (color[v] == color[u]) {
                        return false;                // odd cycle
                    }
                }
            }
        }
        return true;
    }
```

Kode 13.23 (Lanjutan) *Exercises.java: penyelesaian Soal Latihan 13.10 sampai 13.12 dalam Java.*

```
    // Fewest steps from S to T in a grid; '#' is a bookshelf.
    static int gridBfs(char[][] grid) {
        int R = grid.length, C = grid[0].length;
        int[][] dist = new int[R][C], from = new int[R][C];
        for (int[] row : dist) Arrays.fill(row, -1);
        for (int[] row : from) Arrays.fill(row, -1);
        ArrayDeque<Integer> q = new ArrayDeque<>();  // cell (r, c) is r*C + c
        int s = -1, t = -1;
        for (int r = 0; r < R; r++)
            for (int c = 0; c < C; c++) {
                if (grid[r][c] == 'S') s = r * C + c;
                if (grid[r][c] == 'T') t = r * C + c;
            }
        dist[s / C][s % C] = 0;
        q.addLast(s);
        int[] dr = {-1, 1, 0, 0}, dc = {0, 0, -1, 1};
        while (!q.isEmpty()) {
            int u = q.pollFirst();
            int r = u / C, c = u % C;
            for (int k = 0; k < 4; k++) {
                int nr = r + dr[k], nc = c + dc[k];
                if (nr < 0 || nr >= R || nc < 0 || nc >= C) continue;
                if (grid[nr][nc] == '#' || dist[nr][nc] != -1) continue;
                dist[nr][nc] = dist[r][c] + 1;
                from[nr][nc] = u;
                q.addLast(nr * C + nc);
            }
        }
        int steps = dist[t / C][t % C];
        for (int u = from[t / C][t % C]; u != -1 && u != s; u = from[u / C][u % C])
            grid[u / C][u % C] = '*';                // mark the route
        return steps;
    }

    // Number of different paths from s to every vertex of a DAG (Kahn order).
    static long[] countPaths(List<List<Integer>> g, int s) {
        int n = g.size();
        int[] indeg = new int[n];
        for (int u = 0; u < n; u++)
            for (int v : g.get(u)) indeg[v]++;
        ArrayDeque<Integer> q = new ArrayDeque<>();
        for (int u = 0; u < n; u++)
            if (indeg[u] == 0) q.addLast(u);
        long[] ways = new long[n];
        ways[s] = 1;
        while (!q.isEmpty()) {
```

Kode 13.23 (Lanjutan) *Exercises.java: penyelesaian Soal Latihan 13.10 sampai 13.12 dalam Java.*

```
            int u = q.pollFirst();
            for (int v : g.get(u)) {
                ways[v] += ways[u];                  // paths to u extend to v
                if (--indeg[v] == 0) q.addLast(v);
            }
        }
        return ways;
    }

    public static void main(String[] args) {
        List<List<Integer>> campus = undirected(10, new int[][] {
            {0, 1}, {0, 2}, {0, 3}, {1, 4}, {1, 5}, {2, 3},
            {3, 6}, {4, 5}, {5, 6}, {6, 7}, {8, 9}});
        List<List<Integer>> square = undirected(4, new int[][] {
            {0, 1}, {1, 2}, {2, 3}, {3, 0}});
        int[] color = new int[10];
        System.out.printf("13.10 campus map bipartite? %s%n",
                bipartite(campus, color) ? "yes" : "no");
        color = new int[4];
        boolean sq = bipartite(square, color);
        System.out.printf("13.10 4-cycle bipartite? %s, colors", sq ? "yes" : "no");
        for (int c : color) System.out.print(" " + c);
        System.out.println();

        String[] rows = {"S..#....",
                         ".#.#.##.",
                         ".#...#..",
                         ".####.#.",
                         "......#T"};
        char[][] room = new char[rows.length][];
        for (int i = 0; i < rows.length; i++) room[i] = rows[i].toCharArray();
        int steps = gridBfs(room);
        System.out.printf("%n13.11 fewest steps from S to T: %d%n", steps);
        for (char[] row : room) System.out.println("  " + new String(row));

        List<List<Integer>> work = new ArrayList<>();
        for (int i = 0; i < 9; i++) work.add(new ArrayList<>());
        int[][] arcs = {{0, 1}, {1, 2}, {1, 3}, {2, 4}, {3, 5},
                        {3, 7}, {4, 6}, {5, 6}, {6, 8}, {7, 8}};
        for (int[] a : arcs) work.get(a[0]).add(a[1]);
        long[] ways = countPaths(work, 0);
        System.out.printf("%n13.12 paths from order to shelve: %d%n", ways[8]);
        System.out.printf("      paths from order to cover:  %d%n", ways[6]);
    }
}
```

Kode 13.24 `exercises.py`: penyelesaian Soal Latihan 13.10 sampai 13.12 dalam Python.

```
# exercises.py -- solutions to the programming exercises of Chapter 13:
# two-coloring, a grid maze by BFS, and counting paths in a DAG.
from collections import deque


def undirected(n, es):
    g = [[] for _ in range(n)]
    for u, v in es:
        g[u].append(v)
        g[v].append(u)
    return g


def bipartite(g):
    """Two-coloring by BFS; returns (ok, color)."""
    color = [-1] * len(g)
    for s in range(len(g)):
        if color[s] != -1:
            continue
        color[s] = 0
        q = deque([s])
        while q:
            u = q.popleft()
            for v in g[u]:
                if color[v] == -1:
                    color[v] = 1 - color[u]
                    q.append(v)
                elif color[v] == color[u]:
                    return False, color      # an edge inside one color
    return True, color


def grid_bfs(grid):
    """Fewest steps from S to T; '#' is a bookshelf. Marks the route with '*'."""
    R, C = len(grid), len(grid[0])
    dist = [[-1] * C for _ in range(R)]
    came = [[None] * C for _ in range(R)]
    s = next((r, c) for r in range(R) for c in range(C) if grid[r][c] == "S")
    t = next((r, c) for r in range(R) for c in range(C) if grid[r][c] == "T")
    dist[s[0]][s[1]] = 0
    q = deque([s])
    while q:
        r, c = q.popleft()
        for nr, nc in ((r - 1, c), (r + 1, c), (r, c - 1), (r, c + 1)):
            if not (0 <= nr < R and 0 <= nc < C):
                continue
```

Kode 13.24 (Lanjutan) *exercises.py: penyelesaian Soal Latihan 13.10 sampai 13.12 dalam Python.*

```
            if grid[nr][nc] == "#" or dist[nr][nc] != -1:
                continue
            dist[nr][nc] = dist[r][c] + 1
            came[nr][nc] = (r, c)
            q.append((nr, nc))
    cell = came[t[0]][t[1]]
    while cell is not None and cell != s:
        grid[cell[0]][cell[1]] = "*"          # mark the route
        cell = came[cell[0]][cell[1]]
    return dist[t[0]][t[1]]


def count_paths(g, s):
    """Number of different paths from s to every vertex of a DAG (Kahn order)."""
    n = len(g)
    indeg = [0] * n
    for u in range(n):
        for v in g[u]:
            indeg[v] += 1
    q = deque(u for u in range(n) if indeg[u] == 0)
    ways = [0] * n
    ways[s] = 1
    while q:
        u = q.popleft()
        for v in g[u]:
            ways[v] += ways[u]                # every path to u extends to v
            indeg[v] -= 1
            if indeg[v] == 0:
                q.append(v)
    return ways


def main():
    campus = undirected(10, [(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 3),
                             (3, 6), (4, 5), (5, 6), (6, 7), (8, 9)])
    square = undirected(4, [(0, 1), (1, 2), (2, 3), (3, 0)])
    ok, _ = bipartite(campus)
    print(f"13.10 campus map bipartite? {'yes' if ok else 'no'}")
    ok, color = bipartite(square)
    print(f"13.10 4-cycle bipartite? {'yes' if ok else 'no'}, colors "
          + " ".join(str(c) for c in color))
```

Kode 13.24 (Lanjutan) *exercises.py: penyelesaian Soal Latihan 13.10 sampai 13.12 dalam Python.*

```
    room = [list(row) for row in ["S..#....",
                                  ".#.#.##.",
                                  ".#...#..",
                                  ".####.#.",
                                  "......#T"]]
    steps = grid_bfs(room)
    print(f"\n13.11 fewest steps from S to T: {steps}")
    for row in room:
        print("  " + "".join(row))

    work = [[] for _ in range(9)]
    for u, v in [(0, 1), (1, 2), (1, 3), (2, 4), (3, 5),
                 (3, 7), (4, 6), (5, 6), (6, 8), (7, 8)]:
        work[u].append(v)
    ways = count_paths(work, 0)
    print(f"\n13.12 paths from order to shelve: {ways[8]}")
    print(f"      paths from order to cover:  {ways[6]}")


if __name__ == "__main__":
    main()
```

13.13 Ekstraksi (jarak, simpul): (0, PRK), relaksasi AUD 70; (70, AUD), relaksasi ENG 170, MED 230; (170, ENG), relaksasi LIB 320, ADM 230; (230, ADM), relaksasi LIB 320 → 310; (230, MED), relaksasi CAF 410, DRM 320; (310, LIB), CAF akan menjadi 430, tidak; (320, LIB) basi, dilewati; (320, DRM), CAF akan menjadi 410, tidak lebih baik; (410, CAF). Jarak akhir: PRK 0, AUD 70, ENG 170, ADM 230, MED 230, LIB 310, DRM 320, CAF 410; ANX dan WHS tidak dapat dicapai. Rute ke CAF: PRK, AUD, MED, CAF (70 + 160 + 180 = 410). Ya, ada satu entri basi, yaitu (320, LIB) yang lama. Jejak ini dihasilkan dengan menjalankan **Kode 13.18** dari PRK.

13.14 Mula-mula $d[s] = 0$, $d[a] = d[b] = \infty$. Ekstrak s: relaksasi menjadi $d[a] = 2$, $d[b] = 3$. Ekstrak a (2): final. Ekstrak b (3): final; sisi (b, a) akan memberikan $3 - 2 = 1 < 2$, tetapi a sudah selesai dan tidak diperbarui. Hasilnya $d[a] = 2$, padahal jarak sebenarnya 1. Menambahkan konstanta gagal karena cara itu menghukum rute yang sisinya lebih banyak: pada contoh tadi tambahkan 2 pada setiap bobot, sehingga $w(s, a) = 4$, $w(s, b) = 5$, $w(b, a) = 0$; kini rute terpendek ke a adalah sisi langsung (4 melawan 5), sedangkan pada graf asli rute terpendeknya s, b, a. Graf yang digeser mempunyai rute terpendek yang berbeda, sehingga menyelesaikannya tidak menyelesaikan graf aslinya.

13.15 (a) Matriks ketetanggaan: grafnya padat (E mendekati $V^2/2$) dan kecil ($200^2 = 40000$ sel), dan pertanyaan yang sering diajukan adalah pertanyaan sisi $O(1)$. (b) List ketetanggaan: jarang, dan perencanaan rute adalah penelusuran. (c) List sisi: algoritmanya memproses sisi, bukan ketetanggaan, dan mengurutkan list rekaman persis merupakan persoalan Bab 3. (d) List ketetanggaan, dengan list milik akun berderajat tinggi disimpan terurut atau ter-*hash* agar pertanyaan "apakah u mengikuti v" cepat terjawab; matriks atas miliaran akun tidak mungkin (Subbab 13.11.2).

13.16 (a) Keempat daratan berderajat ganjil (A 3, B 5, C 3, D 3). Jalan semacam itu memerlukan nol atau dua simpul ganjil, dan satu jembatan baru mengubah paritas tepat dua daratan. Jadi satu jembatan baru, yang menghubungkan dua daratan mana saja (misalnya A dan D), menyisakan tepat dua simpul ganjil, dan jalan yang memakai setiap jembatan satu kali pun ada, berawal di satu daratan ganjil dan berakhir di daratan ganjil lainnya. (b) Pada komponen utama, LIB, CAF, ENG, MED, AUD, dan PRK berderajat ganjil: enam simpul ganjil, sehingga jalan semacam itu tidak ada; sedikitnya diperlukan dua jalan setapak baru.

Daftar Pustaka

[1] L. Euler, “Solutio problematis ad geometriam situs pertinentis,” *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, vol. 8, pp. 128–140, 1741 (presented 1736).

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 20–22.

[3] E. F. Moore, “The shortest path through a maze,” in *Proc. Int. Symp. Theory of Switching*, Cambridge, MA, USA: Harvard Univ. Press, 1959, pp. 285–292.

[4] C. Y. Lee, “An algorithm for path connections and its applications,” *IRE Trans. Electronic Computers*, vol. EC-10, no. 3, pp. 346–365, Sep. 1961.

[5] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM J. Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[6] J. Hopcroft and R. Tarjan, “Algorithm 447: Efficient algorithms for graph manipulation,” *Communications of the ACM*, vol. 16, no. 6, pp. 372–378, Jun. 1973.

[7] A. B. Kahn, “Topological sorting of large networks,” *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962.

[8] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[9] cppreference.com, “std::priority_queue.” [Online]. Available: https://en.cppreference.com/w/cpp/container/priority_queue. [Accessed: 24-Sep-2026].

[10] Oracle, “Class PriorityQueue,” *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/PriorityQueue.html. [Accessed: 24-Sep-2026].

[11] Python Software Foundation, “heapq — Heap queue algorithm,” *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/heapq.html. [Accessed: 24-Sep-2026].

[12] M. L. Fredman and R. E. Tarjan, “Fibonacci heaps and their uses in improved network optimization algorithms,” *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[13] OpenStreetMap Taginfo, “Database statistics,” data until 23-Sep-2026. [Online]. Available: https://taginfo.openstreetmap.org/reports/database_statistics. [Accessed: 24-Sep-2026].

[14] C. Demetrescu, A. Goldberg, and D. Johnson, “9th DIMACS Implementation Challenge — Shortest Paths: USA road networks,” 2006. [Online]. Available: https://www.diag.uniroma1.it/challenge9/download.shtml. [Accessed: 24-Sep-2026].

[15] Meta Platforms, Inc., “Meta reports second quarter 2026 results,” press release, Jul. 29, 2026. [Online]. Available: https://investor.atmeta.com. [Accessed: 24-Sep-2026].

[16] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck, “Route planning in transportation networks,” in *Algorithm Engineering*, LNCS vol. 9220. Cham, Switzerland: Springer, 2016, pp. 19–80.

[17] S. Bhagat, M. Burke, C. Diuk, I. O. Filiz, and S. Edunov, “Three and a half degrees of separation,” Facebook Research, Feb. 4, 2016. [Online]. Available: https://research.facebook.com/blog/2016/02/three-and-a-half-degrees-of-separation/

[18] Profitina, “Profitina — AI Visibility (GEO) platform.” [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

[19] S. Brin and L. Page, “The anatomy of a large-scale hypertextual Web search engine,” *Computer Networks and ISDN Systems*, vol. 30, no. 1–7, pp. 107–117, 1998.

Bab 14

Heap dan Priority Queue

Di akhir bab ini Anda akan sanggup menjelaskan apa itu *priority queue* (antrean berprioritas) dan mengapa larik terurut maupun larik tak terurut sama-sama bukan wadah yang baik untuknya; menyimpan pohon biner *complete* di dalam larik biasa dan berpindah dari sebuah simpul ke induk dan anak-anaknya hanya dengan aritmetika, baik dengan penomoran berbasis 0 maupun berbasis 1; menyatakan sifat *max-heap* dan *min-heap* dengan tepat dan membuktikan apa yang dijamin keduanya; menjalankan *sift-up* dan *sift-down* dengan tangan, sambil menggambar pohon dan lariknya berdampingan pada setiap langkah; mengimplementasikan `insert`, `getMax`, `extractMax`, `getMin`, `increaseKey`, `decreaseKey`, `buildMaxHeap`, dan `isMaxHeap`, lengkap dengan ongkos persis masing-masing, termasuk alasan mengapa minimum sebuah *max-heap* menuntut pemindaian seluruh daun; membuktikan bahwa membangun *heap* dari bawah ke atas hanya memakan waktu linear, lalu memastikan bukti itu dengan menghitung; menulis *heap sort* secara utuh, di tempat (*in place*), dan menjelaskan mengapa ia tidak stabil; memakai *priority queue* yang sudah tersedia di C++, Java, dan Python, termasuk fungsi-fungsi *max-heap* yang ditambahkan Python 3.14 ke `heapq`; dan menyelesaikan tiga persoalan klasik dengan *heap*: k butir terbesar dari sebuah aliran data, penggabungan k senarai terurut, dan lintasan terpendek algoritma Dijkstra. Sepanjang jalan kita membangun satu kelas `MaxHeap` yang lengkap, dalam C++, Java, dan Python, yang kemudian dikembangkan lebih jauh oleh Bab 16.

14.1 Dari Mana Kita Melanjutkan

Beberapa bab terdahulu sudah mengetuk pintu bab ini. Subbab 3.10 mengurutkan larik dengan *heap sort*: ia membaca larik sebagai pohon, menatanya sehingga setiap simpul paling sedikit sebesar anak-anaknya, lalu berulang kali memindahkan elemen terbesar ke ujung. Subbab itu hanya memperlihatkan *heap* secukupnya untuk mengurutkan, dan berjanji bahwa bab inilah yang akan membangun *heap* dengan sungguh-sungguh, membuktikan ongkosnya, dan memakainya untuk lebih dari sekadar mengurutkan. Subbab 5.15 memperkenalkan *priority queue*, sebuah unit gawat darurat tempat pasien paling mendesak dilayani lebih dulu, bukan pasien yang datang lebih dulu, dan menyimpulkan bahwa implementasi larik yang paling jelas selalu membuat salah satu operasinya lambat. Subbab 5.16 lalu memakai *priority queue* siap pakai dari ketiga pustaka standar tanpa menjelaskan cara kerjanya. Bab 11 memperkenalkan pohon, kosakata akar, induk, anak, daun, level, dan tinggi yang kita pakai di seluruh bab ini, dan bahkan sudah menyimpan pohon biner di dalam larik tanpa pointer (Subbab 11.4.1). Terakhir, Subbab 13.10 menjalankan algoritma lintasan terpendek Dijkstra di atas *priority queue* pustaka dan membiarkan terbuka pertanyaan bagaimana antrean itu bekerja.

Bab ini menepati semua janji itu. Struktur di pusatnya, *binary heap* (heap biner), adalah salah satu gagasan paling berguna di seluruh buku ini, sekaligus salah satu yang paling hemat. Ia sama sekali tidak memerlukan pointer: pohonnya tinggal di dalam larik biasa, dan berpindah dari sebuah simpul ke induk atau anak-anaknya cukup dengan perkalian atau pembagian dengan dua. Dengan satu larik itu saja ia memberi kita elemen terbesar dalam waktu konstan, dan memungkinkan kita menyisipkan elemen baru atau mengambil elemen terbesar dalam waktu sebanding dengan tinggi pohon, yaitu kira-kira $\lg n$: 20 langkah untuk sejuta elemen.

Contoh berjalan kita tetap perpustakaan Pustaka Ceria. Para pembaca mengajukan *permintaan pemesanan* (*hold request*) atas buku, dan tidak semua permintaan sama mendesaknya: dosen yang memerlukan buku untuk mengajar besok, mahasiswa dengan tenggat skripsi minggu ini, dan pembaca yang sekadar ingin membaca-baca bulan depan semuanya menunggu dalam daftar yang sama. Meja sirkulasi harus selalu melayani permintaan yang paling mendesak lebih dulu, sementara permintaan baru terus berdatangan. Itulah *priority queue*, dan pada Subbab 14.12 kita sudah akan membangunnya.

14.2 Melayani yang Paling Mendesak Lebih Dulu

14.2.1 Priority queue sebagai tipe data abstrak

Sebuah *priority queue* menyimpan kumpulan butir, masing-masing dengan sebuah *kunci* (prioritasnya), dan mendukung operasi-operasi berikut. Kita menyatakannya untuk *max-priority queue*, yang kuncinya makin besar berarti makin mendesak; *min-priority queue* adalah bayangan cerminnya, yang kuncinya makin kecil berarti makin mendesak, misalnya tenggat yang lebih awal atau jarak yang lebih pendek.

> **DEFINISI.** Sebuah *max-priority queue* adalah kumpulan kunci dengan operasi `insert(x)` (menambahkan kunci x), `getMax()` (mengembalikan kunci terbesar tanpa mengambilnya), `extractMax()` (mengambil dan mengembalikan kunci terbesar), `increaseKey(i, k)` (menaikkan kunci sebuah butir yang sudah ada di dalam antrean menjadi k), dan `isEmpty()`. Sebuah *min-priority queue* memiliki `getMin`, `extractMin`, dan `decreaseKey` sebagai gantinya.

Tabel 14.1 mendaftar operasi-operasi kelas `MaxHeap` yang dibangun bab ini, beserta ongkos yang nantinya terbukti dimiliki masing-masing. Ongkos-ongkos inilah inti seluruh bab, jadi ada baiknya kita melihatnya lebih dulu sebelum mendapatkannya.

Tabel 14.1 Operasi-operasi kelas `MaxHeap` dalam bab ini beserta ongkosnya.

Operasi	Kegunaannya	Ongkos pada heap berisi n kunci	Subbab
`insert(x)`	menambahkan kunci	$O(\lg n)$	14.7
`getMax()`	membaca kunci terbesar	$O(1)$	14.7
`extractMax()`	mengambil kunci terbesar	$O(\lg n)$	14.7
`getMin()`	membaca kunci terkecil	$\Theta(n)$	14.8
`increaseKey(i, k)`	menaikkan kunci di indeks i	$O(\lg n)$	14.9
`decreaseKey(i, k)`	menurunkan kunci di indeks i	$O(\lg n)$	14.9
`buildMaxHeap(a)`	membuat heap dari seluruh larik	$\Theta(n)$	14.10
`isMaxHeap(a)`	memeriksa sifat heap	$O(n)$	14.11
`heapSort(a)`	mengurutkan larik di tempat	$O(n \lg n)$	14.13

Meskipun bernama demikian, *priority queue* bukanlah *queue* dalam arti pertama masuk pertama keluar seperti di Bab 5. Urutan kedatangan hanya berperan untuk memutus seri di antara kunci yang sama, dan itu pun hanya bila kita memintanya (Subbab 14.14 menunjukkan caranya).

14.2.2 Mengapa struktur yang paling jelas tidak cukup

Andaikan meja sirkulasi menyimpan permintaan pemesanannya di dalam larik. Ada dua cara alami untuk melakukannya, dan cara ketiga yang akan disarankan Bab 11.

- **Larik tak terurut.** Permintaan baru ditambahkan di ujung: $O(1)$. Tetapi menemukan permintaan paling mendesak berarti memindai seluruh n permintaan, dan mengambilnya berarti memindai lalu menambal lubangnya: $\Theta(n)$ setiap kali.
- **Larik terurut.** Permintaan disimpan terurut menurut tingkat kemendesakannya, yang paling mendesak di ujung. Maka `getMax` dan `extractMax` bernilai $O(1)$: jawabannya adalah elemen terakhir. Tetapi `insert` harus menemukan tempat permintaan baru dan menggeser semua elemen sesudahnya satu slot ke kanan, $\Theta(n)$ pergeseran pada kasus terburuk, seperti yang diukur Subbab 1.6.3 untuk penyisipan ke dalam larik.
- **Pohon pencarian biner.** Pohon pencarian yang seimbang mengerjakan keduanya dalam $O(\lg n)$, tetapi ia memerlukan dua pointer per simpul, menjaga urutan terurut penuh yang tidak pernah diminta oleh *priority queue*, dan pohon pencarian biasa (yang tidak seimbang) dapat merosot menjadi rantai sepanjang n, seperti yang diperlihatkan Bab 11.

Tabel 14.2 menjajarkan pilihan-pilihan itu.

Tabel 14.2 Empat cara mengimplementasikan *max-priority queue* berisi n kunci. Hanya *binary heap* yang cepat pada setiap operasi tanpa memori tambahan.

Implementasi	`insert`	`getMax`	`extractMax`	Memori tambahan
Larik tak terurut	$O(1)$	$\Theta(n)$	$\Theta(n)$	tidak ada
Larik terurut	$\Theta(n)$	$O(1)$	$O(1)$	tidak ada
Pohon pencarian seimbang	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	dua pointer per kunci
Binary heap	$O(\lg n)$	$O(1)$	$O(\lg n)$	tidak ada

Kedua versi larik gagal dengan cara yang berlawanan. Larik tak terurut bekerja terlalu sedikit ketika sebuah butir datang, sehingga harus bekerja banyak ketika maksimum diminta; larik terurut bekerja terlalu banyak saat kedatangan, menjaga urutan yang jauh lebih kuat daripada yang diperlukan siapa pun. *Heap* berada di tengah-tengah: ia menjaga urutan *secukupnya* untuk langsung menemukan maksimum, dan urutan yang sedikit itu murah untuk dipulihkan setelah setiap perubahan. Menjaga urutan secukupnya adalah gagasan utama seluruh bab ini.

> **DENGAN BAHASA SEHARI-HARI.** *Priority queue* tidak pernah memerlukan seluruh kumpulannya terurut. Ia hanya perlu tahu, setiap saat, butir mana yang paling depan. *Heap* menjaga urutan sebanyak itu saja, tidak lebih, dan itulah sebabnya ia cepat.

14.2.3 Asal-usul heap

Binary heap diterbitkan pada Juni 1964 oleh J. W. J. Williams sebagai struktur data di balik sebuah algoritma pengurutan baru yang ia namai *heapsort* [1]. Makalah Williams yang pendek itu sudah memuat tata letak larik dalam bab ini dan prosedur untuk menyisipkan elemen serta mengambil elemen terbesar. Enam bulan kemudian Robert W. Floyd menerbitkan versi yang lebih baik, *Treesort 3*, yang sumbangan utamanya adalah membangun *heap* dari larik sembarang dari bawah ke atas dalam waktu linear [2], metode Subbab 14.10. *The Art of Computer Programming* karya Knuth memberikan analisis yang diikuti sebagian besar buku sesudahnya [3], dan Cormen, Leiserson, Rivest, dan Stein memberikan pembahasan buku teks modernnya [4]. Enam puluh tahun kemudian, pustaka standar C++, Java, dan Python masih mengimplementasikan *priority queue* mereka sebagai *binary heap* yang disimpan di dalam larik, pada dasarnya persis seperti yang dijabarkan Williams (Subbab 14.14).

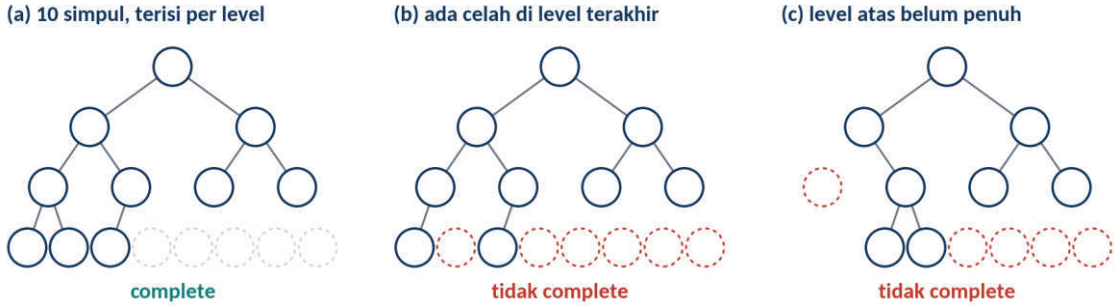
14.3 Pohon Biner Complete di dalam Larik

14.3.1 Pohon biner complete

Bab 11 mendefinisikan pohon biner sebagai pohon yang setiap simpulnya memiliki paling banyak dua anak, anak kiri dan anak kanan. *Heap* memakai pohon biner dengan satu bentuk khusus.

> **DEFINISI.** Pohon biner setinggi h disebut *complete* bila setiap level $0,1,\ldots,h-1$ terisi penuh, dan simpul-simpul level terakhir h dirapatkan sejauh mungkin ke kiri, tanpa celah.

Gambar 14.1 memperlihatkan satu pohon *complete* dan dua pohon yang tidak *complete*. Pohon (a) memiliki sepuluh simpul: level 0, 1, dan 2 penuh (1, 2, dan 4 simpul), dan ketiga simpul level 3 berada di ujung kirinya. Pohon (b) gagal karena level terakhirnya bercelah: posisi kedua kosong sementara posisi sesudahnya terisi. Pohon (c) gagal karena level 2, yang bukan level terakhir, kekurangan satu simpul.



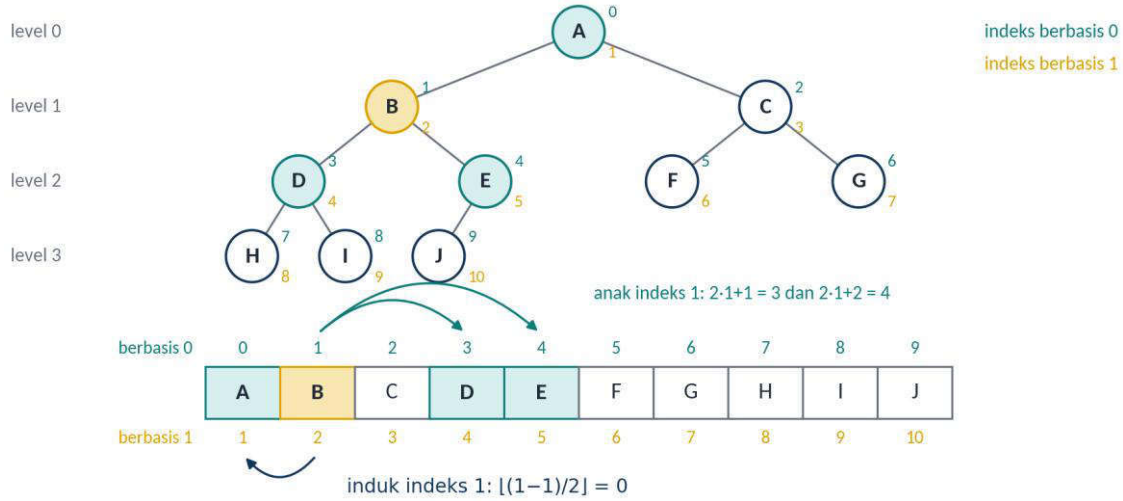
Gambar 14.1 (a) Pohon biner *complete* dengan sepuluh simpul; lingkaran putus-putus menandai tempat kosong di level terakhir, yang terisi dari kiri. (b) dan (c) tidak *complete*: (b) bercelah di level terakhirnya, dan (c) kekurangan simpul di atas level terakhir.

Subbab 11.5.2 sudah bertemu bentuk ini di antara pohon *full*, *perfect*, *complete*, dan *skewed*, serta mengingatkan bahwa nama-nama itu tidak seragam: Cormen dkk. [4], misalnya, menyebutnya *nearly complete* dan menyimpan kata *complete* untuk pohon yang oleh Subbab 11.5.2 disebut *perfect*. Yang penting adalah bentuknya: setiap level penuh kecuali mungkin yang terakhir, dan yang terakhir terisi dari kiri.

Bentuk ini membawa akibat yang luar biasa. Pohon *complete* dengan n simpul sepenuhnya ditentukan oleh bilangan n: hanya ada tepat satu bentuk pohon *complete* bersepuluh simpul, yaitu yang ada pada **Gambar 14.1**(a). Jadi bila simpul-simpulnya kita beri nomor dalam urutan yang tetap, nomor sebuah simpul memberi tahu dengan tepat di mana simpul itu berada, dan kita tidak pernah perlu menyimpan bentuknya sama sekali.

14.3.2 Menomori simpul level demi level

Nomorilah simpul-simpul pohon *complete level demi level, dari kiri ke kanan di dalam setiap level*, mulai dari akar. Inilah urutan kunjungan penelusuran *level order* (Bab 11), dan inilah pula urutan tumbuhnya pohon: simpul berikutnya yang ditambahkan selalu menerima nomor berikutnya. Simpan simpul bernomor i di elemen larik berindeks i. **Gambar 14.2** memperlihatkan pohon sepuluh simpul dari **Gambar 14.1**(a) dengan kunci A sampai J, nomor berbasis 0-nya berwarna teal dan nomor berbasis 1-nya berwarna kuning tua, serta larik yang menyimpannya.



Gambar 14.2 Pohon *complete* yang disimpan level demi level di dalam larik. Setiap simpul membawa indeks berbasis 0 (teal) dan indeks berbasis 1 (kuning tua); busur-busurnya menunjukkan bahwa anak-anak indeks 1 berada di indeks 3 dan 4, dan induknya di indeks 0.

Dengan akar di indeks 0, kerabat simpul berindeks i adalah

$$\text{parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor, \qquad \text{left}(i) = 2i + 1, \qquad \text{right}(i) = 2i + 2. \tag{14.1}$$

Lambang $\lfloor x \rfloor$, *floor* (pembulatan ke bawah) dari x, adalah bilangan bulat terbesar yang tidak melebihi x; di C++, Java, dan Python pembagian bilangan bulat `(i - 1) / 2` menghitungnya untuk setiap $i \geq 1$. Inilah rumus-rumus Subbab 11.4.1, yang dipakai Subbab 3.10 untuk *heap sort*. Periksalah pada **Gambar 14.2**: simpul B berindeks 1; anak-anaknya, D dan E, berindeks $2 \cdot 1 + 1 = 3$ dan $2 \cdot 1 + 2 = 4$; induknya, A, berindeks $\lfloor (1-1)/2 \rfloor = 0$. Simpul E berindeks 4; anak kirinya, J, berindeks $2 \cdot 4 + 1 = 9$, dan anak kanannya akan berindeks 10, yang berada di luar larik bersepuluh elemen, sehingga E hanya punya satu anak.

Banyak buku, termasuk Cormen dkk. [4] dan makalah asli Williams [1], menomori mulai dari 1, dengan slot larik 0 dibiarkan kosong. Dengan akar di indeks 1, kerabat simpul berindeks k adalah

$$\text{parent}(k) = \left\lfloor \frac{k}{2} \right\rfloor, \qquad \text{left}(k) = 2k, \qquad \text{right}(k) = 2k + 1. \tag{14.2}$$

Kedua penomoran menggambarkan pohon yang sama: simpul dengan indeks berbasis 0 i memiliki indeks berbasis 1 $k = i + 1$. Menyubstitusikan $k = i + 1$ ke dalam Persamaan (14.2) mengembalikan Persamaan (14.1): anak kirinya adalah $2k = 2i + 2$ dalam penomoran berbasis 1, yang sama dengan $2i + 1$ dalam penomoran berbasis 0, dan seterusnya. Rumus berbasis 1 sedikit lebih cantik, karena geser kiri satu bit menghasilkan anak kiri dan geser kanan menghasilkan induk, tetapi rumus itu membuang satu slot dan tidak cocok dengan larik berbasis 0 di C++, Java, dan Python. Semua program dalam bab ini memakai rumus berbasis 0 dari Persamaan (14.1); **Tabel 14.3** menghimpun keduanya.

Tabel 14.3 Aritmetika indeks pohon biner *complete* berisi n simpul yang disimpan di dalam larik.

Kerabat simpul	Berbasis 0 (akar di 0)	Berbasis 1 (akar di 1)
induk	$\lfloor (i-1)/2 \rfloor$	$\lfloor k/2 \rfloor$
anak kiri	$2i+1$	$2k$
anak kanan	$2i+2$	$2k+1$
adalah akar	$i=0$	$k=1$
punya anak kiri	$2i+1<n$	$2k \le n$
adalah daun	$i \ge \lfloor n/2 \rfloor$	$k > \lfloor n/2 \rfloor$

14.3.3 Mengapa rumus-rumus itu benar

Rumus yang hanya dicek pada satu contoh barulah tebakan. Subbab 11.4.1 membuktikan rumus berbasis 0 dengan menghitung slot di atas setiap level. Versi berbasis 1 memiliki bukti yang lebih pendek lagi, dan bukti itu memperlihatkan mengapa rumusnya begitu rapi; Persamaan (14.1) lalu kembali diperoleh lewat substitusi di atas.

Level d pada pohon *complete* memuat hingga 2^d simpul, sehingga level-level penuh $0,1,\ldots,d-1$ di atasnya memuat $1+2+\cdots+2^{d-1}=2^d-1$ simpul. Dalam penomoran berbasis 1, simpul pertama level d karenanya mendapat nomor 2^d, dan simpul-simpul level d bernomor $2^d, 2^d+1, \ldots, 2^{d+1}-1$. Ambil simpul k yang merupakan simpul ke-j pada levelnya, dihitung dari $j=0$, sehingga $k=2^d+j$. Masing-masing dari j simpul di sebelah kirinya pada level d memiliki dua anak di level $d+1$, dan $2j$ anak itu muncul paling awal di level $d+1$, dalam urutan kiri-ke-kanan yang sama. Jadi anak kiri k adalah simpul berposisi $2j$ di level $d+1$, yang nomornya

$$2^{d+1}+2j=2(2^d+j)=2k,$$

dan anak kanannya adalah simpul berikutnya, $2k+1$. Sebaliknya, simpul $m \ge 2$ adalah anak kiri dari $m/2$ bila m genap dan anak kanan dari $(m-1)/2$ bila m ganjil; dalam kedua kasus induknya adalah $\lfloor m/2 \rfloor$. Argumen ini hanya memakai fakta bahwa setiap level di atas level terakhir penuh dan level terakhir terisi dari kiri, yang persis merupakan definisi pohon *complete*. Untuk pohon yang bercelah rumus-rumus ini gagal, dan itulah sebabnya *heap* harus selalu tetap *complete*.

14.3.4 Tinggi, dan letak daun-daun

Dua fakta lagi tentang pohon *complete* dipakai di setiap analisis ongkos bab ini. Yang pertama sudah dijumpai di Subbab 11.5.3, yang memperlihatkan bahwa tidak ada pohon biner berisi n simpul yang lebih rendah daripada $\lfloor \lg n \rfloor$ dan bahwa pohon *complete* mencapai batas ini; kita menurunkannya sekali lagi secara langsung untuk pohon *complete*.

Tinggi. Pohon *complete* setinggi h memiliki level 0 sampai $h-1$ yang penuh, 2^h-1 simpul, ditambah antara 1 dan 2^h simpul di level h. Jadi banyak simpulnya memenuhi

$$2^h \le n \le 2^{h+1}-1. \tag{14.3}$$

Mengambil logaritma basis 2, lg, dari ketaksamaan kiri memberi $h \le \lg n$, dan dari ketaksamaan kanan memberi $\lg(n+1) \le h+1$, sehingga $h > \lg n - 1$. Satu-satunya bilangan bulat dalam rentang itu adalah

$$h=\lfloor \lg n \rfloor. \tag{14.4}$$

Heap berisi sepuluh kunci setinggi $\lfloor \lg 10 \rfloor = 3$, *heap* berisi seribu kunci setinggi 9, dan *heap* berisi sejuta kunci setinggi 19. Karena itu setiap operasi dalam bab ini yang menyusuri satu lintasan antara akar dan daun melakukan paling banyak sekitar 20 langkah pada sejuta kunci.

Daun-daun. Dalam penomoran berbasis 0, simpul i mempunyai anak tepat ketika anak kirinya ada, yaitu ketika $2i + 1 \leq n - 1$, atau $i \leq n/2 - 1$. Karena i bilangan bulat, ini berarti $i \leq \lfloor n/2 \rfloor - 1$. Jadi

$$\begin{aligned} &\text{simpul internalnya adalah } 0,1,\ldots,\lfloor n/2 \rfloor \\ &\quad -1, \quad \text{dan daunnya adalah } \lfloor n/2 \rfloor,\ldots,n-1. \end{aligned} \tag{14.5}$$

Pada pohon *complete*, daun-daun selalu menempati paruh kedua larik, dan banyaknya $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$: paling sedikit separuh dari semua simpul adalah daun. Untuk $n = 10$, simpul internalnya 0 sampai 4 dan daunnya 5 sampai 9, seperti yang tampak pada **Gambar 14.2**. Subbab 14.8 memakai fakta ini untuk menemukan minimum *max-heap*, dan Subbab 14.10 memakainya untuk mengetahui dari mana harus mulai membangun *heap*.

Kode 14.1 mencetak seluruh tabel indeks untuk $n = 10$ dalam kedua penomoran, beserta tinggi dan rentang daunnya.

Kode 14.1 `heap_index.cpp`: induk dan anak-anak setiap simpul pohon *complete* bersepuluh simpul, dalam penomoran berbasis 0 dan berbasis 1.

```cpp
// heap_index.cpp -- the index arithmetic of a complete binary tree in an array.
#include <cstdio>

// 0-based: the root is at index 0.
int parent0(int i) { return (i - 1) / 2; }
int left0(int i) { return 2 * i + 1; }
int right0(int i) { return 2 * i + 2; }

// 1-based: the root is at index 1 (slot 0 is left unused).
int parent1(int k) { return k / 2; }
int left1(int k) { return 2 * k; }
int right1(int k) { return 2 * k + 1; }

void cell(int v, int n, int lo) {           // prints v, or "-" if outside
    if (v >= lo && v < n + lo) std::printf("%7d", v);
    else std::printf("%7s", "-");
}

int main() {
    const int n = 10;
    std::printf("n = %d nodes\n", n);
    std::printf("%-12s%7s%7s%7s | %-11s%7s%7s%7s\n", "0-based", "parent",
                "left", "right", "1-based", "parent", "left", "right");
    for (int i = 0; i < n; ++i) {
        int k = i + 1;                      // the same node, numbered from 1
        std::printf("i=%2d        ", i);
        if (i == 0) std::printf("%7s", "-"); else cell(parent0(i), n, 0);
        cell(left0(i), n, 0);
        cell(right0(i), n, 0);
        std::printf(" | k=%2d      ", k);
        if (k == 1) std::printf("%7s", "-"); else cell(parent1(k), n, 1);
        cell(left1(k), n, 1);
        cell(right1(k), n, 1);
        std::printf("\n");
    }
    int h = 0;
    while ((2 << h) <= n) ++h;              // largest h with 2^h <= n
    std::printf("height = floor(lg %d) = %d\n", n, h);
    std::printf("internal nodes: 0..%d, leaves: %d..%d (0-based)\n",
                n / 2 - 1, n / 2, n - 1);
    return 0;
}
```

```
n = 10 nodes
0-based      parent   left  right | 1-based     parent   left  right
i= 0              -      1      2 | k= 1             -      2      3
i= 1              0      3      4 | k= 2             1      4      5
i= 2              0      5      6 | k= 3             1      6      7
i= 3              1      7      8 | k= 4             2      8      9
i= 4              1      9      - | k= 5             2     10      -
i= 5              2      -      - | k= 6             3      -      -
i= 6              2      -      - | k= 7             3      -      -
i= 7              3      -      - | k= 8             4      -      -
i= 8              3      -      - | k= 9             4      -      -
i= 9              4      -      - | k=10             5      -      -
height = floor(lg 10) = 3
internal nodes: 0..4, leaves: 5..9 (0-based)
```

Kedua paruh setiap baris menggambarkan simpul yang sama, dan setiap entri di paruh kanan lebih besar satu daripada entri di sebelah kirinya, seperti yang diramalkan substitusi $k = i + 1$. Tinggi dihitung tanpa bilangan titik mengambang: gelungnya mencari h terbesar dengan $2^h \le n$, yang menurut Persamaan (14.3) sama dengan $\lfloor \lg n \rfloor$. Versi Java dan Python, `HeapIndex.java` dan `heap_index.py`, mencetak tabel yang sama.

14.4 Sifat Heap

14.4.1 Max-heap dan min-heap

Pohon *complete* di dalam larik barulah wadahnya. Yang menjadikannya *heap* adalah sebuah urutan antara setiap simpul dan induknya.

DEFINISI. Larik $A[0..n-1]$, yang dibaca sebagai pohon biner *complete*, adalah sebuah *max-heap* bila setiap simpul selain akar tidak lebih besar daripada induknya:

$$A[\text{parent}(i)] \geq A[i] \qquad \text{untuk setiap } i = 1,2,\ldots,n-1. \tag{14.6}$$

Larik itu adalah *min-heap* bila setiap simpul selain akar tidak lebih kecil daripada induknya:

$$A[\text{parent}(i)] \leq A[i] \qquad \text{untuk setiap } i = 1,2,\ldots,n-1. \tag{14.7}$$

Gambar 14.3 memperlihatkan tujuh kunci yang sama yang ditata sebagai *max-heap* dan sebagai *min-heap*, masing-masing digambar sebagai pohon di samping larik yang menyimpannya. Pada *max-heap*, 90 di akar paling sedikit sebesar anak-anaknya, 65 dan 70; 65 paling sedikit sebesar 55 dan 25; 70 paling sedikit sebesar 40 dan 30. Pada *min-heap* setiap perbandingan menghadap ke arah sebaliknya.



Gambar 14.3 Kunci 25, 30, 40, 55, 65, 70, dan 90 yang ditata (a) sebagai *max-heap*, dengan maksimum di akar, dan (b) sebagai *min-heap*, dengan minimum di akar.

Semua yang kita buktikan untuk *max-heap* berlaku pula untuk *min-heap* dengan setiap perbandingan dibalik. Bab ini bekerja dengan *max-heap*, karena kelas `MaxHeap` kita dan *heap sort* memakainya; *priority queue* pustaka Java dan Python serta aplikasi-aplikasi Subbab 14.15 memakai *min-heap*, dan di sana kita cukup membalik perbandingannya.

14.4.2 Apa yang dijamin oleh sifat itu

Sifat *heap* hanya membandingkan sebuah simpul dengan induknya, tetapi akibatnya bersifat global.

Klaim. Pada *max-heap* yang tidak kosong, akar memuat kunci terbesar: $A[0] \geq A[i]$ untuk setiap i.

Bukti. Ambil sembarang simpul i dan telusuri induk-induknya sampai ke akar: i, parent(i), parent(parent(i)), dan seterusnya. Setiap langkah menuju indeks yang lebih kecil, sehingga penelusuran itu mencapai indeks 0 setelah langkah yang berhingga banyaknya. Menurut Persamaan (14.6), kunci-kunci di sepanjang penelusuran tidak pernah turun: $A[i] \leq A[\text{parent}(i)] \leq \cdots \leq A[0]$. Jadi $A[0] \geq A[i]$. ▫

Argumen yang sama, bila dimulai dari simpul mana pun, memperlihatkan lebih banyak lagi: setiap simpul paling sedikit sebesar setiap simpul di dalam subpohonnya. Dengan kata lain, *setiap lintasan dari akar ke daun pada max-heap terurut tidak naik*. Pada **Gambar 14.3**(a) lintasan 90, 65, 55 menurun, demikian pula 90, 70, 30.

14.4.3 Apa yang tidak dikatakan oleh sifat itu

Sama pentingnya untuk melihat apa yang *tidak* dijanjikan oleh *heap*.

- **Saudara kandung tidak terurut.** Tidak ada aturan yang mengaitkan anak kiri dengan saudara kanannya. Pada **Gambar 14.3**(a) anak kiri akar, 65, lebih kecil daripada anak kanannya, 70; menukar kedua subpohon itu akan menghasilkan *max-heap* lain yang juga sah.
- **Level tidak terurut.** Simpul yang letaknya dalam dapat lebih besar daripada simpul yang lebih tinggi di cabang lain. Pada **Gambar 14.3**(a), 55 di level 2 lebih besar daripada 40 di level yang sama dan lebih kecil daripada 65 di level atasnya, sedangkan 40 lebih kecil daripada kedua tetangganya di level 1, yaitu 65 dan 70; sifat *heap* hanya membandingkan sebuah simpul dengan induknya sendiri, tidak pernah dengan sisa levelnya.
- **Lariknya tidak terurut.** Dibaca dari kiri ke kanan, *max-heap* pada **Gambar 14.3**(a) adalah 90, 65, 70, 55, 25, 40, 30: tidak dalam urutan apa pun. Larik yang terurut (menurun) *merupakan max-heap*, tetapi *max-heap* hampir tidak pernah terurut.
- **Heap bukan pohon pencarian.** Pada pohon pencarian biner Bab 11, subpohon kiri memuat kunci yang lebih kecil dan subpohon kanan kunci yang lebih besar, sehingga sebuah kunci dapat ditemukan dengan satu penelusuran dari akar. Pada *heap* sebuah kunci bisa berada di subpohon mana pun, sehingga mencari kunci sembarang memerlukan pemindaian hingga seluruh n elemen.

Urutan yang lebih lemah ini adalah harga kecepatannya. Pohon pencarian harus menjaga urutan terurut penuh, dan larik terurut lebih-lebih lagi; *heap* menjaga urutan parsial yang murah untuk diperbaiki, dan itu sudah cukup untuk menjawab satu-satunya pertanyaan yang diajukan *priority queue*.

> **PERINGATAN.** Kata *heap* mempunyai arti kedua yang tidak berhubungan. Subbab 1.3.5 memakainya untuk wilayah memori tempat `new` mengalokasikan. Struktur data dalam bab ini tidak ada sangkut pautnya dengan wilayah itu; kedua arti itu hanya kebetulan berbagi nama.

14.5 Sift-Up: Memperbaiki Kunci yang Terlalu Besar

Setiap operasi yang mengubah *heap* merusak sifat *heap* di paling banyak satu tempat, lalu memperbaikinya. Hanya ada dua cara sebuah kunci bisa salah tempat: ia bisa *terlalu besar* untuk posisinya, lebih besar daripada induknya, atau *terlalu kecil*, lebih kecil daripada salah satu anaknya. Yang pertama diperbaiki dengan menaikkan kunci itu, yang kedua dengan menurunkannya. Kedua prosedur perbaikan ini, *sift-up* dan *sift-down*, adalah mesin bagi semua yang menyusul; `insert`, `extractMax`, `increaseKey`, `decreaseKey`, `buildMaxHeap`, dan *heap sort* masing-masing hanyalah beberapa baris di sekitar salah satunya. Sebagian buku menyebutnya

heapify-up dan *heapify-down*, *bubble-up* dan *percolate-down*, atau *swim* dan *sink* [5]; gagasannya sama.

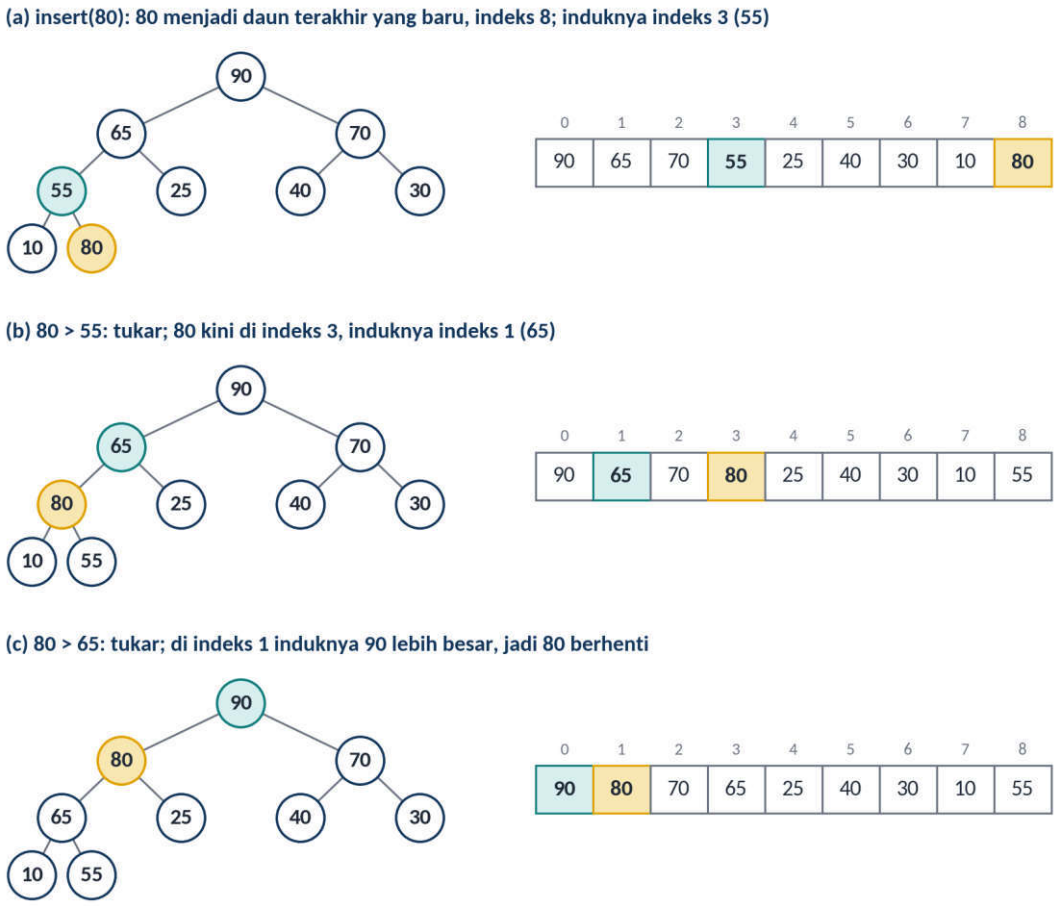
14.5.1 Prosedurnya

Andaikan sifat *heap* berlaku di mana-mana kecuali bahwa kunci di indeks i mungkin lebih besar daripada induknya. *Sift-up* membandingkan kunci itu dengan induknya dan, bila kunci itu lebih besar, menukar keduanya; kunci itu kini naik satu level, dan pertanyaan yang sama diajukan lagi di posisinya yang baru. Prosedur berhenti ketika kunci mencapai akar atau bertemu induk yang paling sedikit sama besarnya. **Pseudokode 14.1** menyajikan prosedur itu.

Pseudokode 14.1 `SIFT-UP`: biarkan kunci di indeks i naik sampai induknya paling sedikit sama besar atau sampai ia mencapai akar.

```
SIFT-UP(A, i)                          // A[i] may be larger than its parent
 1  while i > 0 and A[parent(i)] < A[i]
 2      exchange A[i] with A[parent(i)]
 3      i = parent(i)
```

Gambar 14.4 mengikuti *sift-up* setelah kunci 80 ditempatkan di indeks 8 pada *heap* 90, 65, 70, 55, 25, 40, 30, 10. Pada setiap langkah gambar itu menampilkan pohon dan larik berdampingan: kunci yang bergerak berwarna kuning tua, dan kunci pembandingnya berwarna teal. Induk indeks 8 adalah indeks $\lfloor 7/2 \rfloor = 3$, yang memuat 55; karena $80 > 55$ keduanya ditukar. Di indeks 3 induknya adalah indeks 1, yang memuat 65; $80 > 65$, jadi keduanya ditukar lagi. Di indeks 1 induknya adalah akar, 90, dan $80 < 90$, sehingga gelung berhenti. Dua pertukaran, tiga perbandingan, dan larik itu kembali menjadi *heap*.



Gambar 14.4 *Sift-up* kunci 80 dari indeks 8. Setiap panel menampilkan pohon dan lariknya berdampingan; kunci yang bergerak berwarna kuning tua dan induk pembandingnya berwarna teal.

14.5.2 Mengapa ia benar

Sift-up benar karena ada fakta yang sedikit lebih kuat yang tetap benar pada setiap langkah, sebuah *invarian gelung* (*loop invariant*) dalam arti Subbab 2.5.3, yang dibuktikan dalam tiga bagian yang sama.

Invarian. Pada saat uji baris 1, sifat *heap* $A[\text{parent}(j)] \geq A[j]$ berlaku untuk setiap $j \neq i$; dan bila i mempunyai induk, induk itu paling sedikit sebesar anak-anak i.

Inisialisasi. Satu-satunya pelanggaran yang mungkin ada di i, sesuai asumsi. Bila *sift-up* dipanggil setelah menambahkan daun baru, i sama sekali tidak punya anak; bila dipanggil setelah kunci di i dinaikkan (Subbab 14.9), anak-anak i tidak lebih besar daripada kunci lama di i, yang tidak lebih besar daripada induknya.

Pemeliharaan. Misalkan $p = \text{parent}(i)$, $x = A[i]$, dan $y = A[p]$, dengan $y < x$ sehingga baris 2 dijalankan. Setelah pertukaran, indeks p memuat x dan indeks i memuat y. Di indeks i: anak-anaknya tidak lebih besar daripada y menurut paruh kedua invarian, sehingga sifat itu berlaku di bawah i. Di indeks p: anaknya yang lain, s (saudara i), memenuhi $s \leq y$ sebelumnya, dan $y < x$, jadi $s \leq x$; dan $y < x$ untuk anak i. Jadi satu-satunya pasangan yang mungkin masih salah adalah p dengan induknya sendiri, g. Dan g paling sedikit sebesar anak-anak p: $g \geq y$ menurut sifat *heap* sebelum pertukaran, $y \geq s$, dan y adalah $A[i]$ yang baru. Menetapkan $i = p$ memulihkan invarian.

Terminasi. Gelung berakhir ketika $i = 0$ atau $A[\text{parent}(i)] \geq A[i]$. Dalam kedua kasus satu-satunya pelanggaran yang mungkin sudah hilang, sehingga seluruh larik adalah *max-heap*.

14.5.3 Berapa ongkosnya

Setiap iterasi menaikkan i satu level, dari indeks i ke $\lfloor (i-1)/2 \rfloor$, sehingga gelung berjalan paling banyak sebanyak kedalaman simpul awalnya, yang paling banyak sama dengan tinggi $\lfloor \lg n \rfloor$ menurut Persamaan (14.4). Setiap iterasi melakukan satu perbandingan dan satu pertukaran, dan satu perbandingan lagi mengakhiri gelung. Jadi *sift-up* melakukan paling banyak $\lfloor \lg n \rfloor$ pertukaran dan $\lfloor \lg n \rfloor + 1$ perbandingan: waktu $O(\lg n)$. Ia juga bisa langsung berhenti, bila kuncinya sudah cukup kecil: ongkosnya lalu hanya satu perbandingan.

Sebuah penghalusan kecil, yang dipakai beberapa implementasi pustaka, menghindari sebagian besar operasi tulis. Alih-alih menukar di setiap level, kunci yang bergerak disimpan dalam sebuah variabel, setiap induk yang lebih kecil disalin satu level *turun* ke dalam lubang, dan kunci itu ditulis sekali ke lubang yang terakhir. Cara ini melakukan kira-kira separuh operasi tulis ke memori dibanding pertukaran. Kelas kita tetap memakai pertukaran biasa, yang lebih mudah digambar dan diperiksa; Soal 14.14 meminta Anda menulis versi yang dihaluskan itu.

14.6 Sift-Down: Memperbaiki Kunci yang Terlalu Kecil

14.6.1 Prosedurnya

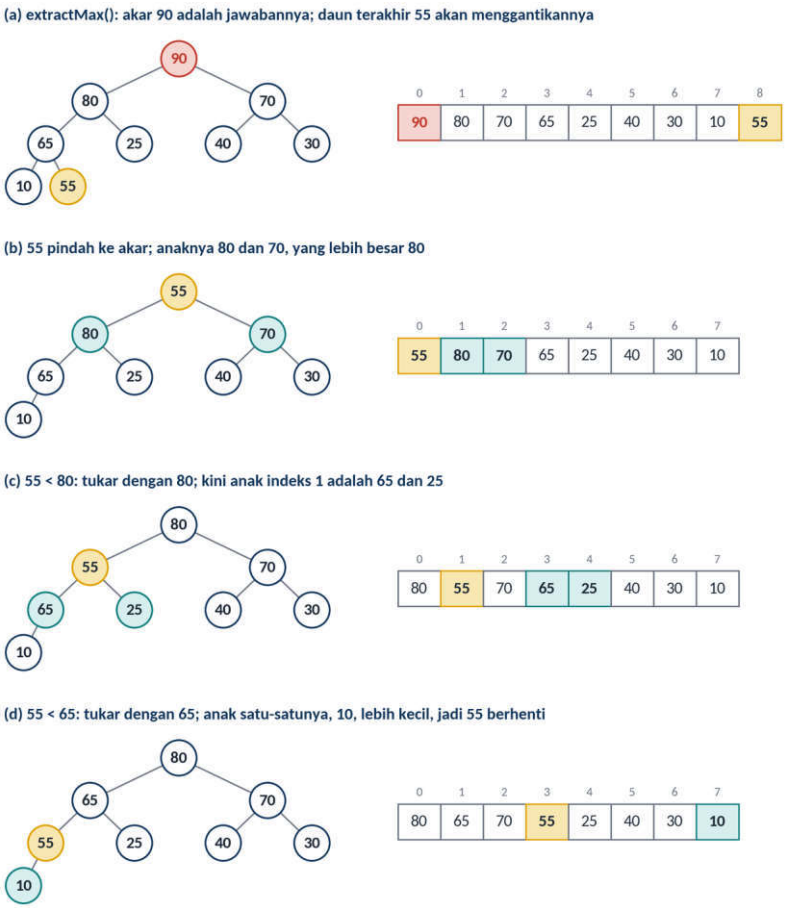
Sekarang andaikan kunci di indeks i mungkin *lebih kecil* daripada salah satu anaknya, sementara kedua subpohon di bawah i adalah *max-heap*. *Sift-down* membandingkan kunci itu dengan anak-anaknya. Bila tidak ada anak yang lebih besar, selesai. Jika tidak, kunci itu ditukar dengan anak yang **lebih besar**, dan pertanyaan yang sama diajukan lagi satu level lebih bawah. **Pseudokode 14.2** menyajikan prosedur untuk *heap* yang menempati $A[0..n-1]$; ukuran n menjadi parameter karena *heap sort* (Subbab 14.13) menjalankan *sift-down* pada *heap* yang hanya merupakan bagian depan larik.

Pseudokode 14.2 `SIFT-DOWN`: biarkan kunci di indeks i turun, menukarnya dengan anak yang lebih besar, sampai tidak ada anak yang lebih besar.

```
SIFT-DOWN(A, i, n)                  // heap is A[0..n-1]; A[i] may be too small
 1  loop
 2      largest = i
 3      l = 2i + 1;  r = 2i + 2
 4      if l < n and A[l] > A[largest]  then largest = l
 5      if r < n and A[r] > A[largest]  then largest = r
 6      if largest == i  then return        // A[i] ≥ both children
 7      exchange A[i] with A[largest]
 8      i = largest
```

Mengapa anak yang *lebih besar*? Setelah pertukaran, anak yang naik menjadi induk bagi anak yang lain. Bila yang kita naikkan anak yang lebih kecil, ia kini berada di atas saudara yang lebih besar: pelanggaran baru. Menaikkan anak yang lebih besar membuatnya paling sedikit sebesar kunci yang turun dan sebesar saudaranya, sehingga sifat itu berlaku di indeks i untuk seterusnya.

Gambar 14.5 memperlihatkan *sift-down* bekerja di dalam `extractMax`, yang dijelaskan Subbab 14.7: akar 90 telah diambil, dan kunci terakhir, 55, telah dipindahkan ke akar *heap* 80, 70, 65, 25, 40, 30, 10. Anak-anak akar adalah 80 dan 70; yang lebih besar 80, dan $55 < 80$, jadi keduanya ditukar. Di indeks 1 anak-anaknya adalah 65 (indeks 3) dan 25 (indeks 4); yang lebih besar 65 dan $55 < 65$, jadi keduanya ditukar. Di indeks 3 satu-satunya anak adalah 10 di indeks 7 (indeks 8 tidak lagi berada di dalam *heap*), dan $55 > 10$, sehingga *sift-down* berhenti.



Gambar 14.5 *Sift-down* kunci 55 dari akar selama `extractMax`: di setiap level ia dibandingkan dengan kedua anaknya (teal) dan ditukar dengan yang lebih besar, sampai tidak ada anak yang lebih besar.

14.6.2 Mengapa ia benar, dan berapa ongkosnya

Argumennya mencerminkan argumen *sift-up*. Pada awal setiap iterasi, subpohon yang berakar di anak-anak i adalah *max-heap*, dan setiap simpul di atas i paling sedikit sebesar semua isi subpohon i. Bila $A[i]$ paling sedikit sebesar kedua anaknya, subpohon i adalah *heap* dan kita berhenti. Jika tidak, misalkan c anak yang lebih besar. Setelah pertukaran, indeks i memuat $A[c]$ yang lama, yang paling sedikit sebesar saudaranya, sebesar semua isi subpohon saudara itu (yang merupakan *heap* di bawah akarnya), dan sebesar $A[i]$ yang lama yang kini berada di c; subpohon saudara itu tidak tersentuh dan tetap *heap*; dan subpohon di c adalah *heap* kecuali mungkin di c sendiri, yang persis merupakan keadaan awal kita, satu level lebih bawah. Gelung pasti berhenti, paling lambat ketika i menjadi daun, karena baris 4 dan 5 tidak lagi menemukan anak.

Setiap iterasi turun satu level, sehingga paling banyak ada $\lfloor \lg n \rfloor$ pertukaran. Setiap iterasi melakukan hingga dua perbandingan, satu per anak, sehingga

$$\text{perbandingan SIFT-DOWN} \le 2\lfloor \lg n \rfloor, \qquad \text{pertukaran} \le \lfloor \lg n \rfloor. \tag{14.8}$$

Itu lagi-lagi $O(\lg n)$, tetapi dengan dua kali lipat perbandingan *sift-up*, karena saat turun ada dua calon di setiap level sedangkan saat naik hanya ada satu induk. Lebih tepatnya, *sift-down* yang dimulai dari simpul setinggi h (panjang lintasan terpanjang dari simpul itu ke bawah sampai daun) melakukan paling banyak h pertukaran dan $2h$ perbandingan. Hitungan yang lebih halus inilah yang memungkinkan pembangunan dalam waktu linear di Subbab 14.10.

14.7 Insert, GetMax, dan ExtractMax

Dengan kedua prosedur perbaikan di tangan, tiga operasi pokok *priority queue* masing-masing hanya memerlukan beberapa baris.

14.7.1 `getMax`

Menurut klaim Subbab 14.4.2, akar memuat kunci terbesar. Jadi `getMax` mengembalikan $A[0]$: satu akses larik, waktu $O(1)$, berapa pun n. Meminta maksimum dari *heap* yang kosong adalah kesalahan, dan kelas kita melaporkannya dengan eksepsi, bukan dengan mengembalikan nilai karangan.

14.7.2 `insert`

Kunci baru harus ditempatkan di suatu tempat, dan pohon *complete* berisi n simpul hanya bisa tumbuh di satu tempat: posisi kosong berikutnya di level terakhir, yaitu indeks larik n. Jadi `insert` menambahkan kunci itu di ujung larik, tempat bentuk pohon otomatis tetap *complete*, lalu memperbaiki urutannya dengan *sift-up*, karena kunci baru itu mungkin lebih besar daripada induknya. **Pseudokode 14.3** menyajikan kedua langkah itu.

Pseudokode 14.3 `INSERT`: tambahkan kunci sebagai daun terakhir, lalu biarkan ia naik.

```
INSERT(A, n, x)                    // A[0..n-1] is a max-heap
 1  A[n] = x                       // the new last leaf keeps the tree complete
 2  n = n + 1
 3  SIFT-UP(A, n - 1)
```

Ongkosnya adalah ongkos *sift-up*, $O(\lg n)$, ditambah penambahan di ujung yang $O(1)$ (teramortisasi $O(1)$ bila larik tumbuh dengan penggandaan, Subbab 1.6.5). **Gambar 14.4** persis merupakan `insert(80)`. Seluruh rangkaian permintaan pemesanan Pustaka Ceria pada Subbab 14.12 membangun *heap* lewat sembilan penyisipan skor kemendesakan 40, 25, 70, 10, 55, 90, 30, 65, dan 80; inilah *heap* setelah setiap penyisipan, seperti yang dicetak program Subbab 14.12:

```
  insert(40)  -> 40
  insert(25)  -> 40 25
  insert(70)  -> 70 25 40
  insert(10)  -> 70 25 40 10
  insert(55)  -> 70 55 40 10 25
  insert(90)  -> 90 55 70 10 25 40
  insert(30)  -> 90 55 70 10 25 40 30
  insert(65)  -> 90 65 70 55 25 40 30 10
  insert(80)  -> 90 80 70 65 25 40 30 10 55
```

Kerjakan beberapa baris dengan tangan. `insert(70)` menaruh 70 di indeks 2, yang induknya indeks 0 berisi 40; $70 > 40$, jadi keduanya ditukar, menghasilkan 70, 25, 40. `insert(55)` menaruh 55 di indeks 4, induknya indeks 1 berisi 25, ditukar; di indeks 1 induknya, 70, lebih besar, sehingga berhenti: 70, 55, 40, 10, 25. `insert(90)` menaruh 90 di indeks 5, induknya indeks 2 berisi 40, ditukar; lalu induknya indeks 0 berisi 70, ditukar; 90 telah mencapai akar. `insert(30)` menaruh 30 di indeks 6 di bawah 70 dan tidak ada yang bergeser. Sebagian besar penyisipan hanya menaikkan kunci baru satu atau dua level, atau tidak sama sekali; hanya kunci yang lebih besar daripada semuanya yang menempuh seluruh tinggi pohon.

14.7.3 `extractMax`

Mengambil maksimum adalah bayangan cerminnya. Maksimumnya adalah akar, tetapi kita tidak bisa begitu saja menghapus akar: itu akan meninggalkan dua subpohon terpisah dan sebuah

lubang di puncak. Satu-satunya simpul yang penghapusannya menjaga pohon tetap *complete* adalah daun *terakhir*, di indeks $n-1$. Jadi `extractMax` (**Pseudokode 14.4**) menyimpan kunci akar untuk dikembalikan, memindahkan kunci terakhir ke akar, mengecilkan *heap* satu elemen, dan memperbaiki akar, tempat kunci yang dipindahkan itu hampir pasti terlalu kecil, dengan *sift-down*.

Pseudokode 14.4 `EXTRACT-MAX`: simpan akar, pindahkan daun terakhir ke akar, kecilkan *heap*, dan biarkan akar yang baru turun.

```
EXTRACT-MAX(A, n)                    // A[0..n-1] is a max-heap, n ≥ 1
 1  if n == 0  then error "the heap is empty"
 2  top = A[0]
 3  A[0] = A[n - 1]                  // the last leaf fills the hole at the root
 4  n = n - 1
 5  if n > 0  then SIFT-DOWN(A, 0, n)
 6  return top
```

Gambar 14.5 adalah `extractMax` pada *heap* berisi sembilan permintaan pemesanan: ia mengembalikan 90 dan menyisakan 80, 65, 70, 55, 25, 40, 30, 10. `extractMax` kedua mengembalikan 80: kunci terakhir, 10, pindah ke akar, ditukar dengan 70 (yang lebih besar di antara 65 dan 70), lalu dengan 40 (yang lebih besar di antara 40 dan 30 di indeks 5 dan 6), dan *heap* menjadi 70, 65, 40, 55, 25, 10, 30. Ongkosnya satu kali *sift-down* dari akar: paling banyak $2\lfloor \lg n \rfloor$ perbandingan, waktu $O(\lg n)$.

Mengapa yang dipindahkan ke akar adalah kunci *terakhir* dan bukan, katakanlah, anak akar yang lebih besar? Karena memindahkan kunci terakhir itulah yang menjaga bentuk pohon tetap *complete*. Menaikkan anak satu per satu memang menambal lubang di akar, tetapi memindahkan lubang itu turun menyusuri suatu lintasan ke daun yang umumnya bukan daun terakhir, sehingga pohon menjadi bercelah. Memisahkan bentuk dan urutan, yaitu lebih dulu bentuknya dengan memindahkan daun terakhir, lalu urutannya dengan *sifting*, adalah metode di balik setiap operasi *heap*.

> **DENGAN BAHASA SEHARI-HARI.** Untuk menambahkan kunci, taruh ia di ujung dan biarkan ia memanjat selama ia mengalahkan induknya. Untuk mengambil yang teratas, taruh kunci terakhir di puncak dan biarkan ia tenggelam, selalu bertukar tempat dengan anaknya yang lebih besar. Kedua perjalanan itu menyusuri satu lintasan pohon, sehingga keduanya memerlukan sekitar $\lg n$ langkah.

14.8 GetMin pada Max-Heap

Max-heap ditata di sekeliling maksimumnya. Bagaimana bila meja sirkulasi juga menginginkan permintaan yang *paling tidak* mendesak, misalnya untuk memutuskan permintaan mana yang dibuang ketika daftarnya penuh? Sifat *heap* tidak mengatakan apa-apa secara langsung tentang minimum, tetapi ia mengatakan di mana minimum tidak mungkin bersembunyi.

Klaim. Pada *max-heap* yang tidak kosong, ada daun yang memuat kunci terkecil. Bila semua kunci berbeda, kunci terkecil *hanya* berada di daun.

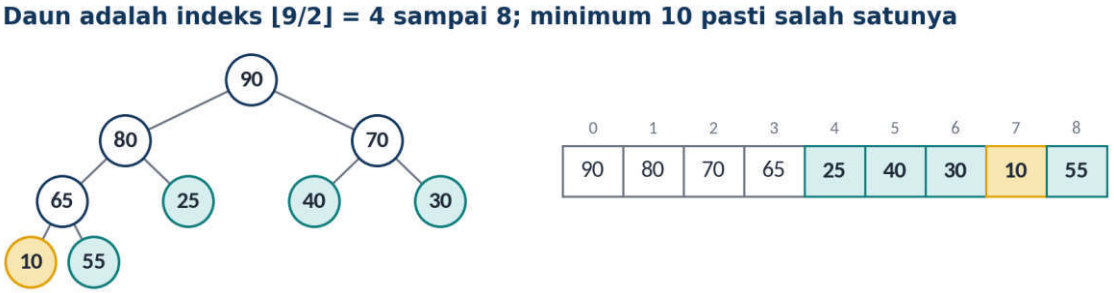
Bukti. Misalkan m kunci terkecil, tersimpan di indeks j. Bila j simpul internal, ia mempunyai anak c, dan menurut sifat *heap* $A[c] \le A[j] = m$; karena m yang terkecil, $A[c] = m$. Ulangi dari c: setiap langkah turun satu level, sehingga setelah paling banyak h langkah kita mencapai daun yang memuat m. Bila semua kunci berbeda, kesamaan $A[c] = m$ mustahil, sehingga j tidak punya anak: ia daun. ▫

Menurut Persamaan (14.5) daun-daun adalah indeks $\lfloor n/2 \rfloor$ sampai $n-1$, paruh kedua larik. Jadi `getMin` cukup memeriksa mereka saja, seperti yang dilakukan **Pseudokode 14.5**.

Pseudokode 14.5 `GET-MIN` untuk *max-heap*: minimum berada di daun, jadi pindailah daun-daun, yaitu paruh kedua larik.

```
GET-MIN(A, n)                          // A[0..n-1] is a max-heap, n ≥ 1
 1  smallest = A[⌊n/2⌋]                // the first leaf
 2  for i = ⌊n/2⌋ + 1 to n - 1
 3      if A[i] < smallest  then smallest = A[i]
 4  return smallest
```

Gambar 14.6 menyorot daun-daun *heap* sembilan kunci: indeks $\lfloor 9/2 \rfloor = 4$ sampai 8, yang memuat 25, 40, 30, 10, dan 55. Pemindaian menemukan 10 setelah empat perbandingan, satu lebih sedikit daripada banyaknya daun.



Gambar 14.6 Daun-daun *max-heap* sembilan kunci adalah indeks 4 sampai 8, paruh kedua larik (teal); minimumnya, 10 (kuning tua), adalah salah satunya.

Memindai separuh larik tetaplah linear: $\lceil n/2 \rceil - 1$ perbandingan, yaitu $\Theta(n)$. Dapatkah cara yang lebih cerdik berbuat lebih baik, misalnya $O(\lg n)$, sebagaimana `getMax` bekerja dalam $O(1)$? Tidak, dan alasannya mencerahkan. **Setiap daun bisa menjadi minimum.** Ambil sembarang *max-heap* dan sembarang daun ℓ, lalu turunkan kunci di ℓ menjadi nilai yang lebih kecil daripada semua kunci lain. Larik itu tetap *max-heap*, karena menurunkan daun hanya bisa membuatnya lebih kecil daripada induknya, yang diperbolehkan sifat itu, dan daun tidak punya anak yang bisa dilanggar. Jadi algoritma yang tidak pernah memeriksa daun ℓ tidak dapat membedakan *heap* ini dari *heap* semula, dan untuk salah satunya ia akan mengembalikan minimum yang salah. Karena itu setiap algoritma yang benar harus memeriksa seluruh $\lceil n/2 \rceil$ daun, dan `getMin` pada *max-heap* memerlukan waktu $\Omega(n)$. Pemindaian kita optimal.

Ini batas struktural, bukan kelemahan kodenya. *Max-heap* menjawab "terbesar" dalam $O(1)$ dan "terkecil" dalam $\Theta(n)$. Bila sebuah program memerlukan keduanya dengan cepat, ia memakai struktur lain: dua *heap* yang dijalankan berdampingan (satu *max-heap* dan satu *min-heap* berisi butir yang sama), satu *min-max heap*, yang berselang-seling antara level maks dan level min serta memberi kedua ekstrem dalam $O(1)$ dan kedua pengambilan dalam $O(\lg n)$ [6], atau pohon pencarian seimbang. Bila yang diinginkan hanya minimum sepanjang riwayat dan butir tidak pernah diambil, satu variabel yang diperbarui pada setiap `insert` sudah cukup.

14.9 IncreaseKey dan DecreaseKey

Prioritas berubah. Permintaan pemesanan seorang pembaca menjadi lebih mendesak ketika tenggatnya makin dekat; sebuah permintaan menjadi kurang mendesak ketika pembacanya berkata bukunya bisa menunggu. *Priority queue* yang mendukung perubahan semacam itu memerlukan dua operasi lagi. Pada *max-heap*, menaikkan kunci hanya bisa membuatnya terlalu

besar untuk posisinya, dan menurunkannya hanya bisa membuatnya terlalu kecil. Jadi setiap perubahan adalah satu penugasan yang diikuti salah satu dari dua perbaikan kita, seperti yang diperlihatkan **Pseudokode 14.6**.

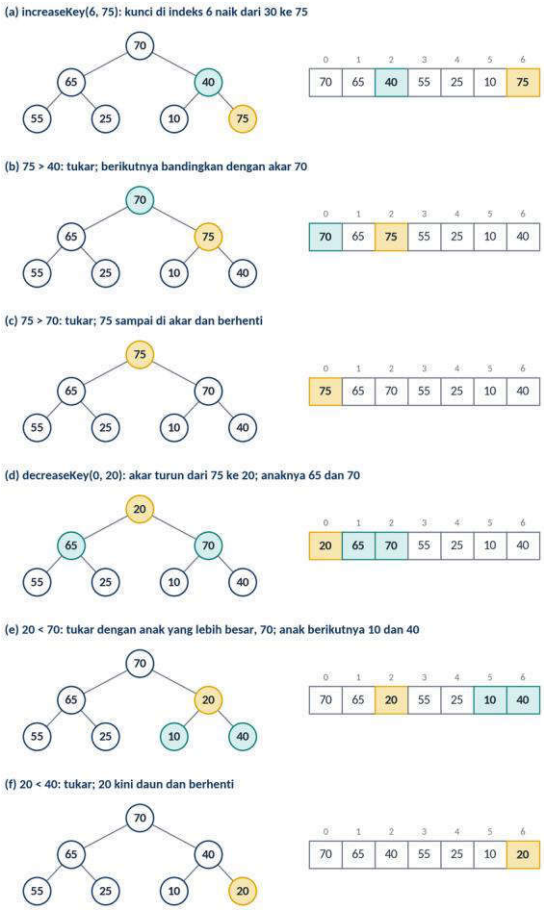
Pseudokode 14.6 `INCREASE-KEY` dan `DECREASE-KEY` pada *max-heap*: ubah kuncinya, lalu biarkan ia naik atau turun.

```
INCREASE-KEY(A, n, i, k)              // raise A[i] to k
 1  if i < 0 or i ≥ n  then error "index is outside the heap"
 2  if k < A[i]  then error "new value is below the key"
 3  A[i] = k
 4  SIFT-UP(A, i)

DECREASE-KEY(A, n, i, k)              // lower A[i] to k
 1  if i < 0 or i ≥ n  then error "index is outside the heap"
 2  if k > A[i]  then error "new value is above the key"
 3  A[i] = k
 4  SIFT-DOWN(A, i, n)
```

Penjaga di baris 1 dan 2 penting. Indeks di luar $0..n-1$ jelas merupakan *bug* pada pemanggil. "Nilai baru" yang lebih kecil daripada kunci sekarang dalam `increaseKey` lebih berbahaya: kodenya tetap berjalan, tetapi *sift-up* tidak akan pernah menurunkan kunci yang lebih kecil itu, dan *heap* diam-diam rusak di bawah indeks i. Menolak panggilan semacam itu lebih aman daripada menebak maksud pemanggil. (Sebuah `changeKey` umum dapat membandingkan nilai baru dengan nilai lama lalu memanggil *sift-up* atau *sift-down* sesuai hasilnya; Soal 14.10 memerlukan tepat logika itu.)

Gambar 14.7 menelusuri kedua operasi pada *heap* 70, 65, 40, 55, 25, 10, 30, yaitu sisa permintaan pemesanan setelah dua panggilan `extractMax`. `increaseKey(6, 75)` menaikkan kunci 30 di indeks 6 menjadi 75. Induknya, indeks 2, memuat 40, jadi keduanya ditukar; di indeks 2 induknya adalah akar, 70, jadi keduanya ditukar lagi, dan 75 menjadi maksimum yang baru: 75, 65, 70, 55, 25, 10, 40. Lalu `decreaseKey(0, 20)` menurunkan akar menjadi 20. Anak-anaknya adalah 65 dan 70; yang lebih besar 70, dan keduanya ditukar; di indeks 2 anak-anaknya adalah 10 dan 40, dan 20 ditukar dengan 40; di indeks 6 ia sudah menjadi daun. Hasilnya 70, 65, 40, 55, 25, 10, 20.



Gambar 14.7 (a) sampai (c) `increaseKey(6, 75)`: kunci yang dinaikkan naik lewat *sift-up*. (d) sampai (f) `decreaseKey(0, 20)`: kunci yang diturunkan tenggelam lewat *sift-down*, selalu ke arah anak yang lebih besar.

Kedua operasi berongkos satu kali *sift*, $O(\lg n)$. Keduanya punya satu kendala praktis: pemanggil harus mengetahui *indeks* butir yang kuncinya berubah. Indeks sebuah butir di dalam *heap* berubah setiap kali *sift-up* atau *sift-down* memindahkannya, sehingga program yang mengubah prioritas butir bernama (seorang pembaca, sebuah simpul graf) harus memelihara tabel kedua dari setiap butir ke indeksnya saat ini, yang diperbarui pada setiap pertukaran. *Indexed priority queue* seperti itu persis yang diperlukan algoritma Dijkstra dengan `decreaseKey`; Subbab 14.15.3 memperlihatkan alternatif yang lebih sederhana yang dimungkinkan oleh pustaka standar.

14.10 Membangun Heap dari Bawah ke Atas dalam Waktu Linear

14.10.1 Dua cara membangun

Sering kali semua kunci sudah diketahui sekaligus: meja sirkulasi dibuka pagi hari dengan seluruh daftar permintaan kemarin, atau *heap sort* menerima larik untuk diurutkan. Salah satu cara membuat *heap* dari kunci-kunci itu adalah memulai dari *heap* kosong dan memanggil `insert` sebanyak n kali. Ongkosnya $O(\lg n)$ per penyisipan dan $O(n \lg n)$ secara keseluruhan, dan kita akan melihat bahwa batas ini memang bisa tercapai.

Metode Floyd [2] lebih baik. Ia mengambil larik *apa adanya*, yang sudah merupakan pohon *complete* menurut Persamaan (14.1), lalu memperbaiki urutannya dari bawah ke atas. Daun-daun, paruh kedua larik menurut Persamaan (14.5), sudah merupakan *heap* bersimpul tunggal: simpul tanpa anak tidak mungkin melanggar apa pun. Jadi **Pseudokode 14.7** mengunjungi

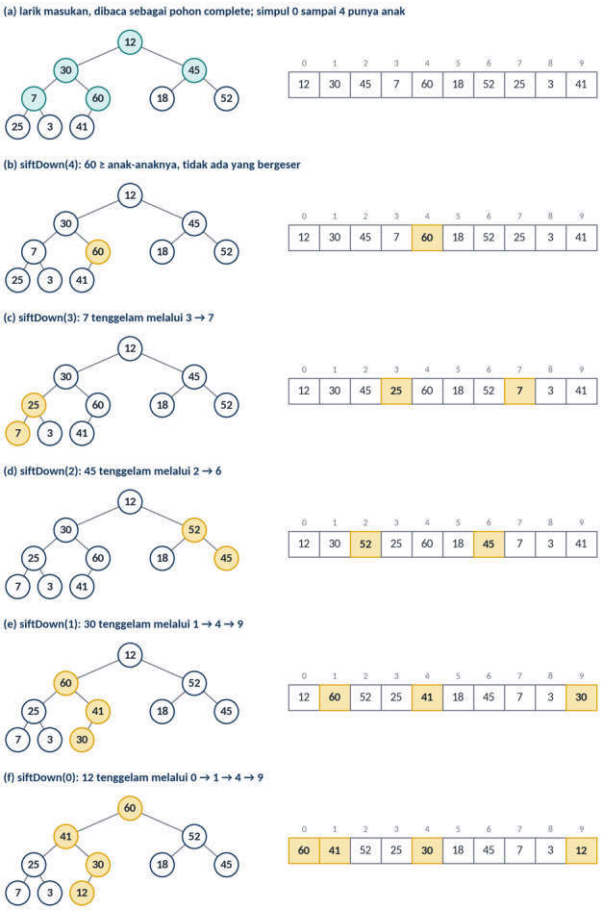
simpul-simpul internal dari yang terakhir, indeks $\lfloor n/2 \rfloor - 1$, mundur sampai akar, dan menjalankan *sift-down* pada masing-masing. Ketika kita sampai di sebuah simpul, kedua subpohonnya sudah dijadikan *heap*, yang persis merupakan prasyarat *sift-down*.

Pseudokode 14.7 `BUILD-MAX-HEAP`: lakukan *sift-down* pada setiap simpul internal, dari yang terakhir mundur sampai akar.

```
BUILD-MAX-HEAP(A, n)
 1  for i = ⌊n/2⌋ - 1 downto 0        // every internal node, bottom-up
 2      SIFT-DOWN(A, i, n)
```

Kebenarannya mengikuti dari sebuah invarian: *pada awal iterasi untuk indeks i, setiap simpul* $i+1, i+2, \ldots, n-1$ *adalah akar sebuah max-heap.* Mula-mula $i = \lfloor n/2 \rfloor - 1$ dan simpul-simpul sesudahnya adalah daun, *heap* bersimpul tunggal. Anak-anak i berindeks lebih besar, sehingga mereka akar *heap*, dan *sift-down* menjadikan i akar sebuah *heap* tanpa mengusik simpul mana pun di luar subpohon i. Ketika gelung berakhir, $i = -1$, sehingga simpul 0, akarnya, adalah akar sebuah *max-heap*: seluruh larik menjadi satu *heap*.

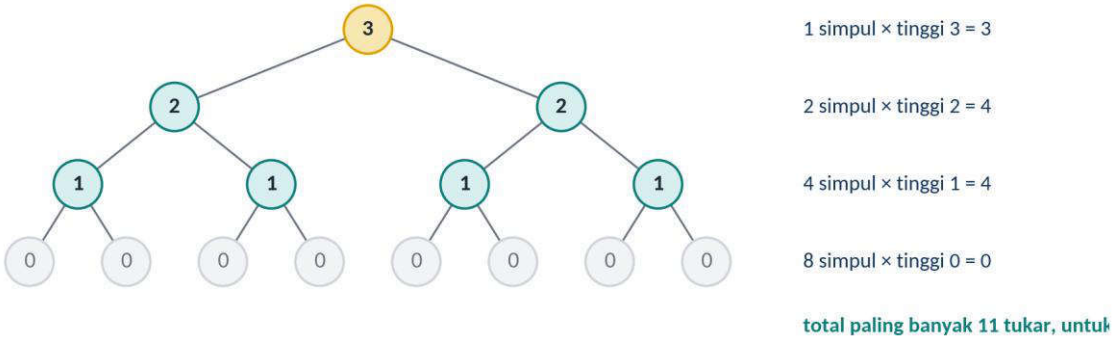
Gambar 14.8 menelusuri `buildMaxHeap` pada larik 12, 30, 45, 7, 60, 18, 52, 25, 3, 41 dari program Subbab 14.12. Dengan $n = 10$ simpul internalnya adalah 0 sampai 4. Simpul 4 memuat 60, lebih besar daripada satu-satunya anaknya, 41: tidak ada yang bergeser. Simpul 3 memuat 7, lebih kecil daripada anak-anaknya 25 dan 3: 7 tenggelam di bawah 25. Simpul 2 memuat 45, dan anaknya yang lebih besar 52: keduanya bertukar tempat. Simpul 1 memuat 30; anaknya yang lebih besar 60 di indeks 4; setelah pertukaran, 30 berada di indeks 4 dengan anak 41 di indeks 9, dan tenggelam sekali lagi. Akhirnya akar 12 tenggelam tiga level, melewati 60, 41, dan 30, dan *heap* 60, 41, 52, 25, 30, 18, 45, 7, 3, 12 pun selesai. Tujuh pertukaran seluruhnya.



Gambar 14.8 `buildMaxHeap` pada 12, 30, 45, 7, 60, 18, 52, 25, 3, 41: *sift-down* diterapkan pada simpul internal 4, 3, 2, 1, dan 0 dengan urutan itu; kuning tua menandai lintasan tempat setiap kunci tenggelam.

14.10.2 Mengapa ongkosnya hanya linear

Taksiran cepat mengatakan: $n/2$ panggilan *sift-down*, masing-masing $O(\lg n)$, jadi $O(n \lg n)$. Batas itu benar tetapi jauh dari ketat. Taksiran itu membebani setiap *sift-down* dengan tinggi seluruh pohon, padahal sebagian besar simpul berada di dekat dasar dan hanya bisa tenggelam sedikit. Separuh dari semua simpul adalah daun dan tidak pernah dikunjungi; seperempatnya berada satu level di atas daun dan paling banyak tenggelam satu level; seperdelapannya paling banyak tenggelam dua level; hanya akar yang bisa tenggelam $\lfloor \lg n \rfloor$ level. **Gambar 14.9** memperlihatkan gambarannya untuk pohon berisi 15 simpul, masing-masing diberi label *tingginya*, yaitu banyak pertukaran terbesar yang dapat dilakukan *sift-down* yang dimulai dari situ.



Gambar 14.9 Setiap simpul *heap* 15 simpul diberi label tingginya, yaitu banyak level terbesar yang dapat ditempuh *sift-down* darinya. Menjumlahkan level demi level memberi paling banyak 11 pertukaran untuk seluruh pembangunan, lebih sedikit daripada 15 simpulnya.

Mari kita buat ini persis. Ambil lebih dulu *heap perfect* setinggi H, yang level terakhirnya juga penuh, sehingga $n = 2^{H+1} - 1$. Level d memuat 2^d simpul, masing-masing setinggi $H - d$. Banyak pertukaran seluruhnya paling banyak

$$S = \sum_{d=0}^{H} 2^d \, (H - d) = \sum_{h=0}^{H} h \; 2^{H-h} = 2^H \sum_{h=0}^{H} \frac{h}{2^h}, \tag{14.9}$$

dengan bentuk tengah menyubstitusikan $h = H - d$. Untuk $H = 3$ ini sama dengan $8 \cdot 0 + 4 \cdot 1 + 2 \cdot 2 + 1 \cdot 3 = 11$, total pada **Gambar 14.9**.

Sekarang kita memerlukan jumlah $h/2^h$. Tuliskan $T = \sum_{h \geq 1} h\,/2^h = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots$ dan bagi dua: $T/2 = \frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \cdots$. Dengan mengurangkan suku demi suku,

$$T - \frac{T}{2} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots = 1, \qquad \text{jadi} \qquad \sum_{h=0}^{\infty} \frac{h}{2^h} = T = 2. \tag{14.10}$$

Jumlah berhingga dalam Persamaan (14.9) lebih kecil daripada jumlah tak berhingganya, jadi $S < 2^H \cdot 2 = 2^{H+1} = n + 1$. (Menghitung jumlah berhingga itu secara persis memberi $S = 2^{H+1} - H - 2 = n - \lg(n+1)$; untuk $n = 15$ hasilnya $15 - 4 = 11$.) Untuk *heap complete* yang tidak *perfect*, hitungan yang sama berlaku level demi level: pada *heap* mana pun berisi n simpul terdapat paling banyak $\lceil n/2^{h+1} \rceil$ simpul setinggi h [4], dan deret yang sama memberi total $O(n)$. Setiap pertukaran disertai paling banyak dua perbandingan, sehingga banyak pertukaran E dan banyak perbandingan C yang dilakukan `BUILD-MAX-HEAP`, serta waktu jalannya T, memenuhi

$$E_{\text{bangun}}(n) < n, \qquad C_{\text{bangun}}(n) < 2n, \qquad T_{\text{bangun}}(n) = \Theta(n), \tag{14.11}$$

Waktunya $\Theta(n)$ dan bukan sekadar $O(n)$ karena gelungnya mengunjungi $\lfloor n/2 \rfloor$ simpul dan melakukan paling sedikit satu perbandingan di masing-masing.

> **DENGAN BAHASA SEHARI-HARI.** Membangun dari bawah ke atas itu murah karena pekerjaannya dilakukan di tempat yang hanya sedikit pekerjaannya. Banyak simpul di dekat dasar masing-masing hanya bisa tenggelam satu atau dua langkah; sedikit simpul di dekat puncak yang bisa tenggelam jauh memang, ya, sedikit. Jumlah "banyak simpul kali jarak" tetap di bawah n.

14.10.3 Membangun lewat penyisipan memang lebih lambat

Mengapa tidak menyisipkan kunci satu per satu saja? Karena penyisipan menaruh pekerjaan di tempat yang salah: sebagian besar kunci disisipkan di dasar, tempat lintasan ke akar *paling panjang*. Bila kunci datang dalam urutan naik, setiap kunci baru adalah yang terbesar sejauh ini dan memanjat sampai ke akar. Kunci yang disisipkan di indeks i (untuk $i \geq 1$) berada di kedalaman $\lfloor \lg(i+1) \rfloor$, sehingga banyak perbandingan seluruhnya adalah

$$\sum_{m=2}^{n} \lfloor \lg m \rfloor = (n+1)\lfloor \lg n \rfloor - 2^{\lfloor \lg n \rfloor + 1} + 2 = \Theta(n \lg n). \tag{14.12}$$

(Soal 14.13 meminta Anda membuktikan bentuk tertutupnya.) Untuk $n = 1000$ hasilnya $1001 \cdot 9 - 1024 + 2 = 7987$.

Kode 14.2 memeriksa kedua analisis itu dengan menghitung. Program ini membangun *heap* berisi 1.000, 10.000, dan 100.000 kunci, dari larik naik, larik acak semu, dan larik turun, sekali lewat n penyisipan dan sekali dari bawah ke atas, lalu mencetak banyak perbandingan kunci dan banyak itu dibagi n. Kunci acak semu berasal dari pembangkit kongruensial linear sederhana, $x \leftarrow (1103515245\,x + 12345) \bmod 2^{31}$, yang ditulis langsung di dalam program sehingga versi C++, Java, dan Python (`BuildCount.java`, `build_count.py`) menghasilkan kunci yang persis sama dan hitungan yang persis sama.

Kode 14.2 `build_count.cpp`: perbandingan kunci yang dilakukan n penyisipan dan satu pembangunan dari bawah ke atas, pada kunci naik, acak semu, dan turun.

```cpp
// build_count.cpp -- comparisons made by n inserts versus one bottom-up build.
#include <cstdio>
#include <utility>
#include <vector>
using namespace std;

long long cmps = 0;

void siftUp(vector<long long>& a, int i) {
    while (i > 0) {
        int p = (i - 1) / 2;
        ++cmps;
        if (!(a[p] < a[i])) return;
        swap(a[p], a[i]);
        i = p;
    }
}

void siftDown(vector<long long>& a, int i, int n) {
    while (true) {
        int largest = i, l = 2 * i + 1, r = 2 * i + 2;
        if (l < n && (++cmps, a[l] > a[largest])) largest = l;
        if (r < n && (++cmps, a[r] > a[largest])) largest = r;
        if (largest == i) return;
        swap(a[i], a[largest]);
        i = largest;
    }
}

long long byInserts(const vector<long long>& in) {
    vector<long long> a;
    cmps = 0;
    for (long long x : in) {
        a.push_back(x);
        siftUp(a, static_cast<int>(a.size()) - 1);
    }
    return cmps;
}

long long byBuild(vector<long long> a) {
    cmps = 0;
    int n = static_cast<int>(a.size());
    for (int i = n / 2 - 1; i >= 0; --i) siftDown(a, i, n);
    return cmps;
}
```

Kode 14.2 (Lanjutan) *build_count.cpp: perbandingan kunci yang dilakukan n penyisipan dan satu pembangunan dari bawah ke atas, pada kunci naik, acak semu, dan turun.*

```
int main() {
    printf("%-11s%8s%13s%9s%12s%9s\n", "input", "n", "n inserts", "per n",
           "build", "per n");
    for (int n : {1000, 10000, 100000}) {
        vector<long long> up(n), down(n), rnd(n);
        unsigned long long x = 2026;                  // a fixed pseudo-random seed
        for (int i = 0; i < n; ++i) {
            up[i] = i;
            down[i] = n - i;
            x = (x * 1103515245ULL + 12345ULL) % 2147483648ULL;
            rnd[i] = static_cast<long long>(x % 1000000);
        }
        pair<const char*, vector<long long>*> inputs[] = {
            {"ascending", &up}, {"random", &rnd}, {"descending", &down}};
        for (auto& [name, v] : inputs) {
            long long ins = byInserts(*v), bld = byBuild(*v);
            printf("%-11s%8d%13lld%9.2f%12lld%9.2f\n", name, n, ins,
                   static_cast<double>(ins) / n, bld, static_cast<double>(bld) / n);
        }
    }
    return 0;
}
```

```
input            n    n inserts    per n       build    per n
ascending     1000         7987     7.99        1982     1.98
random        1000         2236     2.24        1873     1.87
descending    1000          999     1.00         999     1.00
ascending    10000       113631    11.36       19982     2.00
random       10000        22788     2.28       18857     1.89
descending   10000         9999     1.00        9999     1.00
ascending   100000      1468946    14.69      199978     2.00
random      100000       228358     2.28      188116     1.88
descending  100000        99999     1.00       99999     1.00
```

Bacalah tabel itu kolom demi kolom.

- **Pembangunan dari bawah ke atas.** Perbandingan per kunci tetap di bawah atau sama dengan 2,00 untuk setiap masukan dan setiap ukuran, persis seperti yang dijanjikan Persamaan (14.11). Larik naik adalah kasus terburuk (setiap kunci internal lebih kecil daripada anak-anaknya dan tenggelam sejauh mungkin) dan mencapai 1,98 sampai 2,00; larik turun sudah merupakan *heap* dan berongkos tepat $n - 1$ perbandingan, satu per anak.
- **Penyisipan, kunci naik.** Perbandingan per kunci bertambah seiring n: 7,99, 11,36, 14,69. Totalnya, 7.987, 113.631, dan 1.468.946, persis bernilai seperti Persamaan (14.12). Pada $n =$ 100,000 membangun lewat penyisipan melakukan lebih dari tujuh kali lipat perbandingan dibanding membangun dari bawah ke atas.
- **Penyisipan, kunci acak.** Sekitar 2,28 perbandingan per kunci pada setiap ukuran. Kunci baru yang acak biasanya memang berada di dekat dasar, tempat sebagian besar simpul berada, sehingga rata-rata ia hanya memanjat sejumlah konstan level, sebuah fakta yang dibuktikan Porter dan Simon [7]. Pada data acak, n penyisipan rata-rata berjalan linear; $\Theta(n \lg n)$ dari Persamaan (14.12) adalah kasus terburuknya.

Pelajaran untuk praktik: bila semua kunci tersedia sekaligus, pakailah `buildMaxHeap`; bila kunci datang satu per satu, `insert` adalah satu-satunya pilihan, dan biasanya toh murah.

14.11 Memeriksa Sifat Heap: `isMaxHeap`

Heap yang rusak secara diam-diam memberi jawaban salah tanpa pernah *crash*, dan itulah jenis *bug* yang paling buruk. Karena itu fungsi yang *memeriksa* sifat *heap* layak dimiliki, baik sebagai operasi tersendiri (apakah larik yang diterima dari tempat lain ini sebuah *max-heap*?) maupun sebagai alat pengujian yang dapat dipanggil program setelah setiap operasi selama masa pengembangan.

Persamaan (14.6) sudah merupakan algoritma: bandingkan setiap simpul kecuali akar dengan induknya.

Pseudokode 14.8 `IS-MAX-HEAP`: periksa setiap anak terhadap induknya; larik kosong atau berisi satu elemen adalah *heap*.

```
IS-MAX-HEAP(A, n)
 1  for i = 1 to n − 1
 2      if A[parent(i)] < A[i]  then return false
 3  return true
```

Gelungnya melakukan paling banyak $n-1$ perbandingan, waktu $O(n)$, dan berhenti pada pelanggaran pertama. Dijalankan pada larik **Gambar 14.8**(a), 12, 30, 45, 7, 60, 18, 52, 25, 3, 41, ia membandingkan indeks 1 (30) dengan induknya indeks 0 (12), mendapati $12 < 30$, dan langsung mengembalikan `false`. Dijalankan pada *heap* yang dihasilkan `buildMaxHeap`, ia memeriksa kesembilan pasangan induk-anak dan mengembalikan `true`. Dua contoh lagi: pada 10, 7, 8, 5, 2, 3, 1 setiap simpul tidak lebih besar daripada induknya, jadi larik itu *heap*; pada 10, 7, 6, 5, 1, 8, 2 kunci 8 di indeks 5 berinduk 6 di indeks 2, jadi bukan *heap*, meskipun akarnya lebih besar daripada semuanya. Tidak ada algoritma yang bisa lebih baik daripada waktu linear pada kasus terburuk: ketika jawabannya `true`, setiap pasangan dari $n-1$ pasangan induk-anak pasti sudah dibandingkan, karena satu pasangan yang tidak diperiksa bisa saja justru pelanggarannya.

Gelung atas anak-anak, seperti dalam **Pseudokode 14.8**, memiliki keunggulan yang tersembunyi dibanding gelung atas *induk* yang sama wajarnya, `for i = 0 to ⌊n/2⌋ − 1`, yang memeriksa anak kiri dan, bila ada, anak kanan. Gelung atas anak tidak mempunyai kasus khusus: anak kanan yang mungkin tidak ada, larik kosong, dan larik berisi satu elemen semuanya beres dengan sendirinya, karena gelungnya memang tidak berjalan. Gelung atas induk memerlukan penanganan untuk setiap kasus itu, dan di C++ ia menyembunyikan jebakan. Dengan `v.size()` bertipe tak bertanda `size_t`, batas yang ditulis sebagai `(v.size() - 2) / 2` tidak menjadi negatif untuk larik berisi 0 atau 1 elemen: pengurangannya melingkar menjadi sekitar 1.8×10^{19}, gelungnya berjalan, dan ia membaca jauh di luar larik. Subbab 1.5.4 memperlihatkan kerusakan yang dapat ditimbulkan indeks di luar larik; pertahanannya adalah menguji kasus kosong dan kasus satu elemen, dan memilih bentuk gelung yang membuat kasus-kasus itu tidak mungkin salah.

14.12 Kelas `MaxHeap` yang Lengkap

14.12.1 Antarmukanya

Sekarang kita menyatukan setiap operasi ke dalam satu kelas, dalam masing-masing dari ketiga bahasa. **Tabel 14.4** mendaftar antarmuka publiknya. Nama-nama C++ adalah yang dipakai di seluruh bab ini; Java memakai nama yang sama, dan Python mengikuti konvensi penamaannya sendiri dengan garis bawah.

Tabel 14.4 Antarmuka publik kelas `MaxHeap` dalam C++, Java, dan Python.

C++ dan Java	Python	Ongkos	Pada heap kosong atau argumen buruk
`insert(key)`	`insert(key)`	$O(\lg n)$	tidak pernah gagal
`getMax()`	`get_max()`	$O(1)$	kosong: galat
`extractMax()`	`extract_max()`	$O(\lg n)$	kosong: galat
`getMin()`	`get_min()`	$\Theta(n)$	kosong: galat
`increaseKey(i, v)`	`increase_key(i, v)`	$O(\lg n)$	indeks buruk atau v di bawah kunci: galat
`decreaseKey(i, v)`	`decrease_key(i, v)`	$O(\lg n)$	indeks buruk atau v di atas kunci: galat
`buildMaxHeap(a)`	`build_max_heap(a)`	$\Theta(n)$	mengganti isi *heap*
`isMaxHeap(a)` (statis)	`is_max_heap(a)` (statis)	$O(n)$	benar untuk 0 atau 1 kunci
`heapSort(a)` (statis)	`heap_sort(a)` (statis)	$O(n \lg n)$	mengurutkan `a` di tempat
`size()`, `empty()`, `clear()`	`size()`, `empty()`, `clear()`	$O(1)$	—
`printHeap()`, `data()`	`print_heap()`, `data()`	$O(n)$	—

Beberapa keputusan rancangan perlu dijelaskan.

- **Penyimpanannya** adalah larik yang dapat tumbuh: `std::vector<int>` di C++, yang tumbuh sendiri; `int[]` di Java yang digandakan kelas itu ketika penuh, teknik Subbab 1.6.5; dan `list` di Python. Kuncinya berupa nilai `int` biasa, agar algoritmanya tetap tampak jelas; Subbab 14.14 memperlihatkan bagaimana versi pustaka membawa rekaman utuh.
- **Galat berupa eksepsi.** Meminta maksimum dari *heap* kosong tidak mempunyai jawaban yang masuk akal, dan mengembalikan nilai karangan seperti -1 tidak dapat dibedakan dari kunci -1 yang sungguhan. Kelas C++ melempar `std::underflow_error` untuk *heap* kosong, `std::out_of_range` untuk indeks yang buruk, dan `std::invalid_argument` untuk perubahan kunci ke arah yang salah; Java memakai `NoSuchElementException`, `IndexOutOfBoundsException`, dan `IllegalArgumentException`; Python memakai `IndexError` dan `ValueError`. Pesannya sama di ketiga bahasa.
- **`isMaxHeap` dan `heapSort` bersifat statis**, karena keduanya bekerja pada larik apa pun yang diberikan kepadanya, bukan pada isi *heap* itu sendiri. Di C++ anggota statis tetap dapat dipanggil lewat objek, `h.isMaxHeap(v)`, maupun lewat kelasnya, `MaxHeap::isMaxHeap(v)`. Anggota `isValid()` menerapkan pemeriksaan itu pada *heap* itu sendiri, dan dengan cara itulah demonstrasi memastikan bahwa setiap operasi meninggalkan *heap* yang sah.
- **Sift-down menerima larik dan ukurannya sebagai parameter**, sehingga kode yang sama melayani penyimpanan *heap* itu sendiri, `buildMaxHeap`, dan `heapSort` yang bekerja pada larik milik pemanggil.

14.12.2 Kelas C++

Kode 14.3 adalah program lengkapnya: kelas itu dan sebuah `main` yang menjalankan setiap operasi pada permintaan pemesanan Pustaka Ceria.

Kode 14.3 `max_heap.cpp`: kelas `MaxHeap` yang lengkap dan demonstrasi setiap operasinya.

```cpp
// max_heap.cpp -- our MaxHeap class: a max-heap of int keys kept in a vector.
#include <algorithm>
#include <iostream>
#include <stdexcept>
#include <string>
#include <utility>
#include <vector>
using namespace std;

class MaxHeap {
public:
    MaxHeap() = default;

    int size() const { return static_cast<int>(heap.size()); }
    bool empty() const { return heap.empty(); }
    void clear() { heap.clear(); }
    const vector<int>& data() const { return heap; }

    // Prints the array that stores the heap, in index order.
    void printHeap() const {
        for (int i = 0; i < size(); ++i) cout << (i ? " " : "") << heap[i];
        cout << "\n";
    }

    // Appends key as the last leaf, then lets it rise: O(lg n).
    void insert(int key) {
        heap.push_back(key);
        siftUp(size() - 1);
    }

    // Returns the largest key without removing it: O(1).
    int getMax() const {
        requireNonEmpty("getMax");
        return heap[0];
    }

    // Removes and returns the largest key: O(lg n).
    int extractMax() {
        requireNonEmpty("extractMax");
        int top = heap[0];
        heap[0] = heap.back();              // the last leaf fills the hole
        heap.pop_back();
        if (!heap.empty()) siftDown(heap, 0, size());
        return top;
    }
```

Kode 14.3 (Lanjutan) *max_heap.cpp: kelas MaxHeap yang lengkap dan demonstrasi setiap operasinya.*

```
    // Returns the smallest key; it is on a leaf, so scan the leaves: O(n).
    int getMin() const {
        requireNonEmpty("getMin");
        int n = size();
        int smallest = heap[n / 2];         // index n/2 is the first leaf
        for (int i = n / 2 + 1; i < n; ++i) smallest = min(smallest, heap[i]);
        return smallest;
    }

    // Raises the key at index to newValue, then lets it rise: O(lg n).
    void increaseKey(int index, int newValue) {
        checkIndex(index, "increaseKey");
        if (newValue < heap[index])
            throw invalid_argument("increaseKey: new value is below the key");
        heap[index] = newValue;
        siftUp(index);
    }

    // Lowers the key at index to newValue, then lets it sink: O(lg n).
    void decreaseKey(int index, int newValue) {
        checkIndex(index, "decreaseKey");
        if (newValue > heap[index])
            throw invalid_argument("decreaseKey: new value is above the key");
        heap[index] = newValue;
        siftDown(heap, index, size());
    }

    // Replaces the contents by arr and rearranges it bottom-up: O(n).
    void buildMaxHeap(const vector<int>& arr) {
        heap = arr;
        for (int i = size() / 2 - 1; i >= 0; --i) siftDown(heap, i, size());
    }

    // True when every element is no larger than its parent: O(n).
    static bool isMaxHeap(const vector<int>& arr) {
        int n = static_cast<int>(arr.size());
        for (int i = 1; i < n; ++i)
            if (arr[(i - 1) / 2] < arr[i]) return false;
        return true;
    }
    bool isValid() const { return isMaxHeap(heap); }
```

Kode 14.3 (Lanjutan) *max_heap.cpp: kelas MaxHeap yang lengkap dan demonstrasi setiap operasinya.*

```
    // Sorts arr into ascending order in place with heap sort: O(n lg n).
    static void heapSort(vector<int>& arr) {
        int n = static_cast<int>(arr.size());
        for (int i = n / 2 - 1; i >= 0; --i) siftDown(arr, i, n);
        for (int end = n - 1; end > 0; --end) {
            swap(arr[0], arr[end]);         // the maximum goes to its final slot
            siftDown(arr, 0, end);          // repair the heap arr[0..end-1]
        }
    }

private:
    vector<int> heap;                       // heap[0] is the root

    static int parent(int i) { return (i - 1) / 2; }
    static int left(int i) { return 2 * i + 1; }
    static int right(int i) { return 2 * i + 2; }

    // Lets heap[i] rise while it is larger than its parent.
    void siftUp(int i) {
        while (i > 0 && heap[parent(i)] < heap[i]) {
            swap(heap[i], heap[parent(i)]);
            i = parent(i);
        }
    }

    // Lets a[i] sink inside the heap a[0..n-1] until no child is larger.
    static void siftDown(vector<int>& a, int i, int n) {
        while (true) {
            int largest = i, l = left(i), r = right(i);
            if (l < n && a[l] > a[largest]) largest = l;
            if (r < n && a[r] > a[largest]) largest = r;
            if (largest == i) return;
            swap(a[i], a[largest]);
            i = largest;
        }
    }
```

Kode 14.3 (Lanjutan) *max_heap.cpp: kelas MaxHeap yang lengkap dan demonstrasi setiap operasinya.*

```
    void requireNonEmpty(const string& op) const {
        if (heap.empty()) throw underflow_error(op + ": the heap is empty");
    }
    void checkIndex(int index, const string& op) const {
        if (index < 0 || index >= size())
            throw out_of_range(op + ": index " + to_string(index)
                               + " is outside the heap");
    }
};

void show(const string& label, const vector<int>& a) {
    cout << label;
    for (size_t i = 0; i < a.size(); ++i) cout << (i ? " " : "") << a[i];
    cout << "\n";
}

int main() {
    MaxHeap h;
    cout << "Urgent hold requests arrive (urgency scores):\n";
    for (int key : {40, 25, 70, 10, 55, 90, 30, 65, 80}) {
        h.insert(key);
        cout << "  insert(" << key << ")  -> ";
        h.printHeap();
    }
    cout << "size = " << h.size() << ", getMax = " << h.getMax()
         << ", getMin = " << h.getMin() << ", valid = " << boolalpha
         << h.isValid() << "\n\n";

    cout << "extractMax = " << h.extractMax() << "  -> ";
    h.printHeap();
    cout << "extractMax = " << h.extractMax() << "  -> ";
    h.printHeap();

    h.increaseKey(6, 75);
    cout << "increaseKey(6, 75)  -> ";
    h.printHeap();
    h.decreaseKey(0, 20);
    cout << "decreaseKey(0, 20)  -> ";
    h.printHeap();
    cout << "valid = " << h.isValid() << "\n\n";
```

Kode 14.3 (Lanjutan) *max_heap.cpp: kelas MaxHeap yang lengkap dan demonstrasi setiap operasinya.*

```
    vector<int> arr = {12, 30, 45, 7, 60, 18, 52, 25, 3, 41};
    show("array:                ", arr);
    cout << "isMaxHeap(array)     = " << MaxHeap::isMaxHeap(arr) << "\n";
    h.buildMaxHeap(arr);
    cout << "buildMaxHeap(array) -> ";
    h.printHeap();
    cout << "isMaxHeap(heap)      = " << h.isMaxHeap(h.data()) << "\n";
    MaxHeap::heapSort(arr);
    show("heapSort(array)     -> ", arr);
    cout << "\n";

    h.clear();
    cout << "after clear: size = " << h.size() << ", empty = " << h.empty() << "\n";
    try {
        h.extractMax();
    } catch (const exception& e) {
        cout << "error: " << e.what() << "\n";
    }
    h.insert(5);
    try {
        h.increaseKey(0, 3);
    } catch (const exception& e) {
        cout << "error: " << e.what() << "\n";
    }
    try {
        h.increaseKey(4, 99);
    } catch (const exception& e) {
        cout << "error: " << e.what() << "\n";
    }
    return 0;
}
```

Program itu mencetak:

```
Urgent hold requests arrive (urgency scores):
  insert(40)  -> 40
  insert(25)  -> 40 25
  insert(70)  -> 70 25 40
  insert(10)  -> 70 25 40 10
  insert(55)  -> 70 55 40 10 25
  insert(90)  -> 90 55 70 10 25 40
  insert(30)  -> 90 55 70 10 25 40 30
  insert(65)  -> 90 65 70 55 25 40 30 10
  insert(80)  -> 90 80 70 65 25 40 30 10 55
size = 9, getMax = 90, getMin = 10, valid = true

extractMax = 90  -> 80 65 70 55 25 40 30 10
extractMax = 80  -> 70 65 40 55 25 10 30
increaseKey(6, 75)  -> 75 65 70 55 25 10 40
decreaseKey(0, 20)  -> 70 65 40 55 25 10 20
valid = true

array:                12 30 45 7 60 18 52 25 3 41
isMaxHeap(array)     = false
buildMaxHeap(array) -> 60 41 52 25 30 18 45 7 3 12
isMaxHeap(heap)      = true
heapSort(array)     -> 3 7 12 18 25 30 41 45 52 60

after clear: size = 0, empty = true
error: extractMax: the heap is empty
error: increaseKey: new value is below the key
error: increaseKey: index 4 is outside the heap
```

Setiap baris dapat dicocokkan dengan subbab-subbab sebelumnya. Kesembilan penyisipan adalah yang ada di Subbab 14.7.2, berakhir dengan *heap* pada **Gambar 14.4**(c). `getMax` bernilai 90, yaitu

akarnya, dan `getMin` bernilai 10, ditemukan di antara daun seperti pada **Gambar 14.6**. Kedua panggilan `extractMax` adalah yang ada di Subbab 14.7.3; yang pertama adalah **Gambar 14.5**. `increaseKey(6, 75)` dan `decreaseKey(0, 20)` adalah keenam panel **Gambar 14.7**. Larik 12, 30, 45, 7, 60, 18, 52, 25, 3, 41 bukan *heap*, `buildMaxHeap` mengubahnya menjadi *heap* pada **Gambar 14.8**(f), dan `heapSort` mengurutkannya secara naik, seperti yang dijelaskan Subbab 14.13. Tiga baris terakhir memperlihatkan ketiga jenis galat: *heap* kosong, perubahan kunci ke arah yang salah, dan indeks di luar *heap* berisi satu elemen.

Dua hal kecil dalam kodenya. `getMin` memulai pemindaiannya di `heap[n / 2]`, daun pertama menurut Persamaan (14.5), bukan di `heap[0]`: akar tidak pernah menjadi minimum *heap* dengan dua kunci berbeda atau lebih, dan mulai dari daun pertama menghemat satu perbandingan. Dan `extractMax` memanggil *sift-down* hanya bila *heap* tidak kosong setelah pengambilan, karena *heap* yang tadinya berisi satu kunci tidak menyisakan apa pun untuk diperbaiki.

Kami menguji kelas ini di bawah AddressSanitizer dan UndefinedBehaviorSanitizer (Subbab 7.11.3) dan di bawah Valgrind (Subbab 7.11.2): tidak ada akses yang tidak sah, tidak ada kebocoran. `std::vector` mengelola memorinya sendiri, sehingga kelas ini tidak memerlukan destruktor sendiri.

14.12.3 Kelas yang sama dalam Java dan Python

Kode 14.4 dan **Kode 14.5** adalah versi Java dan Python. Keduanya mencetak keluaran yang persis sama dengan di atas.

Kode 14.4 `MaxHeap.java`: kelas `MaxHeap` dalam Java, dengan larik yang digandakan ketika penuh.

```java
// MaxHeap.java -- our MaxHeap class: a max-heap of int keys in a growable array.
import java.util.Arrays;
import java.util.NoSuchElementException;

public class MaxHeap {
    private int[] heap = new int[8];        // heap[0] is the root
    private int n = 0;                      // number of keys in use

    public int size() { return n; }
    public boolean empty() { return n == 0; }
    public void clear() { n = 0; }
    public int[] data() { return Arrays.copyOf(heap, n); }

    // Prints the array that stores the heap, in index order.
    public void printHeap() { System.out.println(join(heap, n)); }

    // Appends key as the last leaf, then lets it rise: O(lg n).
    public void insert(int key) {
        if (n == heap.length) heap = Arrays.copyOf(heap, 2 * n);
        heap[n++] = key;
        siftUp(n - 1);
    }

    // Returns the largest key without removing it: O(1).
    public int getMax() {
        requireNonEmpty("getMax");
        return heap[0];
    }

    // Removes and returns the largest key: O(lg n).
    public int extractMax() {
        requireNonEmpty("extractMax");
        int top = heap[0];
        heap[0] = heap[--n];                // the last leaf fills the hole
        if (n > 0) siftDown(heap, 0, n);
        return top;
    }

    // Returns the smallest key; it is on a leaf, so scan the leaves: O(n).
    public int getMin() {
        requireNonEmpty("getMin");
        int smallest = heap[n / 2];         // index n/2 is the first leaf
        for (int i = n / 2 + 1; i < n; i++) smallest = Math.min(smallest, heap[i]);
        return smallest;
    }
```

Kode 14.4 (Lanjutan) *MaxHeap.java: kelas MaxHeap dalam Java, dengan larik yang digandakan ketika penuh.*

```
    // Raises the key at index to newValue, then lets it rise: O(lg n).
    public void increaseKey(int index, int newValue) {
        checkIndex(index, "increaseKey");
        if (newValue < heap[index])
            throw new IllegalArgumentException(
                    "increaseKey: new value is below the key");
        heap[index] = newValue;
        siftUp(index);
    }

    // Lowers the key at index to newValue, then lets it sink: O(lg n).
    public void decreaseKey(int index, int newValue) {
        checkIndex(index, "decreaseKey");
        if (newValue > heap[index])
            throw new IllegalArgumentException(
                    "decreaseKey: new value is above the key");
        heap[index] = newValue;
        siftDown(heap, index, n);
    }

    // Replaces the contents by arr and rearranges it bottom-up: O(n).
    public void buildMaxHeap(int[] arr) {
        heap = Arrays.copyOf(arr, Math.max(8, arr.length));
        n = arr.length;
        for (int i = n / 2 - 1; i >= 0; i--) siftDown(heap, i, n);
    }

    // True when every element is no larger than its parent: O(n).
    public static boolean isMaxHeap(int[] arr) {
        for (int i = 1; i < arr.length; i++)
            if (arr[(i - 1) / 2] < arr[i]) return false;
        return true;
    }
    public boolean isValid() { return isMaxHeap(data()); }

    // Sorts arr into ascending order in place with heap sort: O(n lg n).
    public static void heapSort(int[] arr) {
        int len = arr.length;
        for (int i = len / 2 - 1; i >= 0; i--) siftDown(arr, i, len);
        for (int end = len - 1; end > 0; end--) {
            swap(arr, 0, end);              // the maximum goes to its final slot
            siftDown(arr, 0, end);          // repair the heap arr[0..end-1]
        }
    }
```

Kode 14.4 (Lanjutan) *MaxHeap.java: kelas MaxHeap dalam Java, dengan larik yang digandakan ketika penuh.*

```
    private static int parent(int i) { return (i - 1) / 2; }
    private static int left(int i) { return 2 * i + 1; }
    private static int right(int i) { return 2 * i + 2; }

    private static void swap(int[] a, int i, int j) {
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }

    // Lets heap[i] rise while it is larger than its parent.
    private void siftUp(int i) {
        while (i > 0 && heap[parent(i)] < heap[i]) {
            swap(heap, i, parent(i));
            i = parent(i);
        }
    }

    // Lets a[i] sink inside the heap a[0..len-1] until no child is larger.
    private static void siftDown(int[] a, int i, int len) {
        while (true) {
            int largest = i, l = left(i), r = right(i);
            if (l < len && a[l] > a[largest]) largest = l;
            if (r < len && a[r] > a[largest]) largest = r;
            if (largest == i) return;
            swap(a, i, largest);
            i = largest;
        }
    }

    private void requireNonEmpty(String op) {
        if (n == 0) throw new NoSuchElementException(op + ": the heap is empty");
    }
    private void checkIndex(int index, String op) {
        if (index < 0 || index >= n)
            throw new IndexOutOfBoundsException(op + ": index " + index
                                                + " is outside the heap");
    }

    static String join(int[] a, int len) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < len; i++) sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.toString();
    }
```

Kode 14.4 (Lanjutan) *MaxHeap.java: kelas MaxHeap dalam Java, dengan larik yang digandakan ketika penuh.*

```
    public static void main(String[] args) {
        MaxHeap h = new MaxHeap();
        System.out.println("Urgent hold requests arrive (urgency scores):");
        for (int key : new int[] {40, 25, 70, 10, 55, 90, 30, 65, 80}) {
            h.insert(key);
            System.out.print("  insert(" + key + ")  -> ");
            h.printHeap();
        }
        System.out.println("size = " + h.size() + ", getMax = " + h.getMax()
                + ", getMin = " + h.getMin() + ", valid = " + h.isValid() + "\n");

        System.out.print("extractMax = " + h.extractMax() + "  -> ");
        h.printHeap();
        System.out.print("extractMax = " + h.extractMax() + "  -> ");
        h.printHeap();

        h.increaseKey(6, 75);
        System.out.print("increaseKey(6, 75)  -> ");
        h.printHeap();
        h.decreaseKey(0, 20);
        System.out.print("decreaseKey(0, 20)  -> ");
        h.printHeap();
        System.out.println("valid = " + h.isValid() + "\n");

        int[] arr = {12, 30, 45, 7, 60, 18, 52, 25, 3, 41};
        System.out.println("array:                " + join(arr, arr.length));
        System.out.println("isMaxHeap(array)      = " + isMaxHeap(arr));
        h.buildMaxHeap(arr);
        System.out.print("buildMaxHeap(array) -> ");
        h.printHeap();
        System.out.println("isMaxHeap(heap)       = " + isMaxHeap(h.data()));
        heapSort(arr);
        System.out.println("heapSort(array)     -> " + join(arr, arr.length));
        System.out.println();

        h.clear();
        System.out.println("after clear: size = " + h.size()
                + ", empty = " + h.empty());
        try {
            h.extractMax();
        } catch (RuntimeException e) {
            System.out.println("error: " + e.getMessage());
        }
        h.insert(5);
        try {
            h.increaseKey(0, 3);
```

Kode 14.4 (Lanjutan) *MaxHeap.java: kelas MaxHeap dalam Java, dengan larik yang digandakan ketika penuh.*

```
        } catch (RuntimeException e) {
            System.out.println("error: " + e.getMessage());
        }
        try {
            h.increaseKey(4, 99);
        } catch (RuntimeException e) {
            System.out.println("error: " + e.getMessage());
        }
    }
}
```

Kode 14.5 `max_heap.py`: kelas `MaxHeap` dalam Python.

```
# max_heap.py -- our MaxHeap class: a max-heap of int keys kept in a list.


class MaxHeap:
    def __init__(self):
        self._heap = []                  # _heap[0] is the root

    def size(self):
        return len(self._heap)

    def empty(self):
        return not self._heap

    def clear(self):
        self._heap.clear()

    def data(self):
        return list(self._heap)

    def print_heap(self):
        """Prints the list that stores the heap, in index order."""
        print(" ".join(map(str, self._heap)))

    def insert(self, key):
        """Appends key as the last leaf, then lets it rise: O(lg n)."""
        self._heap.append(key)
        self._sift_up(len(self._heap) - 1)

    def get_max(self):
        """Returns the largest key without removing it: O(1)."""
        self._require_non_empty("getMax")
        return self._heap[0]

    def extract_max(self):
        """Removes and returns the largest key: O(lg n)."""
        self._require_non_empty("extractMax")
        h = self._heap
        top = h[0]
        last = h.pop()                   # the last leaf fills the hole
        if h:
            h[0] = last
            MaxHeap._sift_down(h, 0, len(h))
        return top
```

Kode 14.5 (Lanjutan) *max_heap.py: kelas MaxHeap dalam Python.*

```
    def get_min(self):
        """Returns the smallest key; it is on a leaf, so scan the leaves: O(n)."""
        self._require_non_empty("getMin")
        return min(self._heap[len(self._heap) // 2:])

    def increase_key(self, index, new_value):
        """Raises the key at index to new_value, then lets it rise: O(lg n)."""
        self._check_index(index, "increaseKey")
        if new_value < self._heap[index]:
            raise ValueError("increaseKey: new value is below the key")
        self._heap[index] = new_value
        self._sift_up(index)

    def decrease_key(self, index, new_value):
        """Lowers the key at index to new_value, then lets it sink: O(lg n)."""
        self._check_index(index, "decreaseKey")
        if new_value > self._heap[index]:
            raise ValueError("decreaseKey: new value is above the key")
        self._heap[index] = new_value
        MaxHeap._sift_down(self._heap, index, len(self._heap))

    def build_max_heap(self, arr):
        """Replaces the contents by arr and rearranges it bottom-up: O(n)."""
        self._heap = list(arr)
        n = len(self._heap)
        for i in range(n // 2 - 1, -1, -1):
            MaxHeap._sift_down(self._heap, i, n)

    @staticmethod
    def is_max_heap(arr):
        """True when every element is no larger than its parent: O(n)."""
        return all(arr[(i - 1) // 2] >= arr[i] for i in range(1, len(arr)))

    def is_valid(self):
        return MaxHeap.is_max_heap(self._heap)

    @staticmethod
    def heap_sort(arr):
        """Sorts arr into ascending order in place with heap sort: O(n lg n)."""
        n = len(arr)
        for i in range(n // 2 - 1, -1, -1):
            MaxHeap._sift_down(arr, i, n)
        for end in range(n - 1, 0, -1):
            arr[0], arr[end] = arr[end], arr[0]   # the maximum goes to its slot
            MaxHeap._sift_down(arr, 0, end)       # repair the heap arr[0..end-1]
```

Kode 14.5 (Lanjutan) *max_heap.py: kelas MaxHeap dalam Python.*

```
    def _sift_up(self, i):
        """Lets _heap[i] rise while it is larger than its parent."""
        h = self._heap
        while i > 0 and h[(i - 1) // 2] < h[i]:
            p = (i - 1) // 2
            h[i], h[p] = h[p], h[i]
            i = p

    @staticmethod
    def _sift_down(a, i, n):
        """Lets a[i] sink inside the heap a[0..n-1] until no child is larger."""
        while True:
            largest, left, right = i, 2 * i + 1, 2 * i + 2
            if left < n and a[left] > a[largest]:
                largest = left
            if right < n and a[right] > a[largest]:
                largest = right
            if largest == i:
                return
            a[i], a[largest] = a[largest], a[i]
            i = largest

    def _require_non_empty(self, op):
        if not self._heap:
            raise IndexError(f"{op}: the heap is empty")

    def _check_index(self, index, op):
        if not 0 <= index < len(self._heap):
            raise IndexError(f"{op}: index {index} is outside the heap")


def join(a):
    return " ".join(map(str, a))


if __name__ == "__main__":
    h = MaxHeap()
    print("Urgent hold requests arrive (urgency scores):")
    for key in (40, 25, 70, 10, 55, 90, 30, 65, 80):
        h.insert(key)
        print(f"  insert({key})  -> ", end="")
        h.print_heap()
    print(f"size = {h.size()}, getMax = {h.get_max()}, getMin = {h.get_min()}, "
          f"valid = {str(h.is_valid()).lower()}\n")
```

Kode 14.5 (Lanjutan) *max_heap.py: kelas MaxHeap dalam Python.*

```
    print(f"extractMax = {h.extract_max()}  -> ", end="")
    h.print_heap()
    print(f"extractMax = {h.extract_max()}  -> ", end="")
    h.print_heap()

    h.increase_key(6, 75)
    print("increaseKey(6, 75)  -> ", end="")
    h.print_heap()
    h.decrease_key(0, 20)
    print("decreaseKey(0, 20)  -> ", end="")
    h.print_heap()
    print(f"valid = {str(h.is_valid()).lower()}\n")

    arr = [12, 30, 45, 7, 60, 18, 52, 25, 3, 41]
    print("array:                " + join(arr))
    print(f"isMaxHeap(array)     = {str(MaxHeap.is_max_heap(arr)).lower()}")
    h.build_max_heap(arr)
    print("buildMaxHeap(array) -> ", end="")
    h.print_heap()
    print(f"isMaxHeap(heap)      = {str(MaxHeap.is_max_heap(h.data())).lower()}")
    MaxHeap.heap_sort(arr)
    print("heapSort(array)     -> " + join(arr) + "\n")

    h.clear()
    print(f"after clear: size = {h.size()}, empty = {str(h.empty()).lower()}")
    try:
        h.extract_max()
    except IndexError as e:
        print(f"error: {e}")
    h.insert(5)
    try:
        h.increase_key(0, 3)
    except ValueError as e:
        print(f"error: {e}")
    try:
        h.increase_key(4, 99)
    except IndexError as e:
        print(f"error: {e}")
```

Kelas Java mengelola lariknya sendiri, sehingga `clear()` cukup menetapkan ukurannya menjadi nol, dan `extractMax` mengecilkan *heap* dengan mengurangi `n`, bukan dengan menghapus elemen; nilai lamanya tetap tinggal di larik, tidak terjangkau, sampai ditimpa. `get_min` di Python memakai irisan (*slicing*) dan fungsi bawaan `min` atas daun-daunnya, `min(self._heap[len(self._heap) // 2:])`, yang merupakan pemindaian yang sama dalam satu baris. `heap_sort` dan `_sift_down` di Python adalah metode statis karena keduanya bekerja pada *list* apa pun.

14.13 Heap Sort secara Utuh

14.13.1 Algoritmanya

Subbab 3.10 memperkenalkan *heap sort* dan menjalankannya pada enam elemen. Kini kita memiliki semua yang diperlukan untuk menyatakannya dengan tepat, membuktikan kebenarannya, dan menganalisisnya. *Heap sort* mengurutkan larik secara naik *di dalam larik itu sendiri*, dalam dua fase, yang dinyatakan secara utuh oleh **Pseudokode 14.9**.

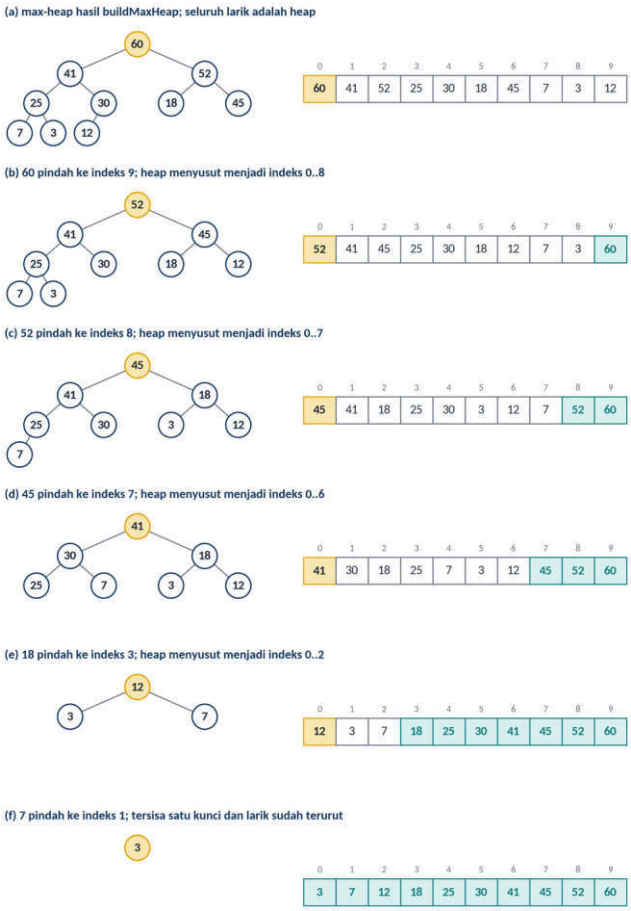
1. **Bangun.** Jadikan seluruh larik *max-heap* dengan `BUILD-MAX-HEAP`.

2. **Ambil.** Ulangi untuk `end` $= n - 1$ turun sampai 1: maksimum *heap* $A[0..end]$ berada di akar; tukarlah dengan $A[end]$, tempat akhirnya; *heap* sekarang adalah $A[0..end-1]$ dengan akar baru yang mungkin terlalu kecil; perbaiki dengan *sift-down*.

Pseudokode 14.9 `HEAP-SORT`: bangun *max-heap*, lalu berulang kali pindahkan akar ke ujung *heap* yang menyusut dan perbaiki akarnya.

```
HEAP-SORT(A, n)
 1  BUILD-MAX-HEAP(A, n)
 2  for end = n - 1 downto 1
 3      exchange A[0] with A[end]      // the largest remaining key to its place
 4      SIFT-DOWN(A, 0, end)           // repair the heap A[0..end-1]
```

Fase kedua adalah `extractMax` dengan satu pelintiran: alih-alih membuang maksimumnya, fase ini menaruhnya di slot yang baru saja dilepaskan oleh *heap* yang menyusut. Karena itu larik selalu terbagi dua: *heap* di depan dan ekor terurut di belakang. **Gambar 14.10** memperlihatkannya pada *heap* yang dibangun **Gambar 14.8**.



Gambar 14.10 *Heap sort* pada *heap* dari **Gambar 14.8**: setelah setiap pertukaran dan *sift-down*, *heap* (pohonnya, akar berwarna kuning tua) menyusut satu dan ekor terurut (teal) bertambah satu. Panel (e) dan (f) melompati langkah-langkah antara yang tampak pada keluaran **Kode 14.6**.

Kode 14.6 mencetak setiap keadaan pengurutan dan menghitung perbandingan kuncinya. `heapSort`-nya berupa templat fungsi yang menerima perbandingan sebagai parameter, `less(x, y)` yang berarti "x datang sebelum y", sehingga kode yang sama mengurutkan bilangan dan, di paruh kedua program, rekaman buku menurut tahun terbitnya.

Kode 14.6 `heap_sort_full.cpp`: *heap sort* dengan jejak setiap keadaannya, dan pengurutan yang sama yang diterapkan pada rekaman buku untuk memperlihatkan bahwa ia tidak stabil.

```cpp
// heap_sort_full.cpp -- heap sort in place, traced; and why it is not stable.
#include <algorithm>
#include <cstdio>
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

long long comparisons = 0;

// Lets a[i] sink inside the heap a[0..n-1]; less(x, y) means "x before y".
template <class T, class Less>
void siftDown(vector<T>& a, int i, int n, Less less) {
    while (true) {
        int largest = i, l = 2 * i + 1, r = 2 * i + 2;
        if (l < n && (++comparisons, less(a[largest], a[l]))) largest = l;
        if (r < n && (++comparisons, less(a[largest], a[r]))) largest = r;
        if (largest == i) return;
        swap(a[i], a[largest]);
        i = largest;
    }
}

template <class T, class Less>
void heapSort(vector<T>& a, Less less,
              void (*show)(const vector<T>&, int) = nullptr) {
    int n = static_cast<int>(a.size());
    for (int i = n / 2 - 1; i >= 0; --i) siftDown(a, i, n, less);   // phase 1
    if (show) show(a, n);
    for (int end = n - 1; end > 0; --end) {                         // phase 2
        swap(a[0], a[end]);
        siftDown(a, 0, end, less);
        if (show) show(a, end);
    }
}

void printInts(const vector<int>& a, int heapSize) {
    cout << "  heap [";
    for (int i = 0; i < heapSize; ++i) cout << (i ? " " : "") << a[i];
    cout << "]  sorted [";
    for (int i = heapSize; i < static_cast<int>(a.size()); ++i)
        cout << (i > heapSize ? " " : "") << a[i];
    cout << "]\n";
}
```

Kode 14.6 (Lanjutan) *heap_sort_full.cpp: heap sort dengan jejak setiap keadaannya, dan pengurutan yang sama yang diterapkan pada rekaman buku untuk memperlihatkan bahwa ia tidak stabil.*

```
struct Book {
    int year;
    string title;
};

string cellOf(const Book& b) { return to_string(b.year) + " " + b.title; }

void printColumns(const vector<Book>& in, const vector<Book>& h,
                  const vector<Book>& s) {
    printf("%-27s%-27s%s\n", "input", "heap sort", "stable sort");
    for (size_t i = 0; i < in.size(); ++i)
        printf("%-27s%-27s%s\n", cellOf(in[i]).c_str(), cellOf(h[i]).c_str(),
               cellOf(s[i]).c_str());
}

int main() {
    vector<int> a = {12, 30, 45, 7, 60, 18, 52, 25, 3, 41};
    cout << "Heap sort of 12 30 45 7 60 18 52 25 3 41\n";
    heapSort(a, [](int x, int y) { return x < y; }, printInts);
    cout << "comparisons = " << comparisons << "\n\n";

    vector<Book> books = {{2012, "Pulang"}, {1980, "Bumi Manusia"},
                          {2012, "Amba"}, {1998, "Saman"},
                          {1980, "Ronggeng Dukuh Paruk"}, {2005, "Laskar Pelangi"}};
    auto byYear = [](const Book& x, const Book& y) { return x.year < y.year; };
    vector<Book> h = books, s = books;
    heapSort(h, byYear);
    stable_sort(s.begin(), s.end(), byYear);
    printColumns(books, h, s);
    return 0;
}
```

```
Heap sort of 12 30 45 7 60 18 52 25 3 41
  heap [60 41 52 25 30 18 45 7 3 12]  sorted []
  heap [52 41 45 25 30 18 12 7 3]  sorted [60]
  heap [45 41 18 25 30 3 12 7]  sorted [52 60]
  heap [41 30 18 25 7 3 12]  sorted [45 52 60]
  heap [30 25 18 12 7 3]  sorted [41 45 52 60]
  heap [25 12 18 3 7]  sorted [30 41 45 52 60]
  heap [18 12 7 3]  sorted [25 30 41 45 52 60]
  heap [12 3 7]  sorted [18 25 30 41 45 52 60]
  heap [7 3]  sorted [12 18 25 30 41 45 52 60]
  heap [3]  sorted [7 12 18 25 30 41 45 52 60]
comparisons = 38

input                       heap sort                   stable sort
2012 Pulang                 1980 Bumi Manusia           1980 Bumi Manusia
1980 Bumi Manusia           1980 Ronggeng Dukuh Paruk   1980 Ronggeng Dukuh Paruk
2012 Amba                   1998 Saman                  1998 Saman
1998 Saman                  2005 Laskar Pelangi         2005 Laskar Pelangi
1980 Ronggeng Dukuh Paruk   2012 Amba                   2012 Pulang
2005 Laskar Pelangi         2012 Pulang                 2012 Amba
```

Baris pertama sesudah judul adalah *heap* setelah fase 1, sama dengan **Gambar 14.8**(f). Setiap baris berikutnya adalah satu iterasi fase 2: 60 pindah ke indeks 9, dan *sift-down* mengubah 12, 41, 52, 25, 30, 18, 45, 7, 3 menjadi *heap* 52, 41, 45, 25, 30, 18, 12, 7, 3; lalu 52 pindah ke indeks 8; dan seterusnya sampai tersisa satu kunci.

14.13.2 Kebenaran

Invarian. Pada awal setiap iterasi gelung baris 2 sampai 4: $A[0..end]$ adalah *max-heap* yang memuat $end + 1$ kunci terkecil larik, dan $A[end + 1..n - 1]$ memuat $n - end - 1$ kunci terbesar dalam urutan naik.

Inisialisasi: $end = n - 1$; seluruh larik adalah *heap* setelah baris 1, dan ekornya kosong. *Pemeliharaan:* akar $A[0]$ adalah kunci terbesar *heap*, jadi yang terbesar di antara $end + 1$ kunci terkecil, sehingga tidak lebih besar daripada kunci mana pun di ekor. Baris 3 menaruhnya di indeks *end*, tepat di depan ekor, yang karenanya tetap terurut dan bertambah satu. Baris 3 juga menaruh $A[end]$ yang lama di akar, satu-satunya pelanggaran yang mungkin di $A[0..end - 1]$, karena kedua subpohon akar masih *heap*; baris 4 memperbaikinya. *Terminasi:* gelung berakhir dengan $end = 0$, sehingga $A[1..n - 1]$ memuat $n - 1$ kunci terbesar secara berurutan dan $A[0]$ memuat yang terkecil. Larik itu terurut.

14.13.3 Ongkos

Fase 1 berongkos $\Theta(n)$ menurut Persamaan (14.11). Fase 2 melakukan $n - 1$ pertukaran dan $n - 1$ *sift-down* dari akar *heap* berukuran $n - 1, n - 2, \ldots, 1$, masing-masing dengan paling banyak $2\lfloor \lg n \rfloor$ perbandingan menurut Persamaan (14.8). Jadi

$$C_{\text{heap sort}}(n) \leq 2n + 2(n - 1)\lfloor \lg n \rfloor = O(n \lg n) \tag{14.13}$$

perbandingan, pada setiap kasus. Pada sepuluh kunci di atas program menghitung 38 perbandingan, jauh di bawah batas $20 + 2 \cdot 9 \cdot 3 = 74$. Tidak ada masukan yang buruk: berbeda dengan *quicksort*, *heap sort* tidak pernah merosot menjadi waktu kuadratik. Karena setiap pengurutan yang bekerja dengan membandingkan kunci memerlukan $\Omega(n \lg n)$ perbandingan pada kasus terburuk (Subbab 3.11), *heap sort* optimal secara asimtotik. Ia memerlukan memori tambahan $O(1)$, beberapa variabel indeks, karena *heap* dan ekor terurut berbagi satu larik; ia bahkan tidak memerlukan tumpukan rekursi seperti *quicksort*, karena *sift-down* kita berupa gelung.

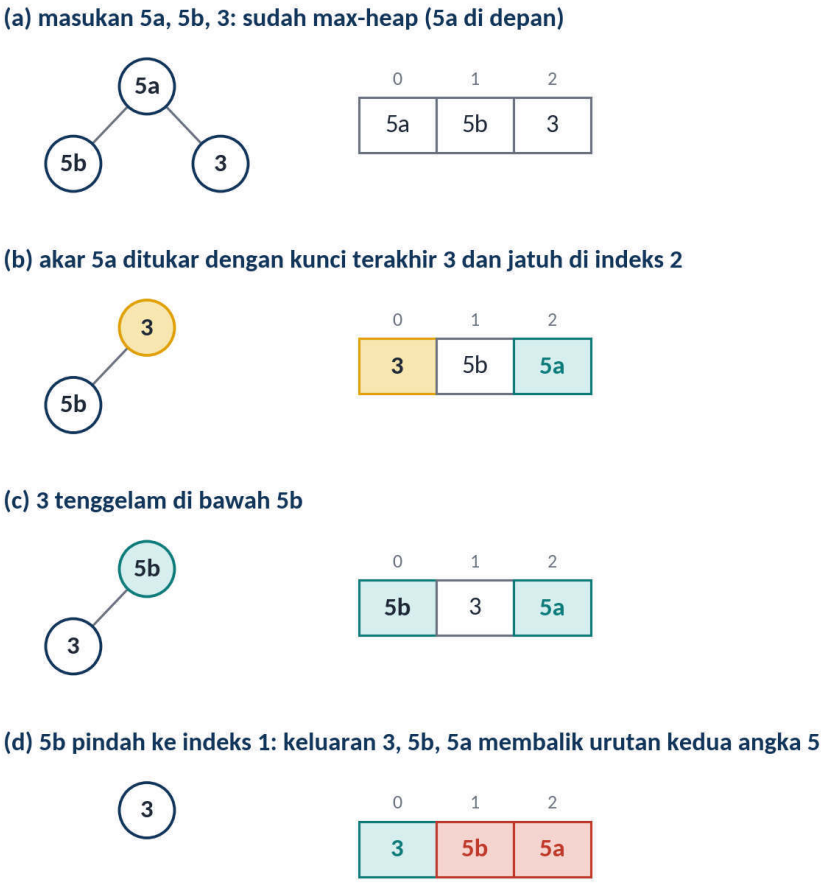
Tabel 14.5 menempatkan *heap sort* di antara pengurutan $O(n \lg n)$ dari Bab 3. Hanya ia yang memadukan jaminan kasus terburuk dengan pengurutan di tempat. Kelemahannya bersifat praktis, bukan asimtotik: *sift-down* melompat dari indeks i ke $2i + 1$, yang letaknya jauh di memori begitu i besar, sehingga *cache* prosesor terpakai dengan buruk, dan dalam praktik *heap sort* biasanya lebih lambat daripada *quicksort* yang baik (Subbab 3.13 mengukur ini); dan ia tidak stabil. Karena alasan yang sama, *introsort* di balik `std::sort` pada pustaka-pustaka C++ utama dimulai sebagai *quicksort* dan beralih ke *heap sort* hanya bila rekursinya menjadi terlalu dalam secara mencurigakan, sehingga memperoleh kecepatan *quicksort* pada masukan biasa dan jaminan kasus terburuk *heap sort* pada masukan yang buruk [8] (Subbab 3.14.1).

Tabel 14.5 *Heap sort* dibandingkan dengan pengurutan perbandingan $O(n \lg n)$ lainnya dari Bab 3.

Pengurutan	Kasus terburuk	Rata-rata	Memori tambahan	Stabil
Merge sort (Subbab 3.8)	$O(n \lg n)$	$O(n \lg n)$	$\Theta(n)$	ya
Quicksort (Subbab 3.9)	$\Theta(n^2)$	$O(n \lg n)$	tumpukan $O(\lg n)$	tidak
Heap sort (subbab ini)	$O(n \lg n)$	$O(n \lg n)$	$O(1)$	tidak

14.13.4 Mengapa heap sort tidak stabil

Sebuah pengurutan disebut *stabil* bila kunci-kunci yang sama besar mempertahankan urutan yang mereka miliki pada masukan (Subbab 3.12). *Heap sort* melanggarnya, dan contoh terkecil memperlihatkan alasannya. **Gambar 14.11** mengurutkan tiga kunci 5, 5, 3, dengan kedua angka 5 diberi label a dan b agar dapat dibedakan. Masukannya sudah *max-heap*. Pertukaran pertama mengirim 5a, sang akar, ke posisi *terakhir*, dan belakangan 5b mendarat di depannya: keluaran 3, 5b, 5a memuat kedua kunci yang sama dalam urutan terbalik.



Gambar 14.11 *Heap sort* atas 5a, 5b, 3 menurut kuncinya: pertukaran pertama memindahkan 5a dari depan ke belakang, melompati 5b, dan keluarannya berakhir dengan 5b sebelum 5a.

Penyebabnya adalah pertukaran jarak jauh: setiap iterasi memindahkan sebuah kunci dari akar langsung ke ujung *heap*, melompati semua yang ada di antaranya, dan kunci-kunci yang sama tidak punya cara untuk mempertahankan urutannya. Paruh kedua **Kode 14.6** memperlihatkan akibatnya pada rekaman sungguhan. Ketika enam buku diurutkan menurut tahun, *heap sort* menaruh *Amba* sebelum *Pulang* meskipun *Pulang* datang lebih dulu pada masukan, sedangkan `std::stable_sort` mempertahankan urutan masukannya. (Kedua buku tahun 1980 kebetulan tetap berurutan pada eksekusi ini; stabilitas adalah jaminan untuk *setiap* masukan, dan *heap sort* tidak memberikannya.) *Heap sort* dapat dibuat stabil dengan mengurutkan pasangan (kunci, posisi asal) alih-alih kuncinya saja, sehingga tidak ada dua elemen yang sama; ongkosnya memori tambahan $\Theta(n)$, yang menghapus keunggulan utama *heap sort*. Soal 14.8 meminta Anda menguraikannya.

14.14 Priority Queue dalam Pustaka Standar

Setelah membangun *heap* dengan tangan, Anda berhak memakai versi pustaka, dan dalam program sungguhan memang sebaiknya begitu, kecuali Anda memerlukan operasi yang tidak dimilikinya. Subbab 5.16 sudah memakainya sekilas. Di sini kita melihatnya lagi dengan bekal pengetahuan tentang cara kerjanya, memakai permintaan pemesanan Pustaka Ceria sebagai rekaman: setiap permintaan memiliki tingkat kemendesakan, nomor kedatangan, nama pembaca, dan judul, dan meja sirkulasi melayani kemendesakan tertinggi lebih dulu dan, di antara kemendesakan yang sama, kedatangan paling awal.

14.14.1 C++: `std::priority_queue`

`std::priority_queue<T, Container, Compare>` adalah sebuah *container adaptor*: secara bawaan ia menyimpan elemen-elemennya di dalam `std::vector<T>` dan memeliharanya sebagai *binary heap*, persis seperti kelas kita [9]. `top()` membaca akar dalam $O(1)$; `push` menambahkan di ujung lalu melakukan *sift-up*; `pop` memindahkan elemen terakhir ke akar lalu melakukan *sift-down*, keduanya dalam $O(\lg n)$. Sejak C++23 ada pula `push_range`, yang menyisipkan seluruh rentang sekaligus [9]. Algoritma yang sama tersedia pada rentang akses acak mana pun sebagai fungsi bebas, `std::make_heap` (pembangunan dari bawah ke atas ala Floyd), `std::push_heap`, `std::pop_heap`, `std::sort_heap`, dan `std::is_heap`, yang berpadanan dengan `buildMaxHeap`, `insert`, `extractMax`, `heapSort`, dan `isMaxHeap` milik kita [10].

Tiga hal kecil menjebak hampir semua orang.

- **Secara bawaan ia adalah max-heap**, karena perbandingan bawaannya `std::less<T>`: elemen yang “terbesar” menurut `<` berada di puncak. Untuk *min-heap*, berikan `std::greater<T>`.
- **Perbandingannya berarti “dilayani lebih belakangan daripada”.** `Compare(a, b)` mengembalikan `true` bila `a` berprioritas *lebih rendah* daripada `b`, yaitu bila `a` seharusnya berada di bawah `b`. Ini terbalik dari cara kebanyakan orang memikirkan prioritas, dan perbandingan yang ditulis terbalik diam-diam membalikkan antreannya.
- **`pop()` tidak mengembalikan apa-apa.** Bacalah `top()` lebih dulu, baru `pop()`. Dan memanggil salah satunya pada antrean kosong adalah perilaku tak terdefinisi, bukan eksepsi: periksa `empty()` lebih dulu.

Kode 14.7 memasukkan permintaan pemesanan ke dalam `priority_queue` dengan perbandingan `ServedLater`, yang menyatakan bahwa permintaan `a` dilayani lebih belakangan daripada `b` bila kemendesakannya lebih rendah, atau bila kemendesakannya sama dan `a` datang lebih belakangan. Program itu lalu memperlihatkan antrean maks biasa dan antrean min berisi bilangan bulat.

Kode 14.7 `library_pq.cpp`: `std::priority_queue` dengan perbandingan buatan sendiri untuk melayani permintaan pemesanan, serta antrean maks bawaan dan antrean min `std::greater` berisi bilangan bulat.

```cpp
// library_pq.cpp -- std::priority_queue serving Pustaka Ceria hold requests.
#include <cstdio>
#include <functional>
#include <queue>
#include <string>
#include <vector>
using namespace std;

struct Request {
    int urgency;      // higher is more urgent
    int arrival;      // order of arrival: 1, 2, 3, ...
    string member;
    string title;
};

// "a is served later than b": the comparison priority_queue expects.
struct ServedLater {
    bool operator()(const Request& a, const Request& b) const {
        if (a.urgency != b.urgency) return a.urgency < b.urgency;
        return a.arrival > b.arrival;   // equal urgency: earlier arrival first
    }
};

int main() {
    priority_queue<Request, vector<Request>, ServedLater> holds;
    holds.push({70, 1, "Ayu", "Laskar Pelangi"});
    holds.push({90, 2, "Budi", "Bumi Manusia"});
    holds.push({40, 3, "Citra", "Saman"});
    holds.push({90, 4, "Dewi", "Pulang"});
    holds.push({55, 5, "Eko", "Amba"});
    holds.push({70, 6, "Fajar", "Cantik Itu Luka"});

    printf("Serving %zu hold requests (urgency, then arrival):\n", holds.size());
    while (!holds.empty()) {
        const Request& r = holds.top();              // O(1): the root
        printf("  urgency %2d  #%d  %-6s %s\n", r.urgency, r.arrival,
               r.member.c_str(), r.title.c_str());
        holds.pop();                                 // O(lg n): extract
    }
```

Kode 14.7 (Lanjutan) *library_pq.cpp: std::priority_queue dengan perbandingan buatan sendiri untuk melayani permintaan pemesanan, serta antrean maks bawaan dan antrean min std::greater berisi bilangan bulat.*

```cpp
    priority_queue<int> maxq;                        // largest on top
    priority_queue<int, vector<int>, greater<int>> minq;  // smallest on top
    for (int d : {12, 3, 8, 14, 5}) {
        maxq.push(d);
        minq.push(d);
    }
    printf("Days until due, largest first:");
    while (!maxq.empty()) { printf(" %d", maxq.top()); maxq.pop(); }
    printf("\nDays until due, smallest first:");
    while (!minq.empty()) { printf(" %d", minq.top()); minq.pop(); }
    printf("\n");
    return 0;
}
```

14.14.2 Java: `java.util.PriorityQueue`

`PriorityQueue<E>` di Java adalah *binary heap* di dalam larik yang dapat tumbuh, dan ia adalah *min-heap*: kepalanya adalah elemen *terkecil* menurut urutan alami atau menurut `Comparator` yang diberikan kepada konstruktornya [11]. `offer` dan `add` menyisipkan, `poll` mengambil kepala,

`peek` membacanya; dokumentasinya menyebut $O(\lg n)$ untuk penyisipan dan pengambilan kepala, waktu konstan untuk `peek` dan `size`, dan waktu linear untuk `remove(Object)` dan `contains(Object)`, yang harus mencari di seluruh larik [11]. Untuk antrean maks, berikan `Collections.reverseOrder()` atau pembanding yang dibalik. Dua fakta lagi dari dokumentasinya: seri di antara elemen yang sama diputus secara sembarang, sehingga bila urutan kedatangan penting, ia harus menjadi bagian dari pembandingnya; dan iteratornya mengunjungi elemen dalam urutan *larik*, yaitu urutan *heap*, bukan urutan terurut. Kelas ini tidak aman untuk banyak *thread*; `PriorityBlockingQueue` adalah versi konkurennya. **Kode 14.8** melayani permintaan pemesanan yang sama dengan pembanding yang mengurutkan menurut kemendesakan, dibalik, lalu menurut kedatangan.

Kode 14.8 `LibraryPQ.java`: `java.util.PriorityQueue` dengan pembanding yang dirakit dari `comparingInt`, `reversed`, dan `thenComparingInt`.

```
// LibraryPQ.java -- java.util.PriorityQueue serving Pustaka Ceria hold requests.
import java.util.Collections;
import java.util.Comparator;
import java.util.PriorityQueue;

public class LibraryPQ {
    record Request(int urgency, int arrival, String member, String title) {}

    public static void main(String[] args) {
        // Highest urgency first; equal urgency: earlier arrival first.
        Comparator<Request> served = Comparator.comparingInt(Request::urgency)
                .reversed().thenComparingInt(Request::arrival);
        PriorityQueue<Request> holds = new PriorityQueue<>(served);
        holds.add(new Request(70, 1, "Ayu", "Laskar Pelangi"));
        holds.add(new Request(90, 2, "Budi", "Bumi Manusia"));
        holds.add(new Request(40, 3, "Citra", "Saman"));
        holds.add(new Request(90, 4, "Dewi", "Pulang"));
        holds.add(new Request(55, 5, "Eko", "Amba"));
        holds.add(new Request(70, 6, "Fajar", "Cantik Itu Luka"));

        System.out.printf("Serving %d hold requests (urgency, then arrival):%n",
                holds.size());
        while (!holds.isEmpty()) {
            Request r = holds.poll();                    // O(lg n): extract
            System.out.printf("  urgency %2d  #%d  %-6s %s%n", r.urgency(),
                    r.arrival(), r.member(), r.title());
        }

        PriorityQueue<Integer> maxq =
                new PriorityQueue<>(Collections.reverseOrder());  // largest first
        PriorityQueue<Integer> minq = new PriorityQueue<>();   // smallest first
        for (int d : new int[] {12, 3, 8, 14, 5}) {
            maxq.add(d);
            minq.add(d);
        }
        StringBuilder a = new StringBuilder("Days until due, largest first:");
        while (!maxq.isEmpty()) a.append(" ").append(maxq.poll());
        StringBuilder b = new StringBuilder("Days until due, smallest first:");
        while (!minq.isEmpty()) b.append(" ").append(minq.poll());
        System.out.println(a);
        System.out.println(b);
    }
}
```

14.14.3 Python: `heapq`, termasuk fungsi max-heap yang baru

Python tidak memiliki kelas *priority queue* di intinya; sebagai gantinya modul `heapq` menyediakan fungsi-fungsi yang memperlakukan `list` biasa sebagai *binary min-heap* [12]. `heapq.heappush(h, x)` dan `heapq.heappop(h)` menyisipkan dan mengambil minimum, `h[0]` adalah minimumnya, `heapq.heapify(h)` membangun *heap* dalam waktu linear, `heapq.heapreplace(h, x)` mengambil minimum dan memasukkan `x` dengan satu kali *sift*, dan `heapq.heappushpop(h, x)` memasukkan lalu mengambil. Modul ini juga menyediakan aplikasi-aplikasi Subbab 14.15 dalam bentuk siap pakai: `heapq.nlargest(k, it)`, `heapq.nsmallest(k, it)`, dan `heapq.merge(*iterables)`.

Selama lebih dari dua puluh tahun modul ini tidak memiliki *max-heap* publik. Para pemrogram menyimpan kunci yang dinegasikan, $-x$, di dalam *min-heap*, yang berhasil untuk bilangan dan sama sekali tidak untuk *string*. **Python 3.14 menambahkan versi max-heap dari kelima fungsi inti**: `heapify_max`, `heappush_max`, `heappop_max`, `heappushpop_max`, dan `heapreplace_max`, semuanya bertanda "Added in version 3.14" di dokumentasinya [12]. **Kode 14.9** memakainya bila tersedia dan beralih ke kunci yang dinegasikan pada versi yang lebih lama, sehingga program itu berjalan, dengan keluaran yang sama persis, mulai dari Python 3.11. (Kami menjalankannya pada Python 3.11 dan pada Python 3.14.)

Kode 14.9 `library_pq.py`: `heapq` dengan *tuple* sebagai rekaman, dan fungsi *max-heap* Python 3.14 dengan cadangan untuk versi yang lebih lama.

```
# library_pq.py -- heapq serving Pustaka Ceria hold requests.
import heapq

try:                                    # Python 3.14 and newer: max-heap functions
    from heapq import heappop_max, heappush_max
except ImportError:                     # older Pythons: a min-heap of negated keys
    def heappush_max(heap, item):
        heapq.heappush(heap, -item)

    def heappop_max(heap):
        return -heapq.heappop(heap)

requests = [(70, 1, "Ayu", "Laskar Pelangi"), (90, 2, "Budi", "Bumi Manusia"),
            (40, 3, "Citra", "Saman"), (90, 4, "Dewi", "Pulang"),
            (55, 5, "Eko", "Amba"), (70, 6, "Fajar", "Cantik Itu Luka")]

holds = []                              # a min-heap of tuples, compared in order
for urgency, arrival, member, title in requests:
    heapq.heappush(holds, (-urgency, arrival, member, title))

print(f"Serving {len(holds)} hold requests (urgency, then arrival):")
while holds:
    neg, arrival, member, title = heapq.heappop(holds)   # O(lg n)
    print(f"  urgency {-neg:2d}  #{arrival}  {member:<6} {title}")

maxq, minq = [], []
for d in (12, 3, 8, 14, 5):
    heappush_max(maxq, d)               # largest on top
    heapq.heappush(minq, d)             # smallest on top
print("Days until due, largest first:",
      " ".join(str(heappop_max(maxq)) for _ in range(len(maxq))))
print("Days until due, smallest first:",
      " ".join(str(heapq.heappop(minq)) for _ in range(len(minq))))
```

Rekamannya berupa *tuple* `(-urgency, arrival, member, title)`. Python membandingkan *tuple* elemen demi elemen, dari kiri, sehingga *heap* mengurutkannya lebih dulu menurut kemendesakan yang dinegasikan (kemendesakan tertinggi berada di puncak *min-heap*) dan kedua menurut nomor kedatangan. Nomor kedatangan itu juga melakukan tugas kedua: ia

menjamin tidak ada dua *tuple* yang sepenuhnya sama, sehingga Python tidak pernah perlu membandingkan medan-medan sisanya. Tanpanya, dua permintaan dengan kemendesakan yang sama akan dibandingkan menurut nama pembaca, yang bukan kehendak meja sirkulasi; dan tipe rekaman yang sama sekali tidak memiliki urutan akan menimbulkan galat. Memakai pencacah sebagai elemen kedua adalah idiom baku [12].

Ketiga program mencetak:

```
Serving 6 hold requests (urgency, then arrival):
  urgency 90  #2  Budi   Bumi Manusia
  urgency 90  #4  Dewi   Pulang
  urgency 70  #1  Ayu    Laskar Pelangi
  urgency 70  #6  Fajar  Cantik Itu Luka
  urgency 55  #5  Eko    Amba
  urgency 40  #3  Citra  Saman
Days until due, largest first: 14 12 8 5 3
Days until due, smallest first: 3 5 8 12 14
```

Kedua permintaan berkemendesakan 90 dilayani menurut urutan kedatangan, #2 sebelum #4, demikian pula kedua permintaan berkemendesakan 70; aturan pemutus seri bekerja di ketiga bahasa. **Tabel 14.6** merangkum ketiga pustaka.

Tabel 14.6 *Priority queue* dalam ketiga pustaka standar. Ketiganya adalah *binary heap* di dalam larik.

	C++ `std::priority_queue`	Java `PriorityQueue`	Python `heapq`
Urutan bawaan	maks lebih dulu	min lebih dulu	min lebih dulu (fungsi maks sejak 3.14)
Membaca puncak	`top()`	`peek()`	`h[0]`
Menyisipkan	`push(x)`	`add(x)`, `offer(x)`	`heappush(h, x)`
Mengambil puncak	`pop()` (mengembalikan `void`)	`poll()`	`heappop(h)`
Membangun dari n butir	konstruktor, `make_heap`	konstruktor dari koleksi	`heapify(h)`
Urutan buatan sendiri	tipe `Compare` (“dilayani belakangan”)	`Comparator`	kunci *tuple*, atau kunci dinegasikan
Puncak antrean kosong	perilaku tak terdefinisi	`peek` mengembalikan `null`	`IndexError`
`decreaseKey`	tidak ada	tidak ada (hanya `remove` dalam $O(n)$)	tidak ada

Baris terakhir penting untuk subbab berikutnya. Tidak satu pun pustaka itu mengizinkan Anda mengubah kunci elemen yang sudah berada di dalam, karena tidak satu pun memberi tahu di mana elemen itu berada. Subbab 14.15.3 memperlihatkan bagaimana algoritma Dijkstra hidup tanpanya.

14.15 Aplikasi: Top-k, Menggabung k Senarai Terurut, dan Dijkstra

Heap membuktikan kegunaannya setiap kali sebuah program harus berulang kali menemukan yang terbaik dari himpunan yang terus berubah. Tiga persoalan klasik memperlihatkan tiga cara yang berbeda dalam memakainya.

14.15.1 k butir terbesar dari sebuah aliran data

Pustaka Ceria menginginkan tiga buku yang paling sering dipinjam tahun ini untuk etalase depannya. Jumlah peminjaman datang sebagai aliran data, bisa jadi jutaan banyaknya, dan kita menginginkan k yang terbesar. Mengurutkan semuanya berongkos waktu $O(n\lg n)$ dan memori $O(n)$. Membangun *max-heap* lalu mengambil k kali berongkos $O(n + k\lg n)$ tetapi tetap memerlukan memori $O(n)$.

Pemecahan yang mengejutkan memakai **min**-*heap* berukuran k. Ia menyimpan k butir terbaik yang terlihat sejauh ini, dengan yang terlemah di antaranya di akar. Untuk setiap butir baru, bandingkan dengan akar: bila ia tidak lebih baik daripada yang terlemah dari k teratas saat ini, ia tidak mungkin termasuk k teratas di akhir dan dibuang; bila lebih baik, ia menggantikan akar, yang lalu tenggelam ke tempatnya.

Pseudokode 14.10 `TOP-K`: simpan k butir terbaik di dalam *min-heap* yang akarnya adalah yang terlemah di antara mereka.

```
TOP-K(stream, k)                    // the k largest items, using a min-heap
 1  H = empty min-heap
 2  for each item x in stream
 3      if size(H) < k  then INSERT(H, x)
 4      else if x > MIN(H)
 5          replace the root of H by x; SIFT-DOWN at the root
 6  return the items of H             // sort them for display if needed
```

Setiap butir berongkos paling banyak satu *sift* di dalam *heap* berisi k elemen, sehingga

$$T_{\text{top-}k}(n) = O(n\lg k) \text{ waktu}, \qquad O(k) \text{ memori}. \tag{14.14}$$

Untuk $k = 3$ dan sejuta butir, itu berarti sejuta perbandingan dengan akar dan, pada kasus terburuk, beberapa pertukaran masing-masing, di dalam *heap* yang tidak pernah memuat lebih dari tiga butir. Batas memorinya inilah yang membuat metode ini tak tergantikan: ia bekerja pada aliran data yang terlalu besar untuk disimpan.

14.15.2 Menggabung k senarai terurut

Pustaka Ceria mempunyai tiga cabang, dan masing-masing menyimpan katalognya terurut menurut nomor buku. Katalog pusat memerlukan satu senarai terurut dari semuanya. Menggabung dua senarai terurut adalah langkah *merge* pada *merge sort* (Subbab 3.8). Untuk k senarai, membandingkan k elemen terdepan di setiap langkah berongkos $O(k)$ per elemen keluaran. *Min-heap* berisi k elemen terdepan bekerja lebih baik (**Pseudokode 14.11**): yang terkecil di antaranya berada di akar; keluarkan ia, dan gantikan dengan elemen berikutnya dari senarai yang sama.

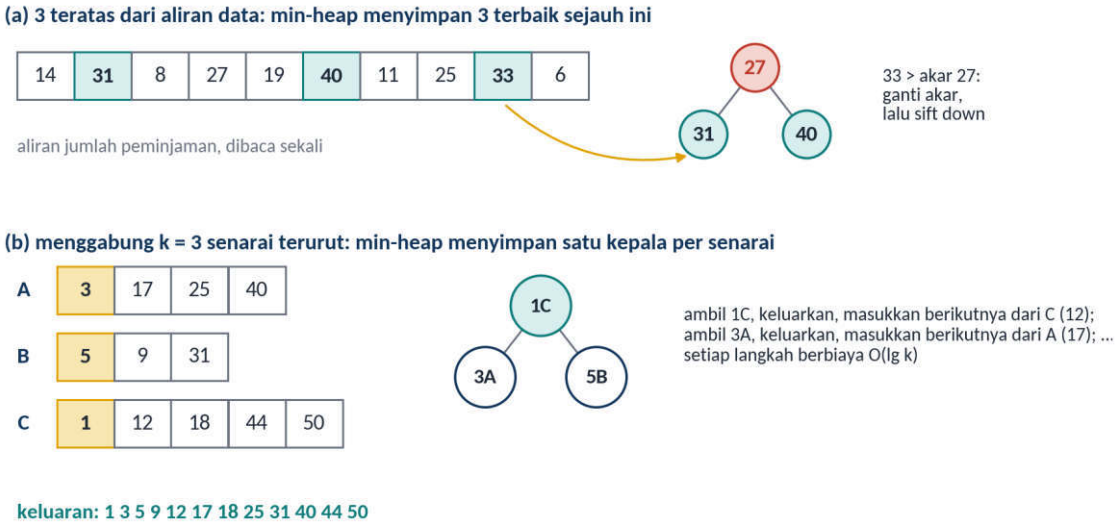
Pseudokode 14.11 `MERGE-K`: gabungkan k senarai terurut dengan *min-heap* yang memuat satu elemen terdepan dari setiap senarai.

```
MERGE-K(L_1, ..., L_k)              // each L_j is sorted
 1  H = empty min-heap of (value, list j, position)
 2  for j = 1 to k
 3      if L_j is not empty  then INSERT(H, (L_j[0], j, 0))
 4  while H is not empty
 5      (v, j, p) = EXTRACT-MIN(H)
 6      output v
 7      if p + 1 < length(L_j)  then INSERT(H, (L_j[p + 1], j, p + 1))
```

Heap itu tidak pernah memuat lebih dari k entri, dan masing-masing dari N elemen seluruhnya melewatinya satu kali, sehingga

$$T_{\text{merge}}(N, k) = O(N \lg k). \tag{14.15}$$

Gagasan yang sama mengurutkan data yang terlalu besar untuk memori (*external sorting*): urutkan potongan-potongan yang muat, tuliskan sebagai deretan terurut (*run*), lalu gabungkan semua deretan sekaligus dengan *heap*. **Gambar 14.12** menggambarkan kedua aplikasi itu, dan **Kode 14.10** menjalankannya.



Gambar 14.12 (a) 3 teratas dari sebuah aliran data: ketika 33 datang, *min-heap* memuat 27, 31, dan 40; karena $33 > 27$, ia menggantikan akar. Teal menandai 3 teratas di akhir aliran. (b) Menggabung tiga senarai terurut: *min-heap* memuat elemen terdepan setiap senarai (kuning tua), dan akarnya selalu menjadi keluaran berikutnya.

Kode 14.10 `topk_merge.cpp`: tiga buku yang paling sering dipinjam dari sebuah aliran data dengan *min-heap* berukuran 3, dan penggabungan tiga katalog cabang yang terurut.

```cpp
// topk_merge.cpp -- two classic heap applications: top-k and k-way merge.
#include <algorithm>
#include <cstdio>
#include <functional>
#include <queue>
#include <string>
#include <tuple>
#include <vector>
using namespace std;

int main() {
    // Part 1: the k most borrowed books, read as a stream, with a min-heap of k.
    vector<pair<int, string>> borrows = {
        {14, "Saman"}, {31, "Laskar Pelangi"}, {8, "Amba"},
        {27, "Bumi Manusia"}, {19, "Pulang"}, {40, "Negeri 5 Menara"},
        {11, "Supernova"}, {25, "Cantik Itu Luka"}, {33, "Ayat-Ayat Cinta"},
        {6, "Ronggeng Dukuh Paruk"}};
    const size_t k = 3;
    priority_queue<pair<int, string>, vector<pair<int, string>>,
                   greater<pair<int, string>>> top;       // smallest on top
    for (const auto& b : borrows) {
        if (top.size() < k) top.push(b);
        else if (b.first > top.top().first) {             // beats the weakest
            top.pop();
            top.push(b);
        }
        printf("read %2d %-21s -> weakest kept: %d\n", b.first, b.second.c_str(),
               top.top().first);
    }
    vector<pair<int, string>> best;
    while (!top.empty()) { best.push_back(top.top()); top.pop(); }
    reverse(best.begin(), best.end());
    printf("Top %zu:", k);
    for (size_t i = 0; i < best.size(); ++i)
        printf("%s %s (%d)", i ? "," : "", best[i].second.c_str(), best[i].first);
    sort(borrows.rbegin(), borrows.rend());               // check: sort everything
    bool same = equal(best.begin(), best.end(), borrows.begin());
    printf("\ncheck against a full sort: %s\n\n", same ? "yes" : "no");
```

Kode 14.10 (Lanjutan) *topk_merge.cpp: tiga buku yang paling sering dipinjam dari sebuah aliran data dengan min-heap berukuran 3, dan penggabungan tiga katalog cabang yang terurut.*

```cpp
    // Part 2: merge k sorted lists of book numbers with a min-heap of k heads.
    vector<vector<int>> branch = {{3, 17, 25, 40}, {5, 9, 31}, {1, 12, 18, 44, 50}};
    const char* name = "ABC";
    using Head = tuple<int, int, int>;                    // value, list, position
    priority_queue<Head, vector<Head>, greater<Head>> heads;
    for (int b = 0; b < static_cast<int>(branch.size()); ++b)
        heads.push({branch[b][0], b, 0});
    printf("Merged catalog:");
    while (!heads.empty()) {
        auto [value, b, pos] = heads.top();
        heads.pop();
        printf(" %d%c", value, name[b]);
        if (pos + 1 < static_cast<int>(branch[b].size()))
            heads.push({branch[b][pos + 1], b, pos + 1});  // that list's next
    }
    printf("\n");
    return 0;
}
```

```
read 14 Saman                -> weakest kept: 14
read 31 Laskar Pelangi       -> weakest kept: 14
read  8 Amba                 -> weakest kept: 8
read 27 Bumi Manusia         -> weakest kept: 14
read 19 Pulang               -> weakest kept: 19
read 40 Negeri 5 Menara      -> weakest kept: 27
read 11 Supernova            -> weakest kept: 27
read 25 Cantik Itu Luka      -> weakest kept: 27
read 33 Ayat-Ayat Cinta      -> weakest kept: 31
read  6 Ronggeng Dukuh Paruk -> weakest kept: 31
Top 3: Negeri 5 Menara (40), Ayat-Ayat Cinta (33), Laskar Pelangi (31)
check against a full sort: yes

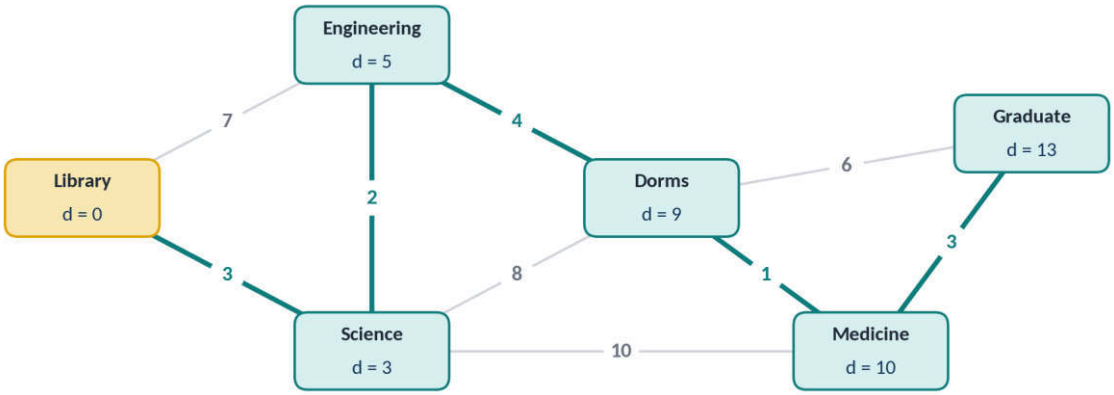
Merged catalog: 1C 3A 5B 9B 12C 17A 18C 25A 31B 40A 44C 50C
```

Kolom "weakest kept" adalah akar *min-heap* setelah setiap butir. Nilainya naik dari 8 ke 14, 19, 27, dan akhirnya 31 ketika butir-butir yang lebih baik menyingkirkan yang lebih lemah; butir 11, 25, dan 6 tidak pernah masuk karena tidak mengalahkan akar. Katalog gabungan memberi label cabang pada setiap nomor. Di C++ *min-heap*-nya adalah `priority_queue` dengan `std::greater`; pasangan dan *tuple* dibandingkan secara leksikografis, sehingga kepala-kepala senarai terurut menurut nilainya. Versi Python, `topk_merge.py`, memakai `heapq.heapreplace` untuk baris 5 **Pseudokode 14.10**, dan memastikan jawabannya dengan `heapq.nlargest`, yang mengimplementasikan algoritma yang sama; `heapq.merge` mengerjakan seluruh Bagian 2 dalam satu panggilan. `TopKMerge.java` adalah versi Java-nya.

14.15.3 Kait ke Bab 13: lintasan terpendek Dijkstra

Subbab 13.10 menemukan rute terpendek pada peta kampus berbobot dengan algoritma Dijkstra [13]: algoritma itu berulang kali mengambil simpul yang belum final dengan jarak sementara terkecil dari *min-priority queue*, menyatakan jarak itu final, dan *merelaksasi* sisi-sisi yang keluar darinya, menurunkan jarak sementara $d[v]$ milik tetangga v bila rute lewat u lebih pendek. Subbab 13.10.6 lalu mencatat bahwa semua bagian waktu jalannya yang tidak linear berasal dari *priority queue*, dan menyerahkan pertanyaan itu kepada bab ini. Kini kita dapat menjawabnya dari sisi *heap*.

Menurunkan $d[v]$ adalah sebuah `decreaseKey` pada *min-heap*. Seperti yang diperlihatkan Subbab 14.14, antrean pustaka tidak memiliki `decreaseKey`, dan itulah sebabnya Subbab 13.10.3 memakai versi *malas* (*lazy*): alih-alih mengubah entri v, masukkan entri *baru* $(d[v], v)$ dan biarkan entri lama yang lebih besar tetap di dalam *heap*; ketika entri lama itu kelak mencapai puncak, simpulnya sudah final, dan entri itu cukup dilewati. Berapa ongkosnya bagi *heap*? Untuk melihatnya, kita menjalankan algoritma malas itu pada peta kecil kedua. Mobil pengantar Pustaka Ceria membawa buku dari perpustakaan pusat ke lima gedung lain di kampus; **Gambar 14.13** memperlihatkan jalan-jalannya beserta waktu tempuhnya dalam menit.



Gambar 14.13 Jalan-jalan pengantaran dengan waktu tempuh dalam menit, dan waktu pengantaran terpendek *d* dari Library ke setiap gedung. Sisi-sisi teal membentuk pohon lintasan terpendek yang ditemukan *priority queue*.

Kode 14.11 mencetak setiap entri yang keluar dari *heap*.

Kode 14.11 `dijkstra_pq.cpp`: algoritma Dijkstra pada jalan-jalan pengantaran dengan `std::priority_queue` sebagai *min-heap* dan penghapusan malas.

```cpp
// dijkstra_pq.cpp -- shortest delivery times with a min-priority queue.
#include <cstdio>
#include <functional>
#include <queue>
#include <utility>
#include <vector>
using namespace std;

const char* place[] = {"Library", "Engineering", "Science", "Dorms",
                       "Medicine", "Graduate"};

int main() {
    const int n = 6, INF = 1000000000;
    vector<vector<pair<int, int>>> adj(n);          // adj[u] = {(v, minutes)}
    auto road = [&](int u, int v, int w) {
        adj[u].push_back({v, w});
        adj[v].push_back({u, w});
    };
    road(0, 1, 7); road(0, 2, 3); road(2, 1, 2); road(1, 3, 4); road(2, 3, 8);
    road(3, 4, 1); road(2, 4, 10); road(4, 5, 3); road(3, 5, 6);

    vector<int> dist(n, INF), prev(n, -1);
    vector<bool> done(n, false);
    using Entry = pair<int, int>;                    // (distance, vertex)
    priority_queue<Entry, vector<Entry>, greater<Entry>> pq;   // smallest first
    dist[0] = 0;
    pq.push({0, 0});
    while (!pq.empty()) {
        auto [d, u] = pq.top();
        pq.pop();
        if (done[u]) {                               // an out-of-date entry
            printf("pop %2d %-12s stale, skipped\n", d, place[u]);
            continue;
        }
        done[u] = true;
        printf("pop %2d %-12s settled\n", d, place[u]);
        for (auto [v, w] : adj[u])
            if (!done[v] && d + w < dist[v]) {        // a shorter way to v
                dist[v] = d + w;
                prev[v] = u;
                pq.push({dist[v], v});
            }
    }
    printf("\nFastest delivery from the Library:\n");
    for (int v = 0; v < n; ++v) {
        printf("  %-11s %2d min  via", place[v], dist[v]);
```

Kode 14.11 (Lanjutan) *dijkstra_pq.cpp: algoritma Dijkstra pada jalan-jalan pengantaran dengan std::priority_queue sebagai min-heap dan penghapusan malas.*

```cpp
        vector<int> path;
        for (int x = v; x != -1; x = prev[x]) path.push_back(x);
        for (int i = static_cast<int>(path.size()) - 1; i >= 0; --i)
            printf(" %s", place[path[i]]);
        printf("\n");
    }
    return 0;
}
```

```
pop  0 Library      settled
pop  3 Science      settled
pop  5 Engineering  settled
pop  7 Engineering  stale, skipped
pop  9 Dorms        settled
pop 10 Medicine     settled
pop 11 Dorms        stale, skipped
pop 13 Medicine     stale, skipped
pop 13 Graduate     settled
pop 15 Graduate     stale, skipped

Fastest delivery from the Library:
  Library      0 min  via Library
  Engineering  5 min  via Library Science Engineering
  Science      3 min  via Library Science
  Dorms        9 min  via Library Science Engineering Dorms
  Medicine    10 min  via Library Science Engineering Dorms Medicine
  Graduate    13 min  via Library Science Engineering Dorms Medicine Graduate
```

Ikuti antreannya. Library menjadi final pada 0 dan memasukkan Engineering pada 7 dan Science pada 3. Science, pada 3, keluar berikutnya dan menemukan jalan yang lebih pendek ke Engineering, $3 + 2 = 5 < 7$: entri baru (5, Engineering) dimasukkan, dan entri lama (7, Engineering) tertinggal. Engineering menjadi final pada 5, dan belakangan entri basi (7, Engineering) keluar dan dilewati. Hal yang sama terjadi pada Dorms (11 digantikan 9), Medicine (13 oleh 10), dan Graduate (15 oleh 13). Seluruhnya empat entri basi, dan setiap waktu terpendek sesuai dengan **Gambar 14.13**.

Dengan penghapusan malas, setiap relaksasi yang berhasil memasukkan satu entri, paling banyak satu per arah sisi, sehingga *heap* memuat $O(|E|)$ entri, dan setiap penyisipan atau pengambilan berongkos $O(\lg|E|) = O(\lg|V|)$, karena $|E| < |V|^2$. Totalnya adalah batas Subbab 13.10.6, kini dengan setiap faktornya terjelaskan:

$$T_{\text{Dijkstra}} = O((|V| + |E|)\lg|V|). \tag{14.16}$$

Dengan *heap* berindeks yang mendukung `decreaseKey` sungguhan, Subbab 14.9, batasnya sama tetapi *heap* tidak pernah memuat lebih dari $|V|$ entri. Sepuluh penyisipan pada eksekusi kita, untuk sembilan jalan dan enam gedung, memperlihatkan harga kemalasan: empat dari sepuluh entri itu basi. *Heap* yang lebih canggih memperbaiki batasnya: *Fibonacci heap* dari Fredman dan Tarjan membuat `decreaseKey` berongkos $O(1)$ teramortisasi dan membawa algoritma Dijkstra ke $O(|E| + |V|\lg|V|)$ [14], seperti yang disebut Subbab 13.10.6. Pada graf biasa, *binary heap* sederhana dengan penghapusan malas biasanya yang tercepat dalam praktik, dan itulah sebabnya ia dipakai sebagian besar program. Versi Java dan Python, `DijkstraPQ.java` dan `dijkstra_pq.py`, mencetak jejak yang sama; di Python antreannya berupa *list* berisi *tuple* `(distance, vertex)` yang dikelola `heapq`.

14.16 Profitina dalam Praktik

Profitina adalah platform visibilitas AI (GEO): ia mengukur seberapa siap situs web sebuah merek untuk dikutip oleh mesin jawaban AI dan membantu memperbaiki apa yang ditemukannya [15]. Servernya adalah aplikasi Python yang dibangun dengan FastAPI, sehingga modul `heapq` dari Subbab 14.14.3 adalah *heap* yang paling dekat di tangan. Beberapa fiturnya memiliki bentuk yang persis sama dengan bab ini. Kami menggambarkan bentuk setiap persoalan dan golongan struktur yang cocok untuknya; pilihan implementasi sesungguhnya di dalam produk adalah urusan produk itu sendiri.

- **Apa yang jatuh tempo berikutnya.** Fitur analitik historis memindai ulang setiap merek secara otomatis, dengan irama yang bergantung pada tingkat langganannya. Penjadwal yang menghadapi banyak merek dengan irama berbeda-beda mengajukan satu pertanyaan berulang-ulang: *pemindaian mana yang paling segera jatuh tempo?* Itu adalah *min-priority queue* yang berkunci waktu jatuh tempo berikutnya. `extractMin` memberi pekerjaan berikutnya dalam $O(\lg n)$; setelah pemindaian, merek itu dimasukkan kembali dengan waktu jatuh tempo berikutnya, satu `insert`. Dibanding memindai waktu jatuh tempo setiap merek pada setiap detak, $\Theta(n)$ per detak, *heap* membuat ongkos penjadwal bergantung pada banyaknya pemindaian yang sungguh dijalankan, bukan pada banyaknya merek. Bentuk yang sama, dengan kunci kemendesakan alih-alih waktu, muncul ketika sebagian pekerjaan lebih mendesak daripada yang lain, seperti pesanan Profitina Laundry pada Soal 14.12.
- **Puncak papan peringkat.** Pengguna memperoleh PROFIT dengan memperbaiki respons AI, dan papan peringkat (*leaderboard*) memperlihatkan siapa yang memimpin. Menampilkan k teratas dari banyak pengguna adalah persoalan top-k Subbab 14.15.1: *min-heap* berukuran k menemukannya dalam satu lintasan dengan waktu $O(n \lg k)$ tanpa mengurutkan semua orang, dan menghitung ulang secara berkala daftar teratas yang kecil tetap murah berapa pun banyaknya pengguna.
- **Dasbor yang digabung dari banyak umpan.** Dasbor menghimpun, dalam satu panggilan, berbagai jenis notifikasi: balasan, suka, *repost*, pesan langsung, sematan, status umpan balik, pembelian PROFIT. Setiap jenis dapat dibaca dari sumbernya sendiri yang sudah terurut menurut waktu, terbaru lebih dulu. Menampilkannya dalam satu linimasa adalah penggabungan k senarai Subbab 14.15.2 dengan k sama dengan banyaknya sumber: $O(N \lg k)$ untuk N notifikasi, dan bila hanya 20 yang terbaru yang ditampilkan, penggabungan dapat berhenti setelah 20 keluaran.
- **Garis depan penjelajahan.** Memindai sebuah situs berarti menemukan halaman-halaman dan memutuskan mana yang diambil berikutnya. Penjelajah (*crawler*) yang mendahulukan halaman paling penting, misalnya yang paling dekat dengan beranda atau yang paling sering ditaut, menyimpan garis depannya (*frontier*) di dalam *priority queue*; penjelajah yang sekadar mengambil halaman menurut urutan penemuannya sedang menjalankan penelusuran melebar Bab 13 dengan *queue* FIFO.

Dalam setiap kasus, *heap* menggantikan pemindaian penuh yang berulang atau pengurutan penuh yang berulang dengan satu langkah logaritmik, dan itulah seluruh pesan bab ini.

14.17 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Mencampur rumus berbasis 0 dan berbasis 1. Dengan akar di indeks 0, anak-anak i adalah $2i + 1$ dan $2i + 2$; dengan akar di 1, anak-anaknya $2k$ dan $2k + 1$. Memakai $2i$ dan $2i + 1$ pada larik berbasis 0 menjadikan akar sebagai anak kirinya sendiri dan diam-diam merusak pohon. Pilih satu penomoran (**Tabel 14.3**) dan pakailah di mana-mana.

Melakukan sift-down ke arah anak yang salah. *Sift-down* harus bertukar dengan anak yang *lebih besar* pada *max-heap* (yang lebih kecil pada *min-heap*). Bertukar dengan anak kiri, atau dengan anak pertama yang ternyata lebih besar, dapat menaruh kunci yang lebih kecil di atas saudara yang lebih besar (Subbab 14.6.1).

Melupakan anak yang tidak ada. Anak kanan simpul internal terakhir mungkin tidak ada; ujilah `l < n` dan `r < n` sebelum membaca `A[l]` dan `A[r]`. Pada *heap sort*, n adalah ukuran *heap* yang *menyusut*, bukan ukuran larik, sebab jika tidak ekor terurut akan tertarik kembali ke dalam *heap*.

Mengambil akar tanpa memindahkan daun terakhir. `extractMax` harus menaruh elemen terakhir di akar sebelum *sift-down*; menghapus akar lalu menaikkan anak-anak meninggalkan celah dan merusak sifat *complete* (Subbab 14.7.3).

Menganggap heap terurut. Hanya akar yang diketahui; larik secara keseluruhan tidak berada dalam urutan yang berguna, dan iterasi atas `PriorityQueue` di Java atau atas *list* `heapq` di Python mengunjungi elemen dalam urutan *heap*, bukan urutan terurut. Untuk memperoleh keluaran terurut, ambillah berulang kali.

Mencari di heap seperti di pohon pencarian. Sebuah kunci bisa berada di subpohon mana pun; menemukan kunci sembarang memerlukan $\Theta(n)$. Itu pula sebabnya `getMin` pada *max-heap* bernilai $\Theta(n)$ (Subbab 14.8).

Membangun lewat n penyisipan padahal semua kunci sudah diketahui. Pakailah pembangunan dari bawah ke atas, $\Theta(n)$ alih-alih $\Theta(n \lg n)$ pada kasus terburuk (**Kode 14.2**); mulailah di indeks $\lfloor n/2 \rfloor - 1$, bukan di $n - 1$ (daun-daun tidak perlu dikerjakan) dan bukan di 0 sambil bergerak naik (*sift-down* memerlukan kedua subpohonnya sudah menjadi *heap*).

Mengubah kunci tanpa memperbaiki. Penugasan `A[i] = k` di dalam *heap* harus diikuti *sift-up* (kunci lebih besar) atau *sift-down* (kunci lebih kecil). `increaseKey` yang menerima nilai lebih kecil lalu melakukan *sift-up* meninggalkan *heap* yang rusak tanpa galat apa pun (Subbab 14.9).

Menulis perbandingan `std::priority_queue` secara terbalik. Perbandingannya menjawab "apakah `a` dilayani lebih belakangan daripada `b`?"; dengan `less` antreannya adalah *max-heap*. Untuk "tenggat paling awal lebih dulu", pakailah `greater`, atau perbandingan yang mengembalikan `true` bila tenggat `a` lebih lambat. Dan bacalah `top()` sebelum `pop()`, yang tidak mengembalikan apa-apa.

Menyerahkan seri pada kebetulan. *Heap* tidak stabil, dan tidak satu pun pustaka menjanjikan urutan di antara kunci yang sama. Bila prioritas yang sama harus dilayani menurut urutan kedatangan, jadikan nomor kedatangan bagian dari kuncinya (Subbab 14.14).

Aritmetika tak bertanda pada batas gelung. Di C++, `v.size() - 1` atau `(v.size() - 2) / 2` melingkar untuk larik yang kecil; ujilah kasus kosong dan kasus satu elemen (Subbab 14.11).

Mencampuradukkan kedua heap. Wilayah memori Subbab 1.3.5 dan struktur data bab ini hanya berbagi nama.

14.18 Ringkasan Bab

- *Priority queue* melayani butir dengan kunci tertinggi (atau terendah) lebih dulu. Larik tak terurut dan larik terurut masing-masing membuat salah satu dari `insert` dan `extractMax` menjadi linear; *binary heap* membuat keduanya $O(\lg n)$ dan `getMax` $O(1)$, tanpa memori tambahan (**Tabel 14.2**). Ia diperkenalkan Williams pada 1964, dengan pembangunan waktu linear dari Floyd.
- **Pohon biner complete**, yang setiap levelnya penuh kecuali yang terakhir, yang terisi dari kiri, disimpan level demi level di dalam larik. Dengan akar di indeks 0, induk i adalah $\lfloor (i-1)/2 \rfloor$ dan anak-anaknya $2i+1$ dan $2i+2$ (Persamaan (14.1)); dengan akar di 1, induk k adalah $\lfloor k/2 \rfloor$ dan anak-anaknya $2k$ dan $2k+1$ (Persamaan (14.2)). Tingginya $\lfloor \lg n \rfloor$ (Persamaan (14.4)), dan daun-daunnya adalah indeks $\lfloor n/2 \rfloor$ sampai $n-1$ (Persamaan (14.5)).
- **Max-heap** memiliki setiap simpul yang tidak lebih besar daripada induknya (Persamaan (14.6)), sehingga akar adalah maksimum dan setiap lintasan akar-ke-daun tidak

naik; **min-heap** adalah bayangan cerminnya (Persamaan (14.7)). Saudara kandung, level, dan larik secara keseluruhan tidak terurut.

- **Sift-up** menaikkan kunci yang terlalu besar ke arah akar: paling banyak $\lfloor \lg n \rfloor$ pertukaran. **Sift-down** menurunkan kunci yang terlalu kecil ke arah daun, selalu bertukar dengan anak yang lebih besar: paling banyak $2\lfloor \lg n \rfloor$ perbandingan (Persamaan (14.8)).
- `insert` menambahkan di ujung lalu *sift-up*; `extractMax` memindahkan daun terakhir ke akar lalu *sift-down*; keduanya $O(\lg n)$. `getMax` bernilai $O(1)$. `getMin` pada *max-heap* memindai daun-daun dan bernilai $\Theta(n)$, dan tidak ada metode yang bisa lebih baik, karena daun mana pun bisa memuat minimum.
- `increaseKey` melakukan *sift-up* dan `decreaseKey` melakukan *sift-down*, masing-masing $O(\lg n)$, tetapi pemanggil harus mengetahui indeks butirnya.
- `buildMaxHeap` melakukan *sift-down* pada simpul-simpul internal dari yang terakhir sampai akar, dalam waktu $\Theta(n)$: kurang dari n pertukaran dan $2n$ perbandingan (Persamaan (14.9), Persamaan (14.10), Persamaan (14.11)). Membangun lewat n penyisipan berongkos $\Theta(n \lg n)$ pada kasus terburuk (Persamaan (14.12)), meskipun hanya linear pada kunci acak.
- `isMaxHeap` memeriksa setiap anak terhadap induknya dalam $O(n)$.
- **Heap sort** membangun *max-heap* lalu berulang kali memindahkan akar ke ujung *heap* yang menyusut: $O(n \lg n)$ perbandingan pada setiap kasus (Persamaan (14.13)), memori tambahan $O(1)$, **tidak stabil** (**Tabel 14.5**).
- **Pustaka**: `std::priority_queue` (maks lebih dulu, perbandingannya berarti "dilayani belakangan", `pop` mengembalikan `void`), `PriorityQueue` di Java (min lebih dulu), dan `heapq` di Python (min lebih dulu, dengan `heapify_max`, `heappush_max`, `heappop_max`, `heappushpop_max`, dan `heapreplace_max` sejak Python 3.14) (**Tabel 14.6**).
- **Aplikasi**: k teratas dari sebuah aliran data dengan *min-heap* berukuran k, $O(n \lg k)$ (Persamaan (14.14)); penggabungan k senarai terurut, $O(N \lg k)$ (Persamaan (14.15)); dan lintasan terpendek Dijkstra dengan penghapusan malas, $O((|V| + |E|) \lg |V|)$ (Persamaan (14.16)).

14.19 Soal Latihan

14.1 (a) Pada *heap* berbasis 0, tentukan induk dan kedua anak indeks 13. (b) Tentukan hal yang sama untuk simpul yang sama dalam penomoran berbasis 1. (c) Pada *heap* berisi $n = 20$ kunci, apakah indeks 13 (berbasis 0) sebuah daun? Indeks mana saja yang merupakan daun?

14.2 Manakah di antara larik berikut yang merupakan *max-heap*? Untuk setiap larik yang bukan, sebutkan sepasang indeks yang melanggar Persamaan (14.6). (a) 50, 30, 40, 10, 20, 35, 5; (b) 50, 40, 30, 45, 10; (c) 9, 9, 9; (d) 10, 9, 8, 7, 6, 5; (e) 20, 15, 18, 16, 14, 17.

14.3 Mulai dari *max-heap* kosong, sisipkan 15, 40, 20, 35, 50, 10, dan 45 dengan urutan itu. Tuliskan larik setelah setiap penyisipan, dan hitung pertukaran yang dilakukan setiap *sift-up*.

14.4 Pada *heap* hasil Soal 14.3, panggil `extractMax` tiga kali. Tuliskan nilai yang dikembalikan dan larik setelah setiap panggilan.

14.5 Jalankan `BUILD-MAX-HEAP` dengan tangan pada larik 1, 2, 3, 4, 5, 6, 7. Tuliskan larik setelah setiap panggilan *sift-down* dan banyak pertukaran seluruhnya, lalu bandingkan total itu dengan batas $n - \lg(n + 1)$ dari Subbab 14.10.2.

14.6 (a) Berapa banyak kunci paling sedikit dan paling banyak pada *heap* setinggi h? (b) Pada *max-heap* dengan kunci-kunci berbeda, di indeks mana saja kunci terbesar kedua bisa berada?

Kunci terbesar ketiga? (c) Dapatkah kunci terkecil *max-heap* dengan kunci-kunci berbeda berada di simpul internal?

14.7 *Heap d-ary* mirip *binary heap*, kecuali bahwa setiap simpul memiliki hingga d anak. (a) Dengan akar di indeks 0, tentukan indeks anak-anak dan induk indeks i. (b) Berapa tinggi *heap d-ary* berisi n kunci? (c) Berapa perbandingan yang dilakukan *sift-up* dan *sift-down* pada kasus terburuk? (d) Mengapa *heap* 4-ary mungkin lebih disukai untuk algoritma Dijkstra pada graf yang sisinya jauh lebih banyak daripada simpulnya?

14.8 (a) Urutkan 4, 10, 3, 5, 1 dengan *heap sort* secara manual, dengan menuliskan larik setelah pembangunan dan setelah setiap iterasi fase 2. (b) Jelaskan cara membuat *heap sort* stabil, dan berapa ongkosnya.

14.9 Tulislah (a) `bool isMinHeap(const vector<int>& a)`, dan (b) `int kthLargest(const vector<int>& heap, int k)`, yang mengembalikan kunci terbesar ke-k dari sebuah *max-heap* tanpa mengubahnya, dalam waktu $O(k \lg k)$. (Petunjuk: kunci terbesar adalah akar; calon berikutnya selalu anak dari kunci yang sudah diambil.)

14.10 Tulislah `void removeAt(vector<int>& heap, int index)`, yang menghapus kunci di indeks mana pun dari *max-heap* dalam waktu $O(\lg n)$. Ujilah pada kasus ketika kunci yang dipindahkan harus naik dan pada kasus ketika ia harus turun.

14.11 Sebuah aliran bilangan datang satu per satu. Setelah setiap bilangan, cetak median semua bilangan sejauh ini, dalam waktu $O(\lg n)$ per bilangan. (Petunjuk: simpan separuh yang lebih kecil di dalam *max-heap* dan separuh yang lebih besar di dalam *min-heap*.) Pakailah aliran 5, 15, 1, 3, 8, 7, 9, 10.

14.12 Konter Profitina Laundry menerima pesanan. Setiap pesanan memiliki nomor pengenal, jenis (ekspres atau reguler), waktu pengambilan, dan nomor kedatangan. Mesin cuci harus mengerjakan pesanan ekspres sebelum semua pesanan reguler, lalu waktu pengambilan paling awal, lalu kedatangan paling awal. Tulislah antreannya dengan *priority queue* pustaka dan cetak urutan pencucian untuk: 101 reguler, ambil 17.00, kedatangan 1; 102 ekspres, 15.00, 2; 103 reguler, 12.00, 3; 104 ekspres, 11.30, 4; 105 reguler, 12.00, 5; 106 ekspres, 15.00, 6.

14.13 Buktikan bahwa $\sum_{m=1}^{n} \lfloor \lg m \rfloor = (n+1)\lfloor \lg n \rfloor - 2^{\lfloor \lg n \rfloor + 1} + 2$, bentuk tertutup Persamaan (14.12), dan periksa terhadap ketiga total "ascending" pada **Kode 14.2**.

14.14 Tulis ulang `SIFT-UP` sehingga ia memindahkan kunci ke tempatnya dengan satu operasi tulis terakhir, alih-alih pertukaran di setiap level (Subbab 14.5.3). Berapa banyak operasi tulis ke larik yang dilakukannya ketika kunci naik s level, dibandingkan dengan **Pseudokode 14.1**?

14.15 (a) Pada algoritma Dijkstra versi malas dari Subbab 13.10.3 dan **Kode 14.11**, jelaskan mengapa melewati entri yang simpulnya sudah final itu benar. (b) Berapa banyak entri paling banyak yang dapat dimuat *priority queue*, untuk graf dengan $|V|$ simpul dan $|E|$ sisi? (c) Apa yang akan salah bila secara keliru dipakai *max-heap*?

14.20 Penyelesaian Soal Latihan

14.1 (a) Menurut Persamaan (14.1): induk $\lfloor 12/2 \rfloor = 6$, anak-anak $2 \cdot 13 + 1 = 27$ dan 28. (b) Simpul yang sama berindeks berbasis 1 14; menurut Persamaan (14.2): induk $\lfloor 14/2 \rfloor = 7$, anak-anak 28 dan 29. Setiap jawaban lebih besar satu daripada di (a), sebagaimana mestinya. (c) Menurut Persamaan (14.5) daun-daun *heap* berisi 20 kunci adalah indeks $\lfloor 20/2 \rfloor = 10$ sampai 19, jadi indeks 13 adalah daun; memang anak kirinya, 27, akan berada di luar larik.

14.2 (a) Ya: 30 dan 40 paling besar 50; 10 dan 20 paling besar 30; 35 dan 5 paling besar 40. (b) Bukan: indeks 3 memuat 45, induknya indeks 1 memuat 40, dan $40 < 45$. (c) Ya: kunci yang sama tidak melanggar $\geq$. (d) Ya: larik yang terurut menurun selalu merupakan *max-heap*, karena setiap induk muncul lebih awal. (e) Bukan: indeks 3 memuat 16 dan induknya, indeks 1, memuat 15.

14.3 Lariknya adalah: 15; 40, 15 (satu pertukaran); 40, 15, 20 (tidak ada); 40, 35, 20, 15 (satu: 35 melewati 15); 50, 40, 20, 15, 35 (dua: 50 ditempatkan di indeks 4, melewati 35 di indeks 1, lalu 40 di akar); 50, 40, 20, 15, 35, 10 (tidak ada); 50, 40, 45, 15, 35, 10, 20 (satu: 45 melewati 20). Seluruhnya lima pertukaran.

14.4 Panggilan pertama mengembalikan 50: 20 pindah ke akar, tenggelam di bawah 45, dan *heap* menjadi 45, 40, 20, 15, 35, 10. Yang kedua mengembalikan 45: 10 pindah ke akar, tenggelam di bawah 40 lalu di bawah 35, menghasilkan 40, 35, 20, 15, 10. Yang ketiga mengembalikan 40: 10 pindah ke akar, tenggelam di bawah 35 lalu di bawah 15, menghasilkan 35, 15, 20, 10.

14.5 Simpul-simpul internalnya adalah 2, 1, dan 0. `siftDown(2)`: 3 lebih kecil daripada anak-anaknya 6 dan 7 dan tenggelam di bawah 7: 1, 2, 7, 4, 5, 6, 3. `siftDown(1)`: 2 tenggelam di bawah yang lebih besar di antara 4 dan 5: 1, 5, 7, 4, 2, 6, 3. `siftDown(0)`: 1 tenggelam di bawah 7, lalu di bawah 6: 7, 5, 6, 4, 2, 1, 3. Seluruhnya empat pertukaran, sama dengan $n - \lg(n+1) = 7 - 3 = 4$: larik naik adalah kasus terburuk bagi pembangunan, dan batas Subbab 14.10.2 tercapai dengan tepat.

14.6 (a) Paling sedikit 2^h (level-level di atas yang terakhir penuh, $2^h - 1$ kunci, dan level terakhir paling sedikit berisi satu) dan paling banyak $2^{h+1} - 1$ (semua level penuh), menurut Persamaan (14.3). (b) Kunci terbesar kedua lebih besar daripada semuanya kecuali akar, sehingga induknya pasti akar: ia berada di indeks 1 atau 2. Kunci terbesar ketiga hanya memiliki akar dan kunci terbesar kedua di atasnya, sehingga ia anak akar atau anak kunci terbesar kedua: indeks mana pun dari 1 sampai 6 (kedalaman 1 atau 2). (c) Tidak; Subbab 14.8 membuktikan bahwa dengan kunci-kunci berbeda, minimum hanya berada di daun.

14.7 (a) Anak-anak i adalah $di + 1, di + 2, \ldots, di + d$, dan induk $i \geq 1$ adalah $\lfloor (i-1)/d \rfloor$; untuk $d = 2$ inilah Persamaan (14.1). (b) Level j memuat hingga d^j kunci, sehingga tingginya sekitar $\log_d n$; tepatnya, h terkecil yang memenuhi $1 + d + \cdots + d^h \geq n$. (c) *Sift-up* membandingkan dengan satu induk per level: sekitar $\log_d n$ perbandingan. *Sift-down* harus mencari yang terbesar di antara hingga d anak per level, d perbandingan per level (termasuk perbandingan dengan kunci itu sendiri): sekitar $d \log_d n$. (d) Algoritma Dijkstra melakukan satu pengambilan per simpul tetapi hingga satu penyisipan (atau `decreaseKey`) per sisi. Penyisipan adalah *sift-up*, yang makin murah seiring bertambahnya d, sementara pengambilan makin mahal; bila sisinya jauh lebih banyak daripada simpulnya, penyisipan yang lebih murah itu yang menang. *Heap* 4-ary juga menaruh keempat anak sebuah simpul bersebelahan di memori, yang cocok dengan *cache* prosesor.

14.8 (a) Pembangunan menghasilkan 10, 5, 3, 4, 1 (4 tenggelam di bawah 10 lalu di bawah 5). Fase 2: tukar 10 dan 1 lalu *sift-down*: 5, 4, 3, 1 | 10. Tukar 5 dan 1 lalu *sift-down*: 4, 1, 3 | 5, 10. Tukar 4 dan 3: 3, 1 | 4, 5, 10. Tukar 3 dan 1: 1 | 3, 4, 5, 10. Hasilnya 1, 3, 4, 5, 10. (b) Urutkan pasangan (kunci, indeks asal) dengan perbandingan "kunci lebih besar, atau kunci sama dan indeks lebih besar, datang belakangan". Dengan begitu tidak ada dua pasangan yang sama, dan kunci-kunci yang sama keluar menurut urutan indeks asalnya. Indeks tambahan itu berongkos memori $\Theta(n)$, sehingga pengurutannya tidak lagi di tempat.

14.9 **Kode 14.12** memuat kedua fungsi itu. `isMinHeap` adalah **Pseudokode 14.8** dengan perbandingan dibalik. `kthLargest` memelihara *max-heap* kecil berisi *calon*, pasangan (kunci, indeks), dimulai dengan akar. Fungsi ini mengambil calon terbaik sebanyak $k-1$ kali, setiap kali memasukkan anak-anak kunci itu sebagai calon baru, lalu calon terbaik yang tersisa adalah jawabannya. Cara ini benar karena induk setiap kunci lebih besar, sehingga kunci terbesar ke-j selalu merupakan anak dari salah satu $j-1$ kunci yang sudah diambil (atau akarnya). *Heap* calon memuat paling banyak k entri, sehingga setiap langkah dari k langkah berongkos $O(\lg k)$: seluruhnya $O(k \lg k)$, tidak bergantung pada n. Pada *heap* 90, 80, 70, 65, 25, 40, 30, 10, 55 fungsi ini mengembalikan 90, 80, 70, 65, 55 untuk $k = 1$ sampai 5.

Kode 14.12 `exercises.cpp`: penyelesaian Soal 14.9 sampai 14.12.

```cpp
// exercises.cpp -- solutions to the programming exercises of Chapter 14.
#include <cstdio>
#include <functional>
#include <queue>
#include <string>
#include <tuple>
#include <utility>
#include <vector>
using namespace std;

void show(const char* label, const vector<int>& a) {
    printf("%s", label);
    for (size_t i = 0; i < a.size(); ++i) printf("%s%d", i ? " " : "", a[i]);
    printf("\n");
}

// Exercise 14.9 (a): every element is at least as large as its parent.
bool isMinHeap(const vector<int>& a) {
    for (size_t i = 1; i < a.size(); ++i)
        if (a[(i - 1) / 2] > a[i]) return false;
    return true;
}

// Exercise 14.9 (b): the k-th largest key of a max-heap in O(k lg k) time.
int kthLargest(const vector<int>& heap, int k) {
    priority_queue<pair<int, int>> cand;          // (key, index), largest first
    cand.push({heap[0], 0});
    for (int step = 1; step < k; ++step) {
        int i = cand.top().second;                // the step-th largest leaves
        cand.pop();
        for (int c : {2 * i + 1, 2 * i + 2})      // its children become candidates
            if (c < static_cast<int>(heap.size())) cand.push({heap[c], c});
    }
    return cand.top().first;
}
```

Kode 14.12 (Lanjutan) *exercises.cpp: penyelesaian Soal 14.9 sampai 14.12.*

```
// Exercise 14.10: remove the key at any index of a max-heap in O(lg n).
void removeAt(vector<int>& a, int index) {
    int n = static_cast<int>(a.size()) - 1;       // size after the removal
    a[index] = a[n];                              // the last leaf fills the hole
    a.pop_back();
    if (index == n) return;                       // we removed the last leaf
    int i = index;
    while (i > 0 && a[(i - 1) / 2] < a[i]) {      // it may have to rise ...
        swap(a[i], a[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
    while (true) {                                // ... or it may have to sink
        int largest = i, l = 2 * i + 1, r = 2 * i + 2;
        if (l < n && a[l] > a[largest]) largest = l;
        if (r < n && a[r] > a[largest]) largest = r;
        if (largest == i) return;
        swap(a[i], a[largest]);
        i = largest;
    }
}

// Exercise 14.11: the median of a stream with two heaps.
void runningMedian(const vector<int>& stream) {
    priority_queue<int> low;                              // smaller half, max top
    priority_queue<int, vector<int>, greater<int>> high; // larger half, min top
    printf("median after each value:");
    for (int x : stream) {
        if (low.empty() || x <= low.top()) low.push(x); else high.push(x);
        if (low.size() > high.size() + 1) { high.push(low.top()); low.pop(); }
        else if (high.size() > low.size()) { low.push(high.top()); high.pop(); }
        double m = low.size() > high.size() ? low.top()
                                            : (low.top() + high.top()) / 2.0;
        printf(" %.1f", m);
    }
    printf("\n");
}

// Exercise 14.12: Profitina Laundry orders -- express first, then the earliest
// pickup time, then the earliest arrival.
struct Order {
    int id;
    bool express;
    int pickup;       // minutes after midnight
    int arrival;
};
```

Kode 14.12 (Lanjutan) *exercises.cpp: penyelesaian Soal 14.9 sampai 14.12.*

```
struct ServedLater {
    bool operator()(const Order& a, const Order& b) const {
        return make_tuple(!a.express, a.pickup, a.arrival) >
               make_tuple(!b.express, b.pickup, b.arrival);
    }
};

int main() {
    vector<int> h = {90, 80, 70, 65, 25, 40, 30, 10, 55};
    show("heap: ", h);
    printf("isMinHeap(heap) = %s, isMinHeap({3, 5, 4, 9, 6}) = %s\n",
           isMinHeap(h) ? "true" : "false",
           isMinHeap({3, 5, 4, 9, 6}) ? "true" : "false");
    printf("k-th largest for k = 1..5:");
    for (int k = 1; k <= 5; ++k) printf(" %d", kthLargest(h, k));
    printf("\n\n");

    vector<int> a = {90, 20, 80, 10, 15, 70, 75};
    show("heap:          ", a);
    removeAt(a, 3);
    show("removeAt(3) -> ", a);
    removeAt(a, 0);
    show("removeAt(0) -> ", a);
    printf("\n");

    runningMedian({5, 15, 1, 3, 8, 7, 9, 10});
    printf("\n");

    priority_queue<Order, vector<Order>, ServedLater> laundry;
    laundry.push({101, false, 17 * 60, 1});
    laundry.push({102, true, 15 * 60, 2});
    laundry.push({103, false, 12 * 60, 3});
    laundry.push({104, true, 11 * 60 + 30, 4});
    laundry.push({105, false, 12 * 60, 5});
    laundry.push({106, true, 15 * 60, 6});
    printf("Profitina Laundry, washing order:\n");
    while (!laundry.empty()) {
        Order o = laundry.top();
        laundry.pop();
        printf("  order %d  %-7s  pickup %02d:%02d  arrived #%d\n", o.id,
               o.express ? "express" : "regular", o.pickup / 60, o.pickup % 60,
               o.arrival);
    }
    return 0;
}
```

```
heap: 90 80 70 65 25 40 30 10 55
isMinHeap(heap) = false, isMinHeap({3, 5, 4, 9, 6}) = true
k-th largest for k = 1..5: 90 80 70 65 55

heap:          90 20 80 10 15 70 75
removeAt(3) -> 90 75 80 20 15 70
removeAt(0) -> 80 75 70 20 15

median after each value: 5.0 10.0 5.0 4.0 5.0 6.0 7.0 7.5

Profitina Laundry, washing order:
  order 104  express  pickup 11:30  arrived #4
  order 102  express  pickup 15:00  arrived #2
  order 106  express  pickup 15:00  arrived #6
  order 103  regular  pickup 12:00  arrived #3
  order 105  regular  pickup 12:00  arrived #5
  order 101  regular  pickup 17:00  arrived #1
```

14.10 `removeAt` dalam **Kode 14.12** memindahkan daun terakhir ke lubangnya, sebagaimana `extractMax` melakukannya untuk akar. Kunci yang dipindahkan berasal dari bagian lain pohon, sehingga ia mungkin terlalu kecil *atau terlalu besar* untuk tempat barunya: fungsi ini lebih dulu membiarkannya naik, lalu membiarkannya turun (paling banyak satu dari kedua gelung itu yang benar-benar memindahkan sesuatu). Pada *heap* 90, 20, 80, 10, 15, 70, 75, `removeAt(3)` menghapus 10; kunci terakhir, 75, mendarat di indeks 3 di bawah 20 dan harus *naik*, menghasilkan 90, 75, 80, 20, 15, 70. `removeAt(0)` menghapus 90; kunci terakhir, 70, mendarat di akar dan harus *turun* di bawah 80, menghasilkan 80, 75, 70, 20, 15. Kedua gelung menyusuri satu lintasan, sehingga waktunya $O(\lg n)$. Kasus khusus `index == n` (kunci yang dihapus adalah daun terakhir) tidak memerlukan perbaikan.

14.11 `runningMedian` menyimpan separuh bilangan yang lebih kecil di dalam *max-heap* `low` dan separuh yang lebih besar di dalam *min-heap* `high`, dengan `low` memuat kunci sebanyak `high` atau satu lebih banyak. Setiap bilangan baru masuk ke `low` bila ia paling besar sama dengan maksimum `low`, dan ke `high` bila tidak; lalu satu kunci dipindahkan ke seberang bila selisih ukurannya melebihi yang diizinkan. Mediannya adalah puncak `low` bila banyaknya ganjil, dan rata-rata kedua puncak bila genap. Setiap langkah berupa sejumlah konstan operasi *heap*, $O(\lg n)$. Untuk 5, 15, 1, 3, 8, 7, 9, 10 mediannya 5, 10, 5, 4, 5, 6, 7, dan 7,5: misalnya, setelah 1, 3, 5, 15 kedua paruhnya adalah {1, 3} dan {5, 15} dan mediannya $(3 + 5)/2 = 4$.

14.12 Kunci sebuah pesanan adalah *triple* (bukan ekspres, waktu pengambilan, kedatangan). *Tuple* dibandingkan dari kiri, dan `false` lebih kecil daripada `true`, sehingga pesanan ekspres (yang komponen pertamanya `false`) datang sebelum setiap pesanan reguler; lalu waktu pengambilan yang lebih awal; lalu kedatangan yang lebih awal. Di C++ perbandingan `ServedLater` mengembalikan `true` bila *triple* `a` *lebih besar*, konvensi "dilayani belakangan" dari Subbab 14.14.1; di Python *triple* itu langsung dimasukkan ke dalam *list* `heapq`, dan di Java sebuah `Comparator` dirangkai dengan `thenComparingInt`. Keluarannya mengambil pesanan ekspres 104, 102, 106 lebih dulu (11.30, lalu dua pesanan pukul 15.00 menurut urutan kedatangan) dan kemudian pesanan reguler 103, 105 (keduanya 12.00, menurut urutan kedatangan) dan 101.

14.13 Kelompokkan suku-sukunya menurut nilai $\lfloor \lg m \rfloor$. Untuk $j < L = \lfloor \lg n \rfloor$, sebanyak 2^j nilai $m = 2^j, \ldots, 2^{j+1} - 1$ masing-masing menyumbang j; nilai-nilai $m = 2^L, \ldots, n$, yang banyaknya $n - 2^L + 1$, masing-masing menyumbang L. Jadi jumlahnya adalah $\sum_{j=0}^{L-1} j\,2^j + L(n - 2^L + 1)$. Bagian pertama sama dengan $(L-2)2^L + 2$ (versi berhingga dari trik Persamaan (14.10): tuliskan $U = \sum j2^j$ lalu kurangkan dari $2U$). Dengan menjumlahkan, $(L-2)2^L + 2 + Ln - L2^L + L = (n+1)L - 2^{L+1} + 2$. Suku $m = 1$ menyumbang 0, sehingga jumlah mulai dari $m = 2$ sama saja. Pemeriksaan: $n = 1000$, $L = 9$: $9009 - 1024 + 2 = 7987$; $n = 10{,}000$, $L = 13$: $130{,}013 - 16{,}384 + 2 = 113{,}631$; $n = 100{,}000$, $L = 16$: $1{,}600{,}016 - 131{,}072 + 2 = 1{,}468{,}946$. Ketiganya sesuai dengan program.

14.14 Simpan kuncinya, pindahkan induk-induk yang lebih kecil turun ke dalam lubang, dan tulis kunci itu sekali di akhir:

Pseudokode 14.12 `SIFT-UP-HOLE`: kunci naik melalui lubang yang berpindah dan ditulis sekali di akhir.

```
SIFT-UP-HOLE(A, i)
 1  key = A[i]
 2  while i > 0 and A[parent(i)] < key
 3      A[i] = A[parent(i)]          // the parent moves down into the hole
 4      i = parent(i)
 5  A[i] = key
```

Bila kunci naik s level, **Pseudokode 14.1** melakukan s pertukaran, yaitu $2s$ operasi tulis (tiga penugasan per pertukaran lewat variabel sementara, dua di antaranya ke larik), sedangkan **Pseudokode 14.12** melakukan $s + 1$ operasi tulis. Banyak perbandingannya sama. Teknik yang sama berlaku untuk *sift-down*.

14.15 (a) Algoritma Dijkstra memfinalkan simpul-simpul dalam urutan jarak yang tidak menurun, dan jarak simpul yang sudah final tidak berubah lagi (argumen kebenaran Subbab 13.10.5, yang memerlukan bobot tak negatif [4]). Entri (d, u) yang diambil setelah u final membawa jarak paling sedikit sebesar jarak terpendek sebenarnya, sehingga tidak dapat memperbaiki apa pun dan boleh dibuang. (b) Setiap relaksasi yang berhasil memasukkan satu entri, dan setiap sisi berarah direlaksasi paling banyak sekali, yaitu ketika pangkalnya menjadi final; ditambah entri awal, antrean memuat paling banyak $|E| + 1$ entri (untuk graf tak berarah, yang setiap jalannya muncul di kedua daftar ketetanggaan, paling banyak $2|E| + 1$). Pada **Kode 14.11**, 9 jalan menghasilkan 10 penyisipan. (c) *Max-heap* akan memfinalkan simpul dengan jarak sementara *terbesar* lebih dulu; jarak-jarak akan dinyatakan final sebelum rute yang lebih pendek sempat diperiksa, dan hasilnya salah. Pada **Gambar 14.13**, Engineering akan difinalkan pada 7 sebelum rute lewat Science (5) ditemukan.

Daftar Pustaka

[1] J. W. J. Williams, “Algorithm 232: Heapsort,” *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, Jun. 1964.

[2] R. W. Floyd, “Algorithm 245: Treesort 3,” *Communications of the ACM*, vol. 7, no. 12, p. 701, Dec. 1964.

[3] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998, sec. 5.2.3.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 6.

[5] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011, sec. 2.4.

[6] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte, “Min-max heaps and generalized priority queues,” *Communications of the ACM*, vol. 29, no. 10, pp. 996–1000, Oct. 1986.

[7] T. Porter and I. Simon, “Random insertion into a priority queue structure,” *IEEE Transactions on Software Engineering*, vol. SE-1, no. 3, pp. 292–298, Sep. 1975.

[8] D. R. Musser, “Introspective sorting and selection algorithms,” *Software: Practice and Experience*, vol. 27, no. 8, pp. 983–993, Aug. 1997.

[9] cppreference.com, “std::priority_queue.” [Online]. Available: https://en.cppreference.com/w/cpp/container/priority_queue. [Accessed: 24-Sep-2026].

[10] cppreference.com, “std::make_heap,” “std::push_heap,” “std::pop_heap,” “std::sort_heap,” “std::is_heap.” [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/make_heap. [Accessed: 24-Sep-2026].

[11] Oracle, “Class PriorityQueue,” *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/PriorityQueue.html. [Accessed: 24-Sep-2026].

[12] Python Software Foundation, “heapq — Heap queue algorithm,” *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/heapq.html. [Accessed: 24-Sep-2026].

[13] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[14] M. L. Fredman and R. E. Tarjan, “Fibonacci heaps and their uses in improved network optimization algorithms,” *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[15] Profitina, “Profitina — AI Visibility (GEO) platform.” [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

Bab 15

Hashing dan Tabel Hash

Di akhir bab ini Anda akan sanggup menjelaskan mengapa sebuah tabel hash dapat menemukan sebuah kunci dalam banyak langkah yang konstan secara rata-rata, sedangkan setiap pencarian berbasis perbandingan memerlukan sedikitnya banyak langkah yang logaritmik; merancang dan menilai fungsi hash untuk bilangan bulat dan *string* (metode pembagian, metode perkalian, hash polinomial yang dihitung dengan aturan Horner beserta pembaruan bergulirnya), serta mengukur sifat *avalanche* yang membedakan pengaduk yang baik dari yang buruk; menghitung seberapa cepat kunci-kunci acak bertabrakan dengan batas ulang tahun (*birthday bound*); membangun tabel hash dengan *separate chaining* (perantaian terpisah) dari simpul-simpul berantai Bab 7, dan tabel lain dengan pengalamatan terbuka (*open addressing*) melalui *linear probing*, *quadratic probing*, dan *double hashing*, termasuk penghapusan dengan nisan (*tombstone*); meramalkan dan mengukur bagaimana faktor muat (*load factor*) menentukan banyaknya *probe*, dan mengapa *rehashing* dengan penggandaan hanya berongkos konstan per penyisipan; mengenali kasus terburuk, serangan *hash flooding* yang memancingnya, dan hashing teracak yang menangkalnya; memilih dengan tepat antara `std::map` dan `std::unordered_map`, serta menjelaskan apa yang dikerjakan `HashMap` milik Java dan `dict` milik Python di dalamnya; dan memakai tabel hash untuk menghitung kata, mendeteksi duplikat, menyelesaikan *two-sum*, dan membangun *cache*. Setiap tabel digambar, setiap rumus diturunkan lalu dicocokkan dengan pengukuran, dan setiap program lengkap, dikompilasi atau dijalankan dalam C++, Java, dan Python, serta diperiksa memorinya.

15.1 Dari Mana Kita Melanjutkan

Buku ini terus-menerus menjawab satu pertanyaan: *diberikan sebuah kunci, apakah kunci itu ada di sini, dan bila ada, di mana?* Bab 2 menjawabnya dengan pencarian sekuensial, yang memeriksa elemen satu per satu dan memerlukan hingga n *probe* untuk n elemen, dan dengan pencarian biner, yang memerlukan sekitar lg$n$ *probe* tetapi hanya berlaku pada data terurut. Subbab 2.12.2 lalu membuktikan sesuatu yang lebih kuat dan sedikit mengecewakan: tidak ada algoritma yang mengenal data hanya dengan *membandingkan* kunci yang dapat mengalahkan $\lceil \lg(n+1) \rceil$ *probe* pada kasus terburuk. Subbab yang sama menyisakan sebuah celah. Pembuktian itu hanya berlaku bagi algoritma yang membandingkan. Algoritma yang melakukan *aritmetika* pada kunci itu sendiri tidak terikat olehnya.

Bab 11 menjumpai batas yang sama dari sisi lain. *Binary search tree* menjaga kunci-kuncinya tetap terurut dan menemukan sebuah kunci dalam $O(h)$ langkah, dengan h tinggi pohon; pohon seimbang pada Subbab 11.9 menjaga h tetap dalam $O(\lg n)$. Itu luar biasa, dan disertai bonus yang tidak diberikan larik semurah itu: penyisipan dan penghapusan juga berongkos $O(\lg n)$. Namun tetap saja itu logaritma, karena pohon pencarian adalah mesin pembanding.

Bab ini memanfaatkan celah tersebut. Sebuah *tabel hash* menghitung, langsung dari kunci, posisi tempat kunci itu seharusnya disimpan, lalu pergi ke sana. Bila perhitungan itu menyebarkan kunci dengan baik, posisinya biasanya tepat pada percobaan pertama, dan sebuah pencarian rata-rata memerlukan banyak langkah yang konstan, berapa pun banyaknya kunci yang tersimpan. Kita sebenarnya sudah dua kali bertemu gagasan ini tanpa namanya. Di Subbab 4.4.3 sebuah larik

berisi 256 pencacah, yang diindeks oleh karakter itu sendiri, menemukan karakter berulang pertama dalam satu lintasan, karena "nilai yang Anda cari memberi tahu di mana harus mencari". Di Subbab 8.7.3 sebuah himpunan berisi nilai-nilai yang sudah terlihat membuang duplikat dari sebuah list dalam waktu linear. Di Subbab 9.9 sebuah *hash map* menunjuk dari setiap judul ke simpulnya di list "baru dipinjam". Ketiganya bersandar pada struktur yang belum dibangun buku ini. Sekarang kita membangunnya, dua kali, mengukurnya, merusaknya dengan sengaja, lalu membaca apa yang sesungguhnya dilakukan pustaka baku C++, Java, dan Python.

Harganya harus disebut sejak awal, karena harga itu membentuk setiap keputusan berikutnya dalam bab ini. Tabel hash melepaskan *urutan*. Ia tidak dapat mendaftar kuncinya dari yang terkecil hingga terbesar tanpa mengurutkannya lebih dulu, tidak dapat menjawab "kode mana saja yang terletak antara B1003 dan B1006?" tanpa memeriksa setiap kunci, dan tidak dapat menemukan kunci berikutnya sesudah kunci tertentu. Ia juga melepaskan *jaminan*: waktu konstan itu adalah rata-rata, dan sekumpulan kunci yang sial atau jahat dapat membuat setiap operasi menjadi linear. Subbab 15.8 memperlihatkan seberapa buruk hal itu dan bagaimana sistem nyata membela diri. Bab 16 membahas soal-soal persis semacam ini, memadukan graf dari Bab 13, *heap* dari Bab 14, dan tabel hash dari bab ini.

15.2 Dari $O(n)$ ke $O(\lg n)$ ke $O(1)$: Gagasan Hashing

15.2.1 Pustakawan yang tidak pernah mencari

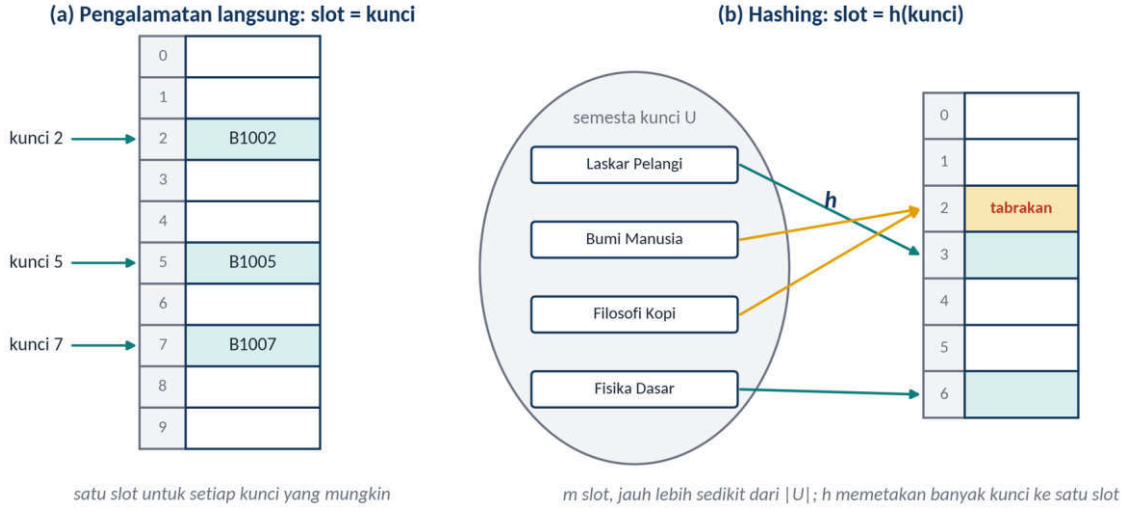
Bayangkan meja pengembalian di Pustaka Ceria pada jam tutup. Seorang pembaca bertanya apakah *Filosofi Kopi* sudah kembali hari ini. Relawan baru memeriksa tumpukan buku yang dikembalikan satu per satu; dengan empat puluh buku di tumpukan, itu bisa memakan empat puluh kali lihat. Mahasiswa yang sudah setahun menjaga meja menyimpan buku-buku yang kembali terurut menurut judul di sebuah rak dan membuka rak itu di tengah, lalu di tengah separuh yang benar, dan seterusnya; empat puluh buku memerlukan paling banyak enam kali lihat. Kepala pustakawan melakukan sesuatu yang lain. Di bawah meja berdiri dua puluh enam baki berlabel A sampai Z. Setiap buku yang kembali masuk ke baki huruf pertama judulnya. Ditanya tentang *Filosofi Kopi*, ia sama sekali tidak mencari: ia merogoh baki F, tempat dua atau tiga buku tergeletak, dan langsung menjawab.

Ketiga orang itu adalah tiga baris dari perjalanan buku ini. Relawan adalah pencarian sekuensial, $O(n)$. Mahasiswa adalah pencarian biner, $O(\lg n)$. Kepala pustakawan adalah hashing: ia *menghitung* di mana buku itu mesti berada, dari judul buku itu sendiri, dan hanya melihat ke sana. Aturannya, "huruf pertama judul", adalah sebuah *fungsi hash*, baki-baki itu adalah *bucket*, dan kenyataan bahwa *Fisika Dasar* dan *Filosofi Kopi* berbagi baki F adalah sebuah *tabrakan* (*collision*). Metodenya tetap cepat hanya selama tidak ada baki yang membesar: bila perpustakaan menerima seribu buku yang judulnya semua diawali "Sejarah", baki S akan menjadi tumpukan yang harus dicari, dan ia kembali menjadi si relawan. Segala sesuatu dalam bab ini adalah penyempurnaan baki-baki itu: aturan yang lebih baik daripada huruf pertama, banyak baki yang masuk akal, rencana untuk buku yang jatuh ke baki yang sama, dan cara menambah baki ketika tumpukan membesar.

15.2.2 Pengalamatan langsung

Versi baki yang paling sederhana tidak memerlukan aturan sama sekali. Misalkan setiap kunci adalah bilangan bulat tak negatif yang kecil, misalnya nomor anggota 0 sampai 999 para pembaca perpustakaan. Simpan sebuah larik `T` berisi 1000 sel dan letakkan rekaman anggota k di `T[k]`.

Pencarian, penyisipan, dan penghapusan masing-masing menjadi satu akses larik, $O(1)$ pada kasus terburuk, tanpa satu pun perbandingan kunci. Inilah *pengalamatan langsung* (*direct addressing*), dan **Gambar 15.1**(a) menggambarkannya. Ke-256 pencacah Subbab 4.4.3 tepat merupakan tabel beralamat langsung yang kuncinya adalah 256 kemungkinan nilai *byte*.



Gambar 15.1 (a) Tabel beralamat langsung memiliki satu slot untuk setiap kunci yang mungkin, sehingga kunci itu sendiri menjadi indeks. (b) Tabel hash memiliki slot jauh lebih sedikit daripada kunci yang mungkin; fungsi hash h memetakan setiap kunci ke sebuah slot, dan dua kunci berbeda dapat dikirim ke slot yang sama, yaitu sebuah tabrakan.

Pengalamatan langsung gagal begitu himpunan U dari kunci yang *mungkin*, yaitu *semesta*, jauh lebih besar daripada himpunan kunci yang benar-benar disimpan. Semesta judul buku adalah setiap *string* huruf, yang untuk keperluan praktis tak terhingga. Semesta bilangan bulat 64-bit memiliki $2^{64} \approx 1.8 \times 10^{19}$ anggota; larik dengan satu *byte* per kunci yang mungkin akan memerlukan 18 *exabyte*. Bahkan semesta sederhana nomor induk kependudukan Indonesia (enam belas digit) akan memerlukan 10^{16} sel untuk menampung barangkali tiga ratus juta orang. Larik itu hampir seluruhnya kosong.

15.2.3 Hashing: kosakatanya

Hashing mempertahankan akses larik yang cepat dari pengalamatan langsung dan membuang lariknya yang raksasa. Kita memilih tabel berisi m slot, dengan m sebanding dengan banyaknya kunci yang kita perkirakan akan disimpan, dan sebuah fungsi yang memampatkan semesta ke dalamnya.

> **DEFINISI.** *Fungsi hash* untuk tabel berisi m slot adalah fungsi $h: U \to \{0,1,\ldots,m-1\}$. Bilangan $h(k)$ adalah *nilai hash* (atau *slot asal*) kunci k, dan kita katakan k *di-hash ke* slot $h(k)$. *Tabel hash* adalah larik $T[0..m-1]$ tempat kunci k disimpan pada, atau mulai dari, slot $h(k)$. Dua kunci berbeda $k_1 \neq k_2$ dengan $h(k_1) = h(k_2)$ *bertabrakan*. *Faktor muat* (*load factor*) tabel yang menyimpan n kunci dalam m slot adalah $\alpha = n/m$.

Sebuah slot juga disebut *bucket*, terutama bila slot itu dapat menampung beberapa kunci. **Gambar 15.1**(b) memperlihatkan empat judul yang di-hash ke tujuh slot: *Bumi Manusia* dan *Filosofi Kopi* bertabrakan di slot 2. Sisa bab ini menjawab tiga pertanyaan yang dibiarkan terbuka oleh definisi tersebut. Seperti apa h yang baik (Subbab 15.3)? Seberapa sering kunci bertabrakan (Subbab 15.4)? Dan apa yang kita lakukan ketika tabrakan terjadi (Subbab 15.5 dan 15.6)?

Gagasan ini hampir setua penyimpanan akses acak itu sendiri. Knuth menelusurinya ke sebuah memorandum internal IBM bulan Januari 1953 karya Hans Peter Luhn, yang sudah mengusulkan

perantaian rekaman-rekaman yang bertabrakan, dan ke *assembler* IBM 701 tahun 1954 karya Gene Amdahl, Elaine McGraw (kemudian Boehme), Nathaniel Rochester, dan Arthur Samuel, yang memakai apa yang kini kita sebut *linear probing* [1]. Uraian terbitan pertama, lengkap dengan analisisnya untuk berkas di cakram, adalah makalah W. Wesley Peterson tahun 1957 tentang "pengalamatan untuk penyimpanan akses acak" [2], dan kata *hashing*, dari makna sehari-hari mencincang sesuatu menjadi potongan kecil lalu mengaduknya, pertama kali muncul di media cetak dalam survei Robert Morris tahun 1968 [1]. Tujuh puluh tahun kemudian tabel hash duduk di dalam setiap *compiler* (tabel pengenal), setiap basis data (*hash join* dan indeks hash), setiap peramban web (*cache* nama *host* yang baru dilihat, yang diuraikan Bab 1), dan di inti bahasa Python, yang objek, modul, dan kelasnya semuanya adalah *dictionary*.

15.2.4 Perjalanan itu dalam satu tabel

Tabel 15.1 menempatkan hashing di samping struktur pencarian dari bab-bab sebelumnya. Kolom yang paling menentukan pilihan adalah kolom terakhir: apa yang dituntut setiap struktur selain waktu.

Tabel 15.1 Mencari sebuah kunci: dari pencarian sekuensial hingga hashing.

Struktur	Pencarian	Penyisipan	Penghapusan	Menjaga urutan?	Yang diperlukan
Larik tak terurut (Subbab 2.3)	$\Theta(n)$	$O(1)$ di ujung	$\Theta(n)$	tidak	tidak ada
Larik terurut (Subbab 2.5)	$\Theta(\lg n)$	$\Theta(n)$	$\Theta(n)$	ya	kunci yang dapat dibandingkan
Pohon pencarian seimbang (Subbab 11.9)	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	ya	kunci yang dapat dibandingkan
Tabel hash (bab ini)	$O(1)$ rata-rata, $\Theta(n)$ terburuk	$O(1)$ rata-rata	$O(1)$ rata-rata	tidak	fungsi hash yang baik dan kesamaan

DENGAN BAHASA SEHARI-HARI. Mencari dengan membandingkan itu seperti bermain tebak "lebih besar atau lebih kecil": setiap jawaban membelah dua sisa kemungkinan, sehingga Anda memerlukan sekitar $\lg n$ pertanyaan. Hashing itu seperti mengetahui nomor rumah: Anda berjalan lurus ke pintunya. Anda hanya mengetuk beberapa pintu yang salah bila dua orang diberi nomor yang sama.

15.3 Fungsi Hash

15.3.1 Apa yang harus dilakukan fungsi hash yang baik

Fungsi hash memiliki satu tugas: mengubah kunci menjadi nomor slot sedemikian rupa sehingga kunci-kunci yang benar-benar kita simpan tersebar merata ke m slot. Empat sifat mengalir dari tugas itu.

1. **Deterministik.** Kunci yang sama harus selalu menghasilkan slot yang sama, selama umur satu tabel. Fungsi yang berkonsultasi dengan jam akan menyimpan kunci di satu slot dan mencarinya di slot lain.
2. **Cepat.** Intinya adalah mengalahkan $\lg n$ perbandingan; fungsi hash yang lebih mahal daripada pencarian biner telah menggagalkan tujuannya sendiri. Fungsi hash yang baik berongkos beberapa instruksi mesin per *byte* kunci.

3. **Seragam.** Setiap slot hendaknya menerima bagian yang kira-kira sama dari kunci-kunci yang *sungguh* disimpan, yang jarang acak: kode buku berurutan, harga berupa ribuan bulat, pengenal berbagi awalan.
4. **Memakai seluruh kunci.** Setiap bagian kunci hendaknya memengaruhi slot. Aturan "huruf pertama" sang kepala pustakawan mengabaikan segala sesuatu sesudah huruf pertama, dan justru karena itulah semua buku *Sejarah* menumpuk di baki S.

Sifat ketiga biasanya dinyatakan sebagai asumsi ideal, tolok ukur bagi fungsi-fungsi nyata, dan di bawah asumsi itulah semua nilai rata-rata dalam bab ini diturunkan.

> **DEFINISI.** Di bawah asumsi *simple uniform hashing* (hashing seragam sederhana), setiap kunci berpeluang sama untuk di-hash ke salah satu dari m slot, bebas dari ke mana kunci-kunci lain di-hash:

$$\Pr[h(k) = j] = \frac{1}{m} \quad \text{untuk setiap slot } j \in \{0,1,\dots,m-1\}. \tag{15.1}$$

Tidak ada fungsi tetap yang dapat memenuhi Persamaan (15.1) untuk setiap himpunan kunci yang mungkin, karena lawan yang mengetahui h selalu dapat memilih kunci-kunci yang bertabrakan (Subbab 15.8 melakukan persis itu). Asumsi ini tetap merupakan sasaran rancangan yang tepat: fungsi yang baik berperilaku *seolah-olah* Persamaan (15.1) berlaku untuk kunci-kunci yang disimpan program nyata, dan Subbab 15.7 memeriksa bahwa pengukuran kita cocok dengan nilai rata-rata yang diramalkannya.

15.3.2 Metode pembagian

Fungsi hash tertua dan paling sederhana untuk kunci bilangan bulat mengambil sisa pembagian oleh ukuran tabel:

$$h(k) = k \bmod m. \tag{15.2}$$

Ongkosnya satu pembagian. Untuk $m = 11$ dan kunci 45000 hasilnya $45000 = 4090 \cdot 11 + 10$, jadi $h(45000) = 10$. Seluruh kesulitannya terletak pada pemilihan m, karena sisa pembagian hanya bergantung pada sebagian informasi kunci, dan bagian mana yang dipakai bergantung pada m.

Misalkan Pustaka Ceria meng-hash harga delapan buku, semuanya ribuan rupiah bulat: 45000, 60000, 75000, 85000, 90000, 120000, 55000, 70000. Bagian pertama keluaran **Kode 15.1**, yang diperlihatkan sedikit kemudian dalam subbab ini, meng-hash-nya dengan tiga ukuran tabel:

```
m = 10:  0  0  0  0  0  0  0  0
m = 16:  8  0  8  8  0  0  8  0
m = 11: 10  6  2  3  9  1  0  7
```

Dengan $m = 10$ setiap harga jatuh di slot 0. Sisa modulo 10 adalah digit desimal terakhir, dan setiap harga berakhir dengan 0; fungsi itu hanya melihat bagian kunci yang tidak pernah berubah. Dengan $m = 16 = 2^4$ sisanya adalah empat digit *biner* terakhir, dan kelipatan 1000 $= 2^3 \cdot 125$ selalu berakhir dengan tiga nol biner, sehingga hanya slot 0 dan 8 yang mungkin. Dengan bilangan prima $m = 11$ kedelapan harga jatuh di slot yang berbeda-beda. Aturan umumnya mengalir dari contoh ini.

- **Hindari $m = 2^p$ pada metode pembagian.** Maka $k \bmod m$ hanyalah p bit terendah k, dan kunci-kunci yang hanya berbeda pada bit-bit yang lebih tinggi semuanya bertabrakan.
- **Hindari** $m = 10^p$ untuk data desimal, dengan alasan yang sama.

- **Utamakan m prima yang tidak terlalu dekat dengan pangkat 2.** Bilangan prima tidak berbagi faktor dengan langkah-langkah teratur yang dimiliki kunci nyata (kelipatan 1000, 8, 12), sehingga setiap langkah menjelajahi semua slot sebelum berulang.

Butir terakhir layak dibuktikan dalam satu baris. Jika kunci-kuncinya $k_0, k_0 + d, k_0 + 2d, \ldots$ dengan beda bersama d, slotnya adalah $k_0 + id \bmod m$. Dua di antaranya, i dan j dengan $0 \le i < j < m$, sama tepat ketika m membagi $(j - i)d$. Jika m prima dan tidak membagi d, bilangan prima itu harus membagi $j - i$, yang mustahil untuk $0 < j - i < m$. Jadi m kunci pertama dari langkah itu mengisi seluruh m slot. Sebaliknya jika $g = \gcd(d, m) > 1$, langkah itu hanya pernah mencapai m/g slot, sebagaimana kelipatan 1000 dalam tabel berukuran 16 hanya mencapai $16/8 = 2$ di antaranya.

15.3.3 Metode perkalian

Metode pembagian memaksa kita memilih m dengan hati-hati. *Metode perkalian* berlaku untuk m apa pun, khususnya untuk pangkat 2, yang praktis karena tabel dapat menggandakan ukurannya. Pilih konstanta A dengan $0 < A < 1$, kalikan kunci dengannya, ambil bagian pecahannya, lalu skalakan ke ukuran tabel:

$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor, \tag{15.3}$$

dengan $kA \bmod 1 = kA - \lfloor kA \rfloor$ bagian pecahan dari kA. Knuth menganjurkan konstanta rasio emas $A = (\sqrt{5} - 1)/2 \approx 0.6180339887$ [1], yang menyebarkan kunci-kunci berurutan dengan sangat merata: bagian pecahan dari $A, 2A, 3A, \ldots$ selalu jatuh ke celah terbesar yang tersisa pada selang $[0,1)$.

Untuk kunci $k = 45000$ dan $m = 16$: $kA = 45000 \times 0.6180339887\ldots = 27811.529\ldots$, bagian pecahannya $0.529\ldots$, dan $\lfloor 16 \times 0.529\ldots \rfloor = \lfloor 8.47\ldots \rfloor = 8$.

Pada mesin sungguhan kita menghindari *floating point*. Dengan *word* berukuran w bit dan $m = 2^p$, misalkan $s = \lfloor A \cdot 2^w \rfloor$. Hasil kali $k \cdot s$ yang dihitung dalam aritmetika tak bertanda w bit (yang diam-diam membuang segala sesuatu di atas bit $w - 1$) adalah bagian pecahan kA yang diskalakan dengan 2^w, dan p bit teratasnya adalah slotnya:

$$h(k) = (k \cdot s \bmod 2^w) \gg (w - p), \qquad s = \lfloor 2^w A \rfloor. \tag{15.4}$$

Untuk $w = 64$, $s = 0\text{x}9\text{E}3779\text{B}97\text{F}4\text{A}7\text{C}15$, dan metode ini sering disebut *Fibonacci hashing*, karena A adalah limit rasio dua bilangan Fibonacci yang berurutan. Persamaan (15.4) berupa satu perkalian dan satu geseran, lebih cepat daripada pembagian, dan itulah yang dihitung fungsi `hashMultInt` pada **Kode 15.1**. Delapan harga yang sama, yang tadi runtuh ke dua slot di bawah k mod 16, kini tersebar ke enam dari enam belas slot, dan bentuk *floating point* maupun bentuk bilangan bulatnya sepakat pada setiap harga (bagian 2 keluaran).

15.3.4 Meng-hash *string*

Sebagian besar kunci nyata adalah *string*: judul, nama, kode, kata, URL. Fungsi hash *string* harus lebih dulu mengubah karakter-karakternya menjadi satu bilangan bulat. Gagasan yang paling jelas, menjumlahkan kode-kode karakter, adalah jebakan:

$$h_{\text{sum}}(s) = \sum_{i=0}^{L-1} s_i,$$

dengan $s_0, s_1, \ldots, s_{L-1}$ kode dari L karakter. Penjumlahan tidak peduli urutan, sehingga setiap *anagram* bertabrakan. Dalam bahasa Indonesia *kasur* dan *rusak* sama-sama berjumlah 550, dan *kita* serta *kiat* sama-sama 425. Lebih buruk lagi, jumlah dari *string* pendek adalah bilangan kecil yang berjejal dalam rentang sempit: 2000 kode buku B0000 hingga B1999 memiliki jumlah antara 258 saja (untuk B0000) dan 286 (untuk B1999), sehingga dalam tabel berisi 101 *bucket* paling banyak 29 *bucket* yang pernah dapat dipakai; keluaran di bawah memperlihatkan tepat 72 dari 101 *bucket* kosong dan salah satunya menampung 150 kode.

Obatnya adalah membuat *posisi* setiap karakter berpengaruh, dengan memperlakukan *string* sebagai bilangan yang ditulis dalam basis B:

$$h(s) = s_0 B^{L-1} + s_1 B^{L-2} + \cdots + s_{L-2} B + s_{L-1} = \sum_{i=0}^{L-1} s_i \, B^{L-1-i}. \tag{15.5}$$

Inilah *hash polinomial*. `String.hashCode` milik Java memakai persis Persamaan (15.5) dengan $B = 31$ dan aritmetika 32-bit [3]; program-program kita memakai $B = 31$ dengan aritmetika 64-bit. Basis ganjil yang kecil bekerja baik karena perkalian dengannya mencampurkan setiap karakter baru ke semua bit di atasnya, dan $31 = 2^5 - 1$ juga murah (satu geseran dan satu pengurangan). Kini urutan berpengaruh: *kasur* di-hash menjadi 101820730 dan *rusak* menjadi 108880570.

Menghitung Persamaan (15.5) pangkat demi pangkat akan memboroskan perkalian. *Aturan Horner*, cara baku menghitung nilai polinomial dengan satu perkalian per koefisien, memfaktorkannya:

$$h(s) = (\cdots((s_0 \cdot B + s_1) \cdot B + s_2) \cdot B + \cdots) \cdot B + s_{L-1},$$

satu perkalian dan satu penjumlahan per karakter, seperti diperlihatkan **Pseudokode 15.1**.

Pseudokode 15.1 Hash polinomial sebuah *string* dengan aturan Horner; M adalah 2^{64} bila luapan tak bertanda mesin mengerjakan reduksinya secara cuma-cuma.

```
POLY-HASH(s, L, B, M)
1  h = 0
2  for i = 0 to L - 1
3      h = (h * B + s[i]) mod M      // geser digit lama ke atas, tambahkan yang baru
4  return h
```

Telusurilah pada kode “B1001”, yang karakter-karakternya berkode 66, 49, 48, 48, 49:

```
h = 0      * 31 + 66 = 66
h = 66     * 31 + 49 = 2095
h = 2095   * 31 + 48 = 64993
h = 64993  * 31 + 48 = 2014831
h = 2014831 * 31 + 49 = 62459810
```

Nilai akhirnya, 62459810, juga persis yang dikembalikan `"B1001".hashCode()` milik Java (Subbab 15.10.1 mencetaknya), karena untuk *string* sependek ini belum terjadi reduksi modulo 2^{32}. Reduksi modulo M pada baris 3 **Pseudokode 15.1** menjaga bilangan-bilangan tetap muat dalam satu *word* mesin; dengan $M = 2^{64}$ dan aritmetika tak bertanda di C++ reduksi itu gratis, karena perangkat keras membuang bit-bit luapan dengan sendirinya. Di Java tipe `long` membungkus dengan cara yang sama; di Python, yang bilangan bulatnya tidak pernah meluap, kita menulis `& MASK` dengan MASK $= 2^{64} - 1$ untuk memperoleh bit-bit yang sama. Indeks tabelnya kemudian $h(s)$ mod m seperti pada Persamaan (15.2).

15.3.5 Pembaruan bergulir

Hash polinomial memiliki bakat kedua. Misalkan kita memerlukan hash dari *setiap* jendela berisi W karakter berurutan pada sebuah teks panjang, misalnya untuk menemukan di mana sebuah frasa muncul. Menghitung ulang setiap jendela dari awal berongkos W langkah per jendela. Namun jendela-jendela yang bertetangga berbagi $W-1$ karakter, dan Persamaan (15.5) memungkinkan kita berpindah dari jendela yang dimulai di $i-1$ ke jendela yang dimulai di i dalam waktu konstan: buang sumbangan karakter yang keluar di kiri, geser semuanya satu tempat ke atas, dan tambahkan karakter yang masuk di kanan.

$$h_i = (h_{i-1} - s_{i-1}B^{W-1}) \cdot B + s_{i+W-1}. \tag{15.6}$$

Rolling hash (hash bergulir) ini adalah jantung algoritma pencocokan *string* Rabin-Karp dan "pemotongan berdasarkan isi" (*content-defined chunking*) yang dipakai program cadangan untuk menemukan blok berulang dalam berkas. Bagian 4 **Kode 15.1** menggerakkan jendela empat huruf sepanjang "pustakaceria" dengan Persamaan (15.6) dan mencocokkan setiap hasil dengan perhitungan baru: kesembilan jendela sepakat. Pengurangannya dapat turun di bawah nol di tengah perhitungan; dengan aritmetika tak bertanda modulo 2^{64} hal itu tidak berbahaya, karena hasil akhirnya tetap benar modulo 2^{64}.

15.3.6 *Avalanche*: seberapa baik sebuah fungsi mengaduk?

Keseragaman menyangkut himpunan kunci. Ada uji yang lebih tajam yang memandang bit tunggal. Sebuah fungsi hash memiliki *sifat avalanche* (sifat longsoran) bila membalik satu bit masukan mana pun membalik setiap bit keluaran dengan peluang $1/2$, sehingga rata-rata separuh bit keluaran berubah. Dengan keluaran 64-bit, idealnya 32 bit terbalik untuk setiap bit masukan. Mengapa peduli? Karena tabel hanya mengambil beberapa bit dari hash: tabel berisi 2^{10} slot memakai sepuluh. Jika sebagian bit masukan tidak dapat menjangkau kesepuluh bit itu, kunci-kunci yang hanya berbeda pada bit-bit masukan tersebut selalu bertabrakan.

Bagian 5 **Kode 15.1** mengukur hal ini untuk empat fungsi bilangan bulat. Untuk 2000 kunci acak 64-bit, program itu membalik setiap bit dari 64 bit masukan bergantian dan menghitung berapa bit keluaran yang berubah. Program melaporkan rata-rata atas semua kunci dan bit, serta bit masukan *terlemah*, yaitu bit yang pembalikannya rata-rata mengubah paling sedikit bit keluaran:

```
function    average  weakest input   bit
identity       1.00           1.00     0
k*31+7         3.67           1.00    63
Fibonacci     16.02           1.00    63
SplitMix64    31.97          31.80    25
```

Fungsi identitas, $h(k) = k$, adalah apa yang sesungguhnya dihitung `std::hash<int>` milik GCC (Subbab 15.9 mencetak `std::hash<int>{}(42) = 42`): membalik satu bit membalik tepat bit itu dan tidak ada yang lain. Perkalian dengan 31 hanya membawa setiap perubahan ke atas, sehingga bit teratas, bit 63, tetap tidak memengaruhi apa pun selain dirinya sendiri. Perkalian Fibonacci jauh lebih baik secara rata-rata, tetapi memiliki titik buta yang sama: perubahan pada bit masukan yang tinggi hanya dapat menggerakkan bit-bit di atasnya. Karena itulah Persamaan (15.4) mengambil bit-bit *teratas* hasil kali, tidak pernah yang terbawah. *Finalizer SplitMix64*, tiga geseran-xor yang diselingi dua perkalian dengan konstanta ganjil, hampir sempurna: setiap bit masukan mengubah sekitar 32 bit keluaran. *Pengaduk* (*mixer*) semacam ini murah (sekitar lima instruksi) dan mengubah bahkan fungsi identitas menjadi hash yang baik, sehingga banyak pustaka tabel hash modern menutup setiap nilai hash dengan langkah pengadukan semacam ini.

15.3.7 Programnya

Kode 15.1 memuat semua fungsi subbab ini dan keenam percobaan yang dikutip di dalamnya. Bacalah `hashPoly` di samping **Pseudokode 15.1**, dan gelung bergulir di samping Persamaan (15.6).

Kode 15.1 `hash_functions.cpp`: metode pembagian dan perkalian, hash *string* penjumlahan dan polinomial, pembaruan bergulir, serta pengukuran *avalanche*.

```
// hash_functions.cpp -- the division, multiplication and string hash functions
// of Section 15.3, the rolling update, and a measurement of the avalanche effect.
#include <cmath>
#include <cstdint>
#include <cstdio>
#include <string>
#include <vector>

typedef std::uint64_t u64;

// Division method: h(k) = k mod m.
u64 hashDivision(u64 k, u64 m) { return k % m; }

// Multiplication method, textbook form: h(k) = floor(m * frac(k * A)).
u64 hashMultFloat(u64 k, u64 m) {
    const double A = (std::sqrt(5.0) - 1.0) / 2.0;   // 0.6180339887...
    double prod = (double)k * A;
    return (u64)std::floor((double)m * (prod - std::floor(prod)));
}

// Multiplication method, integer form for m = 2^p ("Fibonacci hashing").
u64 hashMultInt(u64 k, int p) {
    const u64 s = 0x9E3779B97F4A7C15ULL;             // floor(2^64 * A)
    return (k * s) >> (64 - p);                        // the top p bits of k*s
}

// A poor string hash: add up the character codes.
u64 hashSum(const std::string &s) {
    u64 h = 0;
    for (unsigned char c : s) h += c;
    return h;
}

// Polynomial hash by Horner's rule, base 31, arithmetic modulo 2^64.
u64 hashPoly(const std::string &s) {
    u64 h = 0;
    for (unsigned char c : s) h = h * 31 + c;
    return h;
}
```

Kode 15.1 (Lanjutan) *hash_functions.cpp: metode pembagian dan perkalian, hash string penjumlahan dan polinomial, pembaruan bergulir, serta pengukuran avalanche.*

```
// Four integer-to-integer functions whose avalanche we measure.
u64 fIdentity(u64 k) { return k; }
u64 fTimes31(u64 k) { return k * 31 + 7; }
u64 fFibonacci(u64 k) { return k * 0x9E3779B97F4A7C15ULL; }
u64 fMix(u64 k) {                                       // the SplitMix64 finalizer
    k = (k ^ (k >> 30)) * 0xBF58476D1CE4E5B9ULL;
    k = (k ^ (k >> 27)) * 0x94D049BB133111EBULL;
    return k ^ (k >> 31);
}

u64 rngState = 2026;                                    // a fixed seed: same run
u64 nextRandom() {                                      // every time (SplitMix64)
    rngState += 0x9E3779B97F4A7C15ULL;
    return fMix(rngState);
}

int popcount64(u64 x) {
    int c = 0;
    while (x != 0) { x &= x - 1; ++c; }
    return c;
}

void avalanche(const char *name, u64 (*f)(u64)) {
    const int KEYS = 2000;
    long long total = 0;
    int worstBit = 0;
    double worstAvg = 64.0;
    std::vector<long long> perBit(64, 0);
    rngState = 2026;
    for (int t = 0; t < KEYS; ++t) {
        u64 k = nextRandom(), hk = f(k);
        for (int b = 0; b < 64; ++b) {
            int flips = popcount64(hk ^ f(k ^ (1ULL << b)));
            perBit[b] += flips;
            total += flips;
        }
    }
    for (int b = 0; b < 64; ++b) {
        double avg = (double)perBit[b] / KEYS;
        if (avg < worstAvg) { worstAvg = avg; worstBit = b; }
    }
    std::printf("%-14s %7.2f %14.2f %5d\n", name, (double)total / (64.0 * KEYS),
                worstAvg, worstBit);
}
```

Kode 15.1 (Lanjutan) *hash_functions.cpp: metode pembagian dan perkalian, hash string penjumlahan dan polinomial, pembaruan bergulir, serta pengukuran avalanche.*

```
int main() {
    std::printf("1. Division method: book prices (rupiah) into m buckets\n");
    const u64 price[8] = {45000, 60000, 75000, 85000, 90000, 120000, 55000, 70000};
    const u64 ms[3] = {10, 16, 11};
    for (u64 m : ms) {
        std::printf("   m = %2llu:", (unsigned long long)m);
        for (u64 p : price)
            std::printf(" %2llu", (unsigned long long)hashDivision(p, m));
        std::printf("\n");
    }

    std::printf("\n2. Multiplication method, m = 16 (p = 4)\n");
    std::printf("   price    float  integer\n");
    for (u64 p : price)
        std::printf("   %6llu %6llu %8llu\n", (unsigned long long)p,
                    (unsigned long long)hashMultFloat(p, 16),
                    (unsigned long long)hashMultInt(p, 4));

    std::printf("\n3. Strings: sum of codes versus polynomial (base 31)\n");
    const char *words[6] = {"kasur", "rusak", "kita", "kiat", "B1001", "B1010"};
    for (const char *w : words)
        std::printf("   %-6s sum = %4llu   poly = %llu\n", w,
                    (unsigned long long)hashSum(w),
                    (unsigned long long)hashPoly(w));

    std::printf("\n   Horner's rule for \"B1001\":\n");
    u64 h = 0;
    for (char c : std::string("B1001")) {
        u64 before = h;
        h = h * 31 + (unsigned char)c;
        std::printf("   h = %8llu * 31 + %2d ('%c') = %llu\n",
                    (unsigned long long)before, c, c, (unsigned long long)h);
    }

    std::printf("\n4. Rolling hash of every 4-letter window of \"pustakaceria\"\n");
    const std::string text = "pustakaceria";
    const int W = 4;
    u64 pw = 1;                                          // 31^(W-1)
    for (int i = 1; i < W; ++i) pw *= 31;
    u64 roll = hashPoly(text.substr(0, W));
    int agree = 0, windows = 0;
    for (std::size_t i = 0; i + W <= text.size(); ++i) {
        if (i > 0)                                       // drop the old first letter,
            roll = (roll - (unsigned char)text[i - 1] * pw) * 31
                   + (unsigned char)text[i + W - 1];     // shift, add the new last
        u64 direct = hashPoly(text.substr(i, W));
```

Kode 15.1 (Lanjutan) *hash_functions.cpp: metode pembagian dan perkalian, hash string penjumlahan dan polinomial, pembaruan bergulir, serta pengukuran avalanche.*

```
        ++windows;
        if (roll == direct) ++agree;
        std::printf("   %s %9llu\n", text.substr(i, W).c_str(),
                    (unsigned long long)roll);
    }
    std::printf("   rolling equals direct in %d of %d windows\n", agree, windows);

    std::printf("\n5. Avalanche: output bits that flip when one input bit flips\n");
    std::printf("   (2000 random 64-bit keys x 64 input bits; ideal = 32.00)\n");
    std::printf("%-14s %7s %14s %5s\n", "   function", "average",
                "weakest input", "bit");
    avalanche("   identity", fIdentity);
    avalanche("   k*31+7", fTimes31);
    avalanche("   Fibonacci", fFibonacci);
    avalanche("   SplitMix64", fMix);

    std::printf("\n6. Codes B0000..B1999 into m = 101 buckets\n");
    std::printf("   hash    empty  largest\n");
    for (int which = 0; which < 2; ++which) {
        std::vector<int> count(101, 0);
        for (int i = 0; i < 2000; ++i) {
            char code[16];
            std::snprintf(code, sizeof code, "B%04d", i);
            u64 v = which == 0 ? hashSum(code) : hashPoly(code);
            ++count[v % 101];
        }
        int empty = 0, largest = 0;
        for (int c : count) {
            if (c == 0) ++empty;
            if (c > largest) largest = c;
        }
        std::printf("   %-6s %6d %8d\n", which == 0 ? "sum" : "poly", empty,
                    largest);
    }
    return 0;
}
```

Keluarannya, identik dengan versi Java dan Python di folder kode bab ini:

```
1. Division method: book prices (rupiah) into m buckets
   m = 10:  0  0  0  0  0  0  0  0
   m = 16:  8  0  8  8  0  0  8  0
   m = 11: 10  6  2  3  9  1  0  7

2. Multiplication method, m = 16 (p = 4)
   price    float  integer
    45000      8        8
    60000      0        0
    75000      8        8
    85000     14       14
    90000      0        0
   120000      1        1
    55000     13       13
    70000      6        6

3. Strings: sum of codes versus polynomial (base 31)
   kasur  sum =  550   poly = 101820730
   rusak  sum =  550   poly = 108880570
   kita   sum =  425   poly = 3292235
   kiat   sum =  425   poly = 3291665
   B1001  sum =  260   poly = 62459810
   B1010  sum =  260   poly = 62459840

   Horner's rule for "B1001":
   h =        0 * 31 + 66 ('B') = 66
   h =       66 * 31 + 49 ('1') = 2095
   h =     2095 * 31 + 48 ('0') = 64993
   h =    64993 * 31 + 48 ('0') = 2014831
   h =  2014831 * 31 + 49 ('1') = 62459810

4. Rolling hash of every 4-letter window of "pustakaceria"
   pust   3452710
   usta   3599755
   stak   3540555
   taka   3552387
   akac   2995660
   kace   3284024
   acer   2988111
   ceri   3050009
   eria   3121797
   rolling equals direct in 9 of 9 windows

5. Avalanche: output bits that flip when one input bit flips
   (2000 random 64-bit keys x 64 input bits; ideal = 32.00)
   function    average  weakest input   bit
   identity       1.00          1.00     0
   k*31+7         3.67          1.00    63
   Fibonacci     16.02          1.00    63
   SplitMix64    31.97         31.80    25

6. Codes B0000..B1999 into m = 101 buckets
   hash    empty  largest
   sum        72      150
   poly        0       23
```

Percobaan terakhir meringkas subbab ini. Dua ribu kode yang sama, 101 *bucket* yang sama: penjumlahan karakter membiarkan 72 *bucket* kosong dan menumpuk 150 kode dalam satu *bucket*; hash polinomial memakai setiap *bucket* dan tidak pernah memasukkan lebih dari 23 kode ke satu *bucket*, dekat dengan $2000/101 \approx 19.8$ yang akan diberikan oleh penyebaran sempurna.

> **HASH UNTUK TABEL BUKAN HASH UNTUK KATA SANDI.** Segala sesuatu dalam subbab ini mengejar kecepatan. Hash *kriptografis* (SHA-256) justru mengejar agar mustahil secara komputasi menemukan dua masukan dengan keluaran yang sama, dan hash *kata sandi*

(Argon2id, bcrypt) sengaja dibuat lambat dan rakus memori agar menebak miliaran kata sandi menjadi mahal. Jangan pernah menyimpan kata sandi dengan `hashPoly` atau `std::hash`, dan jangan pernah memakai hash kata sandi untuk mengindeks tabel.

15.4 Tabrakan dan Batas Ulang Tahun

15.4.1 Tabrakan tidak dapat dihindari

Tabrakan bukan tanda fungsi hash yang buruk. Begitu semesta U memiliki lebih banyak kunci daripada slot tabel, *prinsip sarang merpati* (*pigeonhole principle*) menjamin bahwa dua kunci berbagi slot: bila $|U| > m$ kunci ditempatkan ke m slot, suatu slot menerima sedikitnya dua. Untuk judul buku dan $m = 7$ hal ini jelas. Yang mengejutkan adalah seberapa *cepat* tabrakan muncul bahkan ketika hanya sedikit kunci disimpan dalam tabel yang besar.

15.4.2 Soal ulang tahun

Berapa orang harus berada di sebuah ruangan sebelum dua di antaranya kemungkinan besar berulang tahun pada hari yang sama? Banyak orang menebak sekitar 183, separuh dari 365. Jawabannya 23. Perhitungan yang sama memberi tahu seberapa cepat n kunci bertabrakan dalam tabel berisi m slot di bawah hashing seragam sederhana, Persamaan (15.1).

Sisipkan kunci satu per satu dan tanyakan peluang bahwa *belum* terjadi tabrakan. Kunci pertama dapat masuk ke mana saja. Kunci kedua menghindari yang pertama dengan peluang $(m-1)/m = 1 - 1/m$. Kunci ketiga harus menghindari dua slot terisi, peluangnya $1 - 2/m$. Secara umum kunci ke-$(i+1)$ menghindari i slot terisi dengan peluang $1 - i/m$, bebas dari bagaimana kunci-kunci sebelumnya jatuh. Dengan mengalikan,

$$\Pr[\text{tak ada tabrakan di antara } n \text{ kunci}] = \prod_{i=1}^{n-1}\left(1 - \frac{i}{m}\right). \tag{15.7}$$

Hasil kali ini eksak tetapi menyembunyikan bentuknya. Untuk x kecil berlaku $1 - x \approx e^{-x}$ (garis singgung e^{-x} di 0, dan selalu $1 - x \le e^{-x}$), sehingga setiap faktor kira-kira $e^{-i/m}$, dan hasil kali eksponensial adalah eksponensial dari jumlahnya:

$$\Pr[\text{tak ada tabrakan}] \approx \exp\left(-\sum_{i=1}^{n-1}\frac{i}{m}\right) = \exp\left(-\frac{n(n-1)}{2m}\right). \tag{15.8}$$

Peluang tabrakan mencapai setengah ketika $n(n-1)/(2m) = \ln 2$, yaitu ketika $n^2 \approx 2m\ln 2$:

$$n_{1/2} \approx \sqrt{2\ln 2 \cdot m} \approx 1.1774\sqrt{m}. \tag{15.9}$$

Untuk $m = 365$ ini memberi $1.1774 \times 19.10 = 22.5$, jadi 23 orang. *Akar kuadrat* itulah seluruh ceritanya: tabrakan menjadi mungkin ketika banyaknya kunci mencapai sekitar akar kuadrat banyaknya slot, bukan pecahan tetap darinya.

Ada cara yang lebih sederhana lagi untuk melihat bilangan yang sama. Di antara n kunci terdapat $\binom{n}{2} = n(n-1)/2$ pasangan, dan setiap pasangan bertabrakan dengan peluang $1/m$ menurut Persamaan (15.1). Dengan linearitas nilai harapan,

$$E[\text{banyaknya pasangan bertabrakan}] = \frac{n(n-1)}{2m}. \tag{15.10}$$

Untuk 23 kunci dalam 365 slot nilainya $253/365 \approx 0.69$ pasangan bertabrakan yang diharapkan, yaitu eksponen pada Persamaan (15.8).

15.4.3 Bilangan-bilangannya, dihitung

Kode 15.2 menghitung Persamaan (15.7) secara eksak, membandingkannya dengan Persamaan (15.8), mencari n terkecil yang mencapai 50% dan 99% untuk beberapa ukuran tabel, dan memeriksa aritmetikanya dengan simulasi 100.000 ruangan berisi 23 kunci acak.

Kode 15.2 `birthday.cpp`: peluang tabrakan eksak dan hampiran, banyaknya kunci untuk peluang seimbang dan hampir pasti, serta sebuah simulasi.

```
// birthday.cpp -- how soon do random keys collide? The birthday bound of
// Section 15.4: exact probabilities, the exponential approximation, the number
// of keys that gives even odds, and a simulation that checks the arithmetic.
#include <cmath>
#include <cstdint>
#include <cstdio>
#include <vector>

typedef std::uint64_t u64;

u64 rngState = 15;                                  // SplitMix64, fixed seed
u64 nextRandom() {
    u64 z = (rngState += 0x9E3779B97F4A7C15ULL);
    z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9ULL;
    z = (z ^ (z >> 27)) * 0x94D049BB133111EBULL;
    return z ^ (z >> 31);
}

// Exact probability that n keys thrown into m slots are all different.
double allDistinct(long long n, double m) {
    double p = 1.0;
    for (long long i = 1; i < n; ++i) p *= 1.0 - (double)i / m;
    return p;
}

// Smallest n for which the probability of a collision reaches `target`.
long long keysFor(double m, double target) {
    double p = 1.0;                                 // P(all distinct) so far
    long long n = 1;
    while (1.0 - p < target) {
        p *= 1.0 - (double)n / m;
        ++n;
    }
    return n;
}
```

Kode 15.2 (Lanjutan) *birthday.cpp: peluang tabrakan eksak dan hampiran, banyaknya kunci untuk peluang seimbang dan hampir pasti, serta sebuah simulasi.*

```
int main() {
    std::printf("1. m = 365 slots: probability of at least one collision\n");
    std::printf("    n    exact   approx  expected pairs\n");
    const int ns[8] = {10, 20, 23, 30, 40, 50, 57, 70};
    for (int n : ns) {
        double exact = 1.0 - allDistinct(n, 365.0);
        double approx = 1.0 - std::exp(-(double)n * (n - 1) / (2.0 * 365.0));
        std::printf("  %3d   %.4f   %.4f   %6.3f\n", n, exact, approx,
                    (double)n * (n - 1) / (2.0 * 365.0));
    }

    std::printf("\n2. Keys needed for a 50%% and a 99%% chance of a collision\n");
    std::printf("  %-8s %12s %14s %12s\n", "m", "n (50%)", "1.1774*sqrt(m)",
                "n (99%)");
    const double msz[5] = {365.0, 1000.0, 1e6, 4294967296.0,
                           18446744073709551616.0};
    const char *mname[5] = {"365", "1000", "10^6", "2^32", "2^64"};
    for (int i = 0; i < 5; ++i) {
        double m = msz[i];
        double rule = std::sqrt(2.0 * m * std::log(2.0));
        if (i < 4) {
            std::printf("  %-8s %12lld %14.1f %12lld\n", mname[i], keysFor(m, 0.5),
                        rule, keysFor(m, 0.99));
        } else {                            // too many steps: formula only
            std::printf("  %-8s %12s %14.3e %12.3e\n", mname[i], "(formula)", rule,
                        std::sqrt(2.0 * m * std::log(100.0)));
        }
    }

    std::printf("\n3. Simulation: 23 random keys into 365 slots, 100000 trials\n");
    const int TRIALS = 100000;
    int withCollision = 0;
    std::vector<int> seen(365, -1);
    for (int t = 0; t < TRIALS; ++t) {
        bool hit = false;
        for (int k = 0; k < 23; ++k) {
            int slot = (int)(nextRandom() % 365);
            if (seen[slot] == t) hit = true;   // slot already used this trial
            seen[slot] = t;
        }
        if (hit) ++withCollision;
    }
    std::printf("  trials with a collision: %d  (%.4f; exact %.4f)\n",
                withCollision, (double)withCollision / TRIALS,
                1.0 - allDistinct(23, 365.0));
```

Kode 15.2 (Lanjutan) *birthday.cpp: peluang tabrakan eksak dan hampiran, banyaknya kunci untuk peluang seimbang dan hampir pasti, serta sebuah simulasi.*

```
    std::printf("\n4. n = m = 1000 keys into 1000 buckets: "
                "how many buckets stay empty?\n");
    std::vector<int> count(1000, 0);
    for (int k = 0; k < 1000; ++k) ++count[nextRandom() % 1000];
    int empty = 0, largest = 0;
    for (int c : count) {
        if (c == 0) ++empty;
        if (c > largest) largest = c;
    }
    std::printf("  empty buckets: %d  (expected 1000*(1-1/1000)^1000 = %.1f)\n",
                empty, 1000.0 * std::pow(1.0 - 1.0 / 1000.0, 1000.0));
    std::printf("  largest bucket: %d keys\n", largest);
    return 0;
}
```

```
1. m = 365 slots: probability of at least one collision
    n    exact   approx  expected pairs
   10   0.1169   0.1160    0.123
   20   0.4114   0.4058    0.521
   23   0.5073   0.5000    0.693
   30   0.7063   0.6963    1.192
   40   0.8912   0.8820    2.137
   50   0.9704   0.9651    3.356
   57   0.9901   0.9874    4.373
   70   0.9992   0.9987    6.616

2. Keys needed for a 50% and a 99% chance of a collision
  m             n (50%) 1.1774*sqrt(m)      n (99%)
  365               23          22.5           57
  1000              38          37.2           95
  10^6            1178        1177.4         3034
  2^32           77164       77162.7       198892
  2^64       (formula)     5.057e+09    1.303e+10

3. Simulation: 23 random keys into 365 slots, 100000 trials
  trials with a collision: 50650  (0.5065; exact 0.5073)

4. n = m = 1000 keys into 1000 buckets: how many buckets stay empty?
  empty buckets: 353  (expected 1000*(1-1/1000)^1000 = 367.7)
  largest bucket: 5 keys
```

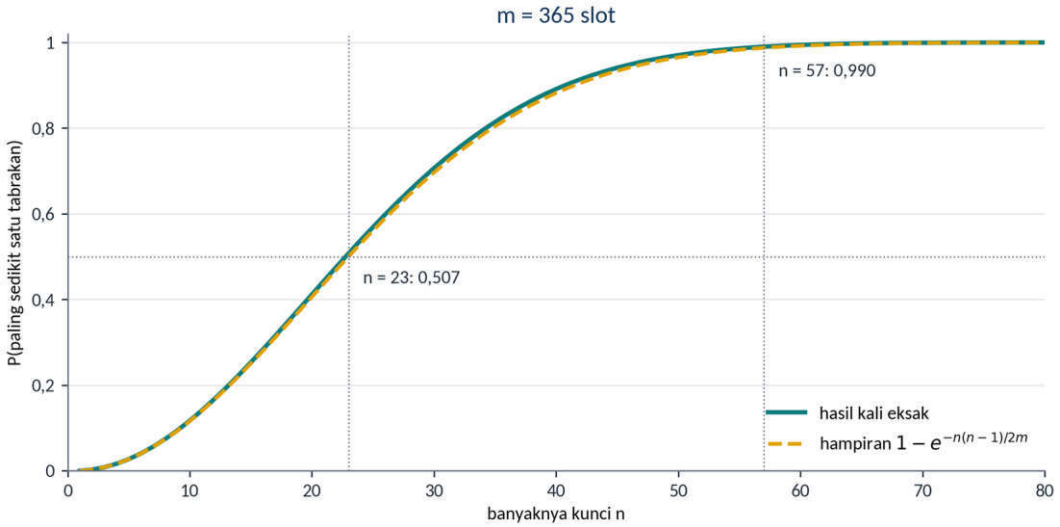
Tabel 15.2 menghimpun bagian kedua keluaran itu. Bacalah baris terakhirnya pelan-pelan: bahkan dengan 2^{64} kemungkinan nilai hash, sekitar lima miliar kunci acak sudah cukup untuk peluang tabrakan yang seimbang. Lima miliar adalah banyaknya halaman yang ditangani sebuah *crawler* web besar, dan karena itulah sistem yang mengenali objek dengan hash 64-bit (dan mengandaikan tidak ada tabrakan) harus memakai 128 bit atau memeriksa kunci lengkapnya.

Tabel 15.2 Berapa banyak kunci acak yang membuat tabrakan mungkin terjadi, dihitung oleh **Kode 15.2** dari Persamaan (15.7); kolom tengah adalah hampiran Persamaan (15.9).

Slot m	n Untuk peluang 50%	$1.1774\sqrt{(m)}$	n Untuk peluang 99%
365	23	22,5	57
1000	38	37,2	95
10^6	1178	1177,4	3034
2^{32}	77.164	77.162,7	198.892
2^{64}	—	5.057×10^9	1.303×10^{10}

Gambar 15.2 menggambarkan peluang itu untuk $m = 365$. Hasil kali eksak dan hampirannya nyaris tak dapat dibedakan; hampirannya selalu sedikit lebih rendah, karena $1 - x \leq e^{-x}$

membuat peluang sebenarnya untuk *tidak* bertabrakan sedikit lebih kecil daripada kata hampiran itu.



Gambar 15.2 Peluang sedikitnya satu tabrakan di antara n kunci acak dalam 365 slot: 0,507 pada $n = 23$ dan 0,990 pada $n = 57$. Kurva putus-putus adalah hampiran Persamaan (15.8).

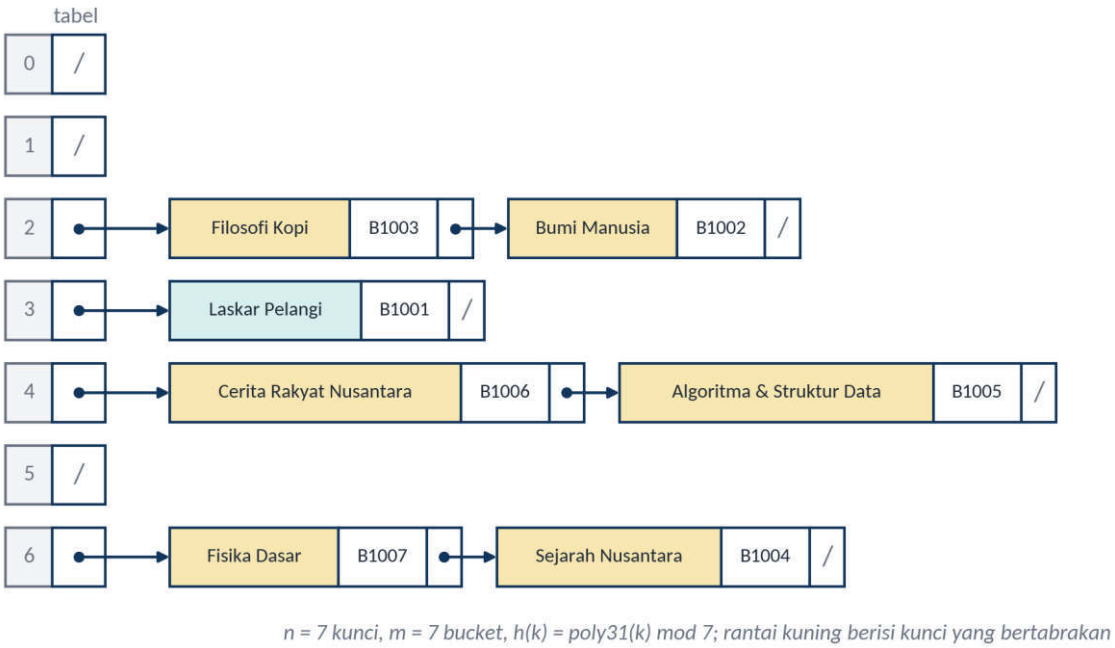
15.4.4 Artinya bagi tabel hash

Tabel hash tidak berusaha menghindari tabrakan; batas ulang tahun memperlihatkan bahwa hal itu mustahil, karena tabel akan memerlukan sekitar n^2 slot untuk n kunci. Tabel hash *merencanakan* tabrakan. Bagian 4 keluaran membuat rencana itu konkret: 1000 kunci dalam 1000 *bucket* menyisakan 353 *bucket* kosong (nilai harapannya $1000 \cdot (1 - 1/1000)^{1000} \approx 1000/e \approx 367.7$) dan menaruh lima kunci di *bucket* terpenuh. Jadi tabel dengan $n = m$ memiliki sekitar 37% *bucket* kosong dan beberapa *bucket* berisi beberapa kunci, dan menurut Persamaan (15.10) tabel itu memuat sekitar $1000 \cdot 999/2000 \approx 500$ pasangan bertabrakan. Dua subbab berikut memberikan dua cara klasik untuk hidup dengan hal itu: biarkan *bucket* menampung sebuah list (*chaining*), atau biarkan kunci berpindah ke slot lain (pengalamatan terbuka).

15.5 *Separate Chaining*

15.5.1 Sebuah *linked list* di setiap *bucket*

Pada *separate chaining* (perantaian terpisah) setiap slot tabel menyimpan pointer kepala sebuah *single linked list*, list dari Bab 7, dan setiap kunci yang di-hash ke slot itu menjadi simpul list tersebut. **Gambar 15.3** memperlihatkan tabel yang dibangun **Kode 15.3** dari tujuh judul Pustaka Ceria dan tabel berisi $m = 7$ *bucket*. Kuncinya judul dan nilainya kode buku, sehingga tabel itu menjawab "kode apa yang dimiliki judul ini?"



Gambar 15.3 *Separate chaining*: tujuh judul dalam tujuh *bucket* dengan $h(k) = \text{poly31}(k) \bmod 7$. Setiap *bucket* menyimpan kepala sebuah *single linked list*; *bucket* 2, 4, dan 6 masing-masing berisi dua judul yang bertabrakan, dan *bucket* 0, 1, dan 5 kosong.

Ketiga operasinya menyusut menjadi operasi list Bab 7 yang diterapkan pada satu list pendek.

Pseudokode 15.2 Pencarian, penyisipan, dan penghapusan pada tabel hash dengan *separate chaining*.

```
CHAINED-SEARCH(T, k)
1  p = T[h(k)]                        // kepala rantai untuk k
2  while p != NIL and p.key != k
3      p = p.next
4  return p                           // NIL bila k tidak ada

CHAINED-PUT(T, k, v)
1  p = CHAINED-SEARCH(T, k)
2  if p != NIL
3      p.value = v                    // ada: perbarui di tempat
4  else
5      sisipkan simpul baru (k, v) di depan rantai T[h(k)]
6      n = n + 1
7      if n / m > maxLoad
8          REHASH(T, bilangan prima berikutnya >= 2m)

CHAINED-DELETE(T, k)
1  telusuri rantai T[h(k)] dengan prev dan cur hingga cur.key == k
2  lepaskan cur (seperti DELETE-KEY pada Subbab 7.6.5) dan bebaskan
```

Penyisipan dilakukan di *depan* rantai, `frontInsertion` $O(1)$ dari Subbab 7.5.2, tetapi hanya sesudah sebuah pencarian, karena tabel memetakan setiap kunci ke *satu* nilai: memasukkan kunci yang sudah ada sekali lagi harus memperbaruinya, bukan menambah simpul kedua. Penghapusan adalah `deleteKey` dari Subbab 7.6.5 dengan pointer pengekornya `prev`. Baris 7 `CHAINED-PUT` mengawasi faktor muat; Subbab 15.7 menjelaskan mengapa dan kapan melakukan *rehash*.

15.5.2 Programnya

Kode 15.3 mengimplementasikan **Pseudokode 15.2** sebagai sebuah kelas. Anggota `table_` adalah `std::vector<Node *>` berisi kepala-kepala rantai, sehingga `table_[b]` memerankan `head`

dari Bab 7, satu per *bucket*. Destruktornya menyusuri setiap rantai dan menghapus setiap simpul, dan penyalinan dilarang, sesuai aturan tiga (*rule of three*) pada Subbab 7.10.2. *Rehash*-nya memindahkan simpul alih-alih menyalinnya: setiap simpul dilepas dari rantai lamanya dan didorong ke depan rantai barunya, sehingga sama sekali tidak ada `new` atau `delete` selama *rehash*.

Kode 15.3 `chaining.cpp`: tabel hash dengan *separate chaining* yang dibangun dari simpul-simpul *single linked list*, dengan pencarian, pembaruan, penghapusan, dan *rehashing*.

```
// chaining.cpp -- a hash table with separate chaining, built by hand from the
// singly linked nodes of Chapter 7. Keys are book titles, values are book codes.
#include <cstdint>
#include <cstdio>
#include <string>
#include <vector>

std::uint64_t hashPoly(const std::string &s) {       // Section 15.3.4, base 31
    std::uint64_t h = 0;
    for (unsigned char c : s) h = h * 31 + c;
    return h;
}

struct Node {
    std::string key;
    std::string value;
    Node *next;
};

class ChainedHashTable {
public:
    explicit ChainedHashTable(std::size_t m = 7, double maxLoad = 1.0)
        : table_(m, nullptr), n_(0), maxLoad_(maxLoad) {}

    ~ChainedHashTable() {                            // release every node
        for (Node *head : table_) {
            while (head != nullptr) {
                Node *del = head;
                head = head->next;
                delete del;
            }
        }
    }

    ChainedHashTable(const ChainedHashTable &) = delete;   // no copies (rule of
    ChainedHashTable &operator=(const ChainedHashTable &) = delete;   // three)

    std::size_t bucketOf(const std::string &key) const {
        return hashPoly(key) % table_.size();
    }
```

Kode 15.3 (Lanjutan) *chaining.cpp: tabel hash dengan separate chaining yang dibangun dari simpul-simpul single linked list, dengan pencarian, pembaruan, penghapusan, dan rehashing.*

```
    // Insert key with value, or replace the value if key is already present.
    void put(const std::string &key, const std::string &value) {
        std::size_t b = bucketOf(key);
        for (Node *p = table_[b]; p != nullptr; p = p->next) {
            if (p->key == key) {                    // present: update in place
                p->value = value;
                return;
            }
        }
        table_[b] = new Node{key, value, table_[b]};   // absent: front insertion
        ++n_;
        if ((double)n_ / table_.size() > maxLoad_)
            rehash(nextPrime(2 * table_.size()));
    }

    // Return a pointer to the value stored for key, or nullptr if key is absent.
    // `compared` receives the number of keys compared along the way.
    const std::string *get(const std::string &key, int &compared) const {
        compared = 0;
        for (Node *p = table_[bucketOf(key)]; p != nullptr; p = p->next) {
            ++compared;
            if (p->key == key) return &p->value;
        }
        return nullptr;
    }

    bool erase(const std::string &key) {            // deleteKey of Section 7.6.5
        std::size_t b = bucketOf(key);
        Node *prev = nullptr, *cur = table_[b];
        while (cur != nullptr && cur->key != key) {
            prev = cur;
            cur = cur->next;
        }
        if (cur == nullptr) return false;           // not present
        if (prev == nullptr) table_[b] = cur->next;  // first node of the chain
        else prev->next = cur->next;
        delete cur;
        --n_;
        return true;
    }
```

Kode 15.3 (Lanjutan) *chaining.cpp: tabel hash dengan separate chaining yang dibangun dari simpul-simpul single linked list, dengan pencarian, pembaruan, penghapusan, dan rehashing.*

```
    void show() const {
        int empty = 0, longest = 0;
        for (std::size_t b = 0; b < table_.size(); ++b) {
            int len = 0;
            std::printf("  %2zu:", b);
            for (Node *p = table_[b]; p != nullptr; p = p->next, ++len)
                std::printf(" %s=%s ->", p->key.c_str(), p->value.c_str());
            std::printf(len == 0 ? " (empty)\n" : " NULL\n");
            if (len == 0) ++empty;
            if (len > longest) longest = len;
        }
        std::printf("  n = %zu, m = %zu, load factor = %.2f, longest chain = %d, "
                    "empty buckets = %d\n", n_, table_.size(),
                    (double)n_ / table_.size(), longest, empty);
    }

private:
    static bool isPrime(std::size_t x) {
        if (x < 2) return false;
        for (std::size_t d = 2; d * d <= x; ++d)
            if (x % d == 0) return false;
        return true;
    }
    static std::size_t nextPrime(std::size_t x) {
        while (!isPrime(x)) ++x;
        return x;
    }

    // Move every node into a new array of newM buckets. No node is created or
    // deleted: each one is unlinked from its old chain and pushed on the front
    // of its new chain, because its bucket number depends on m.
    void rehash(std::size_t newM) {
        std::printf("  -- load factor %.2f > %.2f: rehash from %zu to %zu "
                    "buckets\n",
                    (double)n_ / table_.size(), maxLoad_, table_.size(), newM);
        std::vector<Node *> old(newM, nullptr);
        old.swap(table_);                           // table_ is now the new array
        for (Node *head : old) {
            while (head != nullptr) {
                Node *moving = head;
                head = head->next;
                std::size_t b = bucketOf(moving->key);
                moving->next = table_[b];
                table_[b] = moving;
            }
        }
```

Kode 15.3 (Lanjutan) *chaining.cpp: tabel hash dengan separate chaining yang dibangun dari simpul-simpul single linked list, dengan pencarian, pembaruan, penghapusan, dan rehashing.*

```
    }

    std::vector<Node *> table_;                     // table_[b] = head of chain b
    std::size_t n_;                                 // number of keys stored
    double maxLoad_;                                // rehash when n/m exceeds it
};

void lookup(const ChainedHashTable &t, const std::string &title) {
    int compared;
    const std::string *code = t.get(title, compared);
    std::printf("  get(\"%s\") -> %s, bucket %zu, %d key(s) compared\n",
                title.c_str(), code ? code->c_str() : "not found",
                t.bucketOf(title), compared);
}

int main() {
    const char *title[8] = {"Laskar Pelangi", "Bumi Manusia", "Filosofi Kopi",
                            "Sejarah Nusantara", "Algoritma & Struktur Data",
                            "Cerita Rakyat Nusantara", "Fisika Dasar",
                            "Negeri 5 Menara"};
    ChainedHashTable t(7, 1.0);

    std::printf("1. Seven titles into m = 7 buckets, h(k) = poly31(k) mod 7\n");
    for (int i = 0; i < 7; ++i) {
        std::string code = "B100" + std::to_string(i + 1);
        std::printf("  put(\"%s\", %s) -> bucket %zu\n", title[i], code.c_str(),
                    t.bucketOf(title[i]));
        t.put(title[i], code);
    }
    t.show();

    std::printf("\n2. Searching\n");
    lookup(t, "Filosofi Kopi");
    lookup(t, "Bumi Manusia");
    lookup(t, "Ayat-Ayat Cinta");
    lookup(t, "Pulang");

    std::printf("\n3. Updating and deleting\n");
    t.put("Laskar Pelangi", "B1001+B1011");          // a second copy arrived
    lookup(t, "Laskar Pelangi");
    std::printf("  erase(\"Algoritma & Struktur Data\") -> %s\n",
                t.erase("Algoritma & Struktur Data") ? "deleted" : "not found");
    std::printf("  erase(\"Pulang\") -> %s\n",
                t.erase("Pulang") ? "deleted" : "not found");
    t.show();
```

Kode 15.3 (Lanjutan) *chaining.cpp: tabel hash dengan separate chaining yang dibangun dari simpul-simpul single linked list, dengan pencarian, pembaruan, penghapusan, dan rehashing.*

```
    std::printf("\n4. Two more titles: the eighth key pushes n/m above 1\n");
    t.put("Algoritma & Struktur Data", "B1005");
    t.put("Negeri 5 Menara", "B1008");
    t.show();
    lookup(t, "Bumi Manusia");
    return 0;
}
```

```
1. Seven titles into m = 7 buckets, h(k) = poly31(k) mod 7
  put("Laskar Pelangi", B1001) -> bucket 3
  put("Bumi Manusia", B1002) -> bucket 2
  put("Filosofi Kopi", B1003) -> bucket 2
  put("Sejarah Nusantara", B1004) -> bucket 6
  put("Algoritma & Struktur Data", B1005) -> bucket 4
  put("Cerita Rakyat Nusantara", B1006) -> bucket 4
  put("Fisika Dasar", B1007) -> bucket 6
   0: (empty)
   1: (empty)
   2: Filosofi Kopi=B1003 -> Bumi Manusia=B1002 -> NULL
   3: Laskar Pelangi=B1001 -> NULL
   4: Cerita Rakyat Nusantara=B1006 -> Algoritma & Struktur Data=B1005 -> NULL
   5: (empty)
   6: Fisika Dasar=B1007 -> Sejarah Nusantara=B1004 -> NULL
  n = 7, m = 7, load factor = 1.00, longest chain = 2, empty buckets = 3

2. Searching
  get("Filosofi Kopi") -> B1003, bucket 2, 1 key(s) compared
  get("Bumi Manusia") -> B1002, bucket 2, 2 key(s) compared
  get("Ayat-Ayat Cinta") -> not found, bucket 4, 2 key(s) compared
  get("Pulang") -> not found, bucket 1, 0 key(s) compared

3. Updating and deleting
  get("Laskar Pelangi") -> B1001+B1011, bucket 3, 1 key(s) compared
  erase("Algoritma & Struktur Data") -> deleted
  erase("Pulang") -> not found
   0: (empty)
   1: (empty)
   2: Filosofi Kopi=B1003 -> Bumi Manusia=B1002 -> NULL
   3: Laskar Pelangi=B1001+B1011 -> NULL
   4: Cerita Rakyat Nusantara=B1006 -> NULL
   5: (empty)
   6: Fisika Dasar=B1007 -> Sejarah Nusantara=B1004 -> NULL
  n = 6, m = 7, load factor = 0.86, longest chain = 2, empty buckets = 3

4. Two more titles: the eighth key pushes n/m above 1
  -- load factor 1.14 > 1.00: rehash from 7 to 17 buckets
   0: Bumi Manusia=B1002 -> NULL
   1: Fisika Dasar=B1007 -> NULL
   2: (empty)
   3: (empty)
   4: Negeri 5 Menara=B1008 -> NULL
   5: Cerita Rakyat Nusantara=B1006 -> NULL
   6: (empty)
   7: Laskar Pelangi=B1001+B1011 -> NULL
   8: (empty)
   9: Filosofi Kopi=B1003 -> NULL
  10: (empty)
  11: (empty)
  12: (empty)
  13: (empty)
  14: Sejarah Nusantara=B1004 -> NULL
  15: Algoritma & Struktur Data=B1005 -> NULL
  16: (empty)
  n = 8, m = 17, load factor = 0.47, longest chain = 1, empty buckets = 9
  get("Bumi Manusia") -> B1002, bucket 0, 1 key(s) compared
```

Ikutilah jalannya program bersama **Gambar 15.3**. Bagian 1 menyisipkan tujuh judul; *Filosofi Kopi* masuk ke *bucket* 2 sesudah *Bumi Manusia*, sehingga penyisipan di depan menaruhnya pertama dalam rantai. Bagian 2 memperlihatkan ongkos pencarian dalam *perbandingan kunci*: *Filosofi Kopi* ditemukan sesudah satu perbandingan, *Bumi Manusia*, yang kedua dalam rantai yang sama, sesudah dua. Judul yang tidak ada berongkos sepanjang rantainya: *Ayat-Ayat Cinta* di-hash ke *bucket* 4, dibandingkan dengan kedua judul di sana lalu menyerah, sedangkan *Pulang* di-hash ke

bucket 1 yang kosong dan ditolak tanpa satu perbandingan pun. Bagian 3 memperbarui nilai sebuah kunci yang sudah ada (ukurannya tetap 7) dan menghapus simpul *kedua* rantai 4, yang menguji cabang `prev->next = cur->next`. Pada bagian 4 kunci kedelapan membuat $n/m = 8/7 \approx 1.14$ melampaui 1, tabel tumbuh menjadi 17 *bucket*, bilangan prima berikutnya sesudah 14, dan setiap judul pindah ke *bucket* baru yang dihitung modulo 17. Sesudah *rehash* tidak ada rantai yang lebih panjang dari 1.

15.5.3 Tabel yang sama dalam Java dan Python

Kode 15.4 dan **Kode 15.5** adalah terjemahan baris demi baris. Keduanya mencetak keluaran yang persis sama dengan di atas, karena memakai hash polinomial yang sama (aritmetika `long` di Java, `& MASK` di Python) alih-alih `hashCode` atau `hash` bawaan bahasa, yang berbeda antarbahasa dan, di Python, antarjalannya program.

Kode 15.4 `Chaining.java`: tabel hash berantai dalam Java.

```java
// Chaining.java -- a hash table with separate chaining, built by hand from the
// singly linked nodes of Chapter 7. Keys are book titles, values are book codes.
import java.util.Arrays;

public class Chaining {
    static long hashPoly(String s) {                // Section 15.3.4, base 31
        long h = 0;
        for (int i = 0; i < s.length(); i++) h = h * 31 + s.charAt(i);
        return h;
    }

    static class Node {
        final String key;
        String value;
        Node next;

        Node(String key, String value, Node next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }

    static class ChainedHashTable {
        private Node[] table;                       // table[b] = head of chain b
        private int n;                              // number of keys stored
        private final double maxLoad;               // rehash when n/m exceeds it

        ChainedHashTable(int m, double maxLoad) {
            table = new Node[m];
            this.maxLoad = maxLoad;
        }

        int bucketOf(String key) {
            return (int) Long.remainderUnsigned(hashPoly(key), table.length);
        }
```

Kode 15.4 (Lanjutan) *Chaining.java: tabel hash berantai dalam Java.*

```
    // Insert key with value, or replace the value if key is already present.
    void put(String key, String value) {
        int b = bucketOf(key);
        for (Node p = table[b]; p != null; p = p.next) {
            if (p.key.equals(key)) {            // present: update in place
                p.value = value;
                return;
            }
        }
        table[b] = new Node(key, value, table[b]);   // absent: front insertion
        n++;
        if ((double) n / table.length > maxLoad)
            rehash(nextPrime(2 * table.length));
    }

    // Return the value stored for key, or null; compared[0] gets the count.
    String get(String key, int[] compared) {
        compared[0] = 0;
        for (Node p = table[bucketOf(key)]; p != null; p = p.next) {
            compared[0]++;
            if (p.key.equals(key)) return p.value;
        }
        return null;
    }

    boolean erase(String key) {                 // deleteKey of Section 7.6.5
        int b = bucketOf(key);
        Node prev = null, cur = table[b];
        while (cur != null && !cur.key.equals(key)) {
            prev = cur;
            cur = cur.next;
        }
        if (cur == null) return false;          // not present
        if (prev == null) table[b] = cur.next;  // first node of the chain
        else prev.next = cur.next;
        n--;
        return true;
    }
```

Kode 15.4 (Lanjutan) *Chaining.java: tabel hash berantai dalam Java.*

```
    void show() {
        int empty = 0, longest = 0;
        for (int b = 0; b < table.length; b++) {
            int len = 0;
            StringBuilder sb = new StringBuilder(String.format("  %2d:", b));
            for (Node p = table[b]; p != null; p = p.next, len++)
                sb.append(' ').append(p.key).append('=').append(p.value)
                  .append(" ->");
            sb.append(len == 0 ? " (empty)" : " NULL");
            System.out.println(sb);
            if (len == 0) empty++;
            if (len > longest) longest = len;
        }
        System.out.printf("  n = %d, m = %d, load factor = %.2f, longest "
                        + "chain = %d, "
                          + "empty buckets = %d%n", n, table.length,
                          (double) n / table.length, longest, empty);
    }

    private static boolean isPrime(int x) {
        if (x < 2) return false;
        for (int d = 2; d * d <= x; d++)
            if (x % d == 0) return false;
        return true;
    }

    private static int nextPrime(int x) {
        while (!isPrime(x)) x++;
        return x;
    }

    // Move every node into a new array of newM buckets, relinking, not copying.
    private void rehash(int newM) {
        System.out.printf("  -- load factor %.2f > %.2f: rehash from %d to %d "
                          + "buckets%n", (double) n / table.length, maxLoad,
                          table.length, newM);
        Node[] old = table;
        table = new Node[newM];
        for (Node head : old) {
            while (head != null) {
                Node moving = head;
                head = head.next;
                int b = bucketOf(moving.key);
                moving.next = table[b];
                table[b] = moving;
            }
```

Kode 15.4 (Lanjutan) *Chaining.java: tabel hash berantai dalam Java.*

```
            }
        }
    }

    static void lookup(ChainedHashTable t, String title) {
        int[] compared = new int[1];
        String code = t.get(title, compared);
        System.out.printf("  get(\"%s\") -> %s, bucket %d, %d key(s) compared%n",
                          title, code != null ? code : "not found",
                          t.bucketOf(title), compared[0]);
    }

    public static void main(String[] args) {
        String[] title = {"Laskar Pelangi", "Bumi Manusia", "Filosofi Kopi",
                          "Sejarah Nusantara", "Algoritma & Struktur Data",
                          "Cerita Rakyat Nusantara", "Fisika Dasar",
                          "Negeri 5 Menara"};
        ChainedHashTable t = new ChainedHashTable(7, 1.0);

        System.out.println("1. Seven titles into m = 7 buckets, h(k) = poly31(k) "
                           + "mod 7");
        for (int i = 0; i < 7; i++) {
            String code = "B100" + (i + 1);
            System.out.printf("  put(\"%s\", %s) -> bucket %d%n", title[i], code,
                              t.bucketOf(title[i]));
            t.put(title[i], code);
        }
        t.show();

        System.out.println("\n2. Searching");
        lookup(t, "Filosofi Kopi");
        lookup(t, "Bumi Manusia");
        lookup(t, "Ayat-Ayat Cinta");
        lookup(t, "Pulang");

        System.out.println("\n3. Updating and deleting");
        t.put("Laskar Pelangi", "B1001+B1011");      // a second copy arrived
        lookup(t, "Laskar Pelangi");
        boolean gone = t.erase("Algoritma & Struktur Data");
        System.out.println("  erase(\"Algoritma & Struktur Data\") -> "
                           + (gone ? "deleted" : "not found"));
        System.out.println("  erase(\"Pulang\") -> "
                           + (t.erase("Pulang") ? "deleted" : "not found"));
        t.show();
```

Kode 15.4 (Lanjutan) *Chaining.java: tabel hash berantai dalam Java.*

```
        System.out.println("\n4. Two more titles: the eighth key pushes n/m "
                           + "above 1");
        t.put("Algoritma & Struktur Data", "B1005");
        t.put("Negeri 5 Menara", "B1008");
        t.show();
        lookup(t, "Bumi Manusia");
    }
}
```

Kode 15.5 `chaining.py`: tabel hash berantai dalam Python.

```
# chaining.py -- a hash table with separate chaining, built by hand from the
# singly linked nodes of Chapter 7. Keys are book titles, values are book codes.
MASK = (1 << 64) - 1


def hash_poly(s):                                       # Section 15.3.4, base 31
    h = 0
    for c in s:
        h = (h * 31 + ord(c)) & MASK
    return h


class Node:
    __slots__ = ("key", "value", "next")

    def __init__(self, key, value, nxt):
        self.key, self.value, self.next = key, value, nxt


def is_prime(x):
    if x < 2:
        return False
    d = 2
    while d * d <= x:
        if x % d == 0:
            return False
        d += 1
    return True


def next_prime(x):
    while not is_prime(x):
        x += 1
    return x


class ChainedHashTable:
    def __init__(self, m=7, max_load=1.0):
        self.table = [None] * m                         # table[b] = head of chain b
        self.n = 0                                      # number of keys stored
        self.max_load = max_load                        # rehash when n/m exceeds it

    def bucket_of(self, key):
        return hash_poly(key) % len(self.table)
```

Kode 15.5 (Lanjutan) *chaining.py: tabel hash berantai dalam Python.*

```
    def put(self, key, value):
        """Insert key with value, or replace the value if key is present."""
        b = self.bucket_of(key)
        p = self.table[b]
        while p is not None:
            if p.key == key:                          # present: update in place
                p.value = value
                return
            p = p.next
        self.table[b] = Node(key, value, self.table[b])   # absent: front insertion
        self.n += 1
        if self.n / len(self.table) > self.max_load:
            self._rehash(next_prime(2 * len(self.table)))

    def get(self, key):
        """Return (value or None, number of keys compared)."""
        compared = 0
        p = self.table[self.bucket_of(key)]
        while p is not None:
            compared += 1
            if p.key == key:
                return p.value, compared
            p = p.next
        return None, compared

    def erase(self, key):                             # deleteKey of Section 7.6.5
        b = self.bucket_of(key)
        prev, cur = None, self.table[b]
        while cur is not None and cur.key != key:
            prev, cur = cur, cur.next
        if cur is None:                               # not present
            return False
        if prev is None:                              # first node of the chain
            self.table[b] = cur.next
        else:
            prev.next = cur.next
        self.n -= 1
        return True
```

Kode 15.5 (Lanjutan) *chaining.py: tabel hash berantai dalam Python.*

```
    def show(self):
        empty = longest = 0
        for b, head in enumerate(self.table):
            parts, p = [], head
            while p is not None:
                parts.append(" %s=%s ->" % (p.key, p.value))
                p = p.next
            tail = " NULL" if parts else " (empty)"
            print("  %2d:" % b + "".join(parts) + tail)
            if not parts:
                empty += 1
            longest = max(longest, len(parts))
        m = len(self.table)
        print("  n = %d, m = %d, load factor = %.2f, longest chain = %d, "
              "empty buckets = %d" % (self.n, m, self.n / m, longest, empty))

    def _rehash(self, new_m):
        """Move every node into a new list of new_m buckets, relinking."""
        print("  -- load factor %.2f > %.2f: rehash from %d to %d buckets"
              % (self.n / len(self.table), self.max_load, len(self.table), new_m))
        old, self.table = self.table, [None] * new_m
        for head in old:
            while head is not None:
                moving, head = head, head.next
                b = self.bucket_of(moving.key)
                moving.next = self.table[b]
                self.table[b] = moving


def lookup(t, title):
    code, compared = t.get(title)
    print('  get("%s") -> %s, bucket %d, %d key(s) compared'
          % (title, code if code is not None else "not found", t.bucket_of(title),
             compared))


def main():
    title = ["Laskar Pelangi", "Bumi Manusia", "Filosofi Kopi", "Sejarah Nusantara",
             "Algoritma & Struktur Data", "Cerita Rakyat Nusantara", "Fisika Dasar",
             "Negeri 5 Menara"]
    t = ChainedHashTable(7, 1.0)
```

Kode 15.5 (Lanjutan) *chaining.py: tabel hash berantai dalam Python.*

```
    print("1. Seven titles into m = 7 buckets, h(k) = poly31(k) mod 7")
    for i in range(7):
        code = "B100%d" % (i + 1)
        print('  put("%s", %s) -> bucket %d' % (title[i], code,
                                                t.bucket_of(title[i])))
        t.put(title[i], code)
    t.show()

    print("\n2. Searching")
    for x in ("Filosofi Kopi", "Bumi Manusia", "Ayat-Ayat Cinta", "Pulang"):
        lookup(t, x)

    print("\n3. Updating and deleting")
    t.put("Laskar Pelangi", "B1001+B1011")          # a second copy arrived
    lookup(t, "Laskar Pelangi")
    print('  erase("Algoritma & Struktur Data") -> '
          + ("deleted" if t.erase("Algoritma & Struktur Data") else "not found"))
    gone = t.erase("Pulang")
    print('  erase("Pulang") -> ' + ("deleted" if gone else "not found"))
    t.show()

    print("\n4. Two more titles: the eighth key pushes n/m above 1")
    t.put("Algoritma & Struktur Data", "B1005")
    t.put("Negeri 5 Menara", "B1008")
    t.show()
    lookup(t, "Bumi Manusia")


if __name__ == "__main__":
    main()
```

Di Java dan Python simpul yang dihapus sekadar menjadi tak terjangkau dan pengumpul sampah (*garbage collector*) mengambilnya kembali; di C++ `erase` harus melakukan `delete`, dan destruktor harus membebaskan sisanya. Valgrind memastikan bahwa **Kode 15.3** melepaskan ke-25 blok yang dialokasikannya, termasuk *string* di dalam simpul dan kedua larik *bucket*:

```
==32382== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==32382== Command: /tmp/ch_g
==32382==
==32382==
==32382== HEAP SUMMARY:
==32382==     in use at exit: 0 bytes in 0 blocks
==32382==   total heap usage: 25 allocs, 25 frees, 78,946 bytes allocated
==32382==
==32382== All heap blocks were freed -- no leaks are possible
==32382==
==32382== For lists of detected and suppressed errors, rerun with: -s
==32382== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

15.5.4 Seberapa panjang rantainya?

Di bawah hashing seragam sederhana setiap dari n kunci jatuh di sebuah *bucket* tertentu dengan peluang $1/m$, sehingga panjang harapan rantai mana pun adalah $n \cdot (1/m) = \alpha$, faktor muat. Pencarian yang *gagal* menghitung $h(k)$ dalam waktu $O(1)$ lalu menyusuri seluruh rantai, yang dipilih secara seragam oleh kunci acak yang tidak ada:

$$E[\text{perbandingan, gagal}] = \alpha, \qquad \text{waktu } \Theta(1+\alpha). \tag{15.11}$$

Pencarian yang *berhasil* untuk kunci x membandingkan dengan x sendiri dan dengan setiap kunci di depannya. Dengan penyisipan di depan, kunci-kunci di depan x adalah kunci yang disisipkan

sesudah x ke *bucket* yang sama. Bila x adalah kunci ke-i yang disisipkan, $n-i$ kunci datang sesudahnya, dan masing-masing jatuh di *bucket* x dengan peluang $1/m$. Dengan merata-ratakan atas n kunci,

$$E[\text{perbandingan, berhasil}] = \frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{n-i}{m}\right) = 1+\frac{n-1}{2m} \approx 1+\frac{\alpha}{2}. \tag{15.12}$$

(Jumlah $\sum_{i=1}^{n}(n-i) = 0+1+\cdots+(n-1) = n(n-1)/2$ adalah jumlah yang menghitung perbandingan *bubble sort* pada Subbab 3.3.5.) Kedua ongkos hanya bergantung pada α, tidak pada n. Bila tabel menjaga α di bawah suatu konstanta, misalnya 1, dengan memperbesar m seiring bertambahnya n, setiap operasi rata-rata memerlukan waktu $O(1)$. Subbab 15.7 mengukur Persamaan (15.11) dan Persamaan (15.12) pada 16.384 *bucket* dan mendapati keduanya tepat hingga dua desimal.

15.6 Pengalamatan Terbuka

15.6.1 Kunci yang berpindah, bukan rantai yang memanjang

Chaining menyimpan kunci-kunci yang bertabrakan di *luar* tabel, di dalam simpul. *Pengalamatan terbuka* (*open addressing*) menyimpan setiap kunci di *dalam* larik tabel itu sendiri, satu kunci per slot, dan bila slot asal sebuah kunci sudah terisi, kunci itu mencoba slot-slot lain dalam urutan tetap hingga menemukan slot bebas. Tidak ada pointer, tidak ada simpul, dan tidak ada alokasi per kunci, sehingga memori lebih hemat dan data tersimpan dalam satu blok yang ramah *cache*. Harganya, tabel dapat penuh: selalu $n \le m$, jadi $\alpha \le 1$.

Urutan slot yang dicoba sebuah kunci adalah *barisan probe*-nya (*probe sequence*). Kita memperluas fungsi hash dengan argumen kedua, nomor percobaan $i = 0,1,2,\ldots$:

$$h(k,0),\ h(k,1),\ \ldots,\ h(k,m-1), \tag{15.13}$$

dan mensyaratkan bahwa untuk setiap kunci barisan ini mengunjungi setiap slot dari m slot tepat satu kali, sehingga slot bebas selalu ditemukan bila ada. Penyisipan menyusuri barisan itu hingga menemukan slot kosong. Pencarian menyusuri barisan yang *sama* hingga menemukan kuncinya atau mencapai slot kosong: slot kosong membuktikan kunci itu tidak ada, karena penyisipan pasti berhenti di sana. Tiga pilihan klasik untuk Persamaan (15.13) berbeda dalam cara melangkah dari satu *probe* ke *probe* berikutnya. Jejak pada **Kode 15.6** menyisipkan delapan kunci yang sama, 1003, 1019, 1035, 1012, 1028, 1013, 1044, 1051, ke tabel berisi $m = 16$ slot dengan masing-masing cara. Slot asalnya $h_1(k) = k \bmod 16$ adalah 11, 11, 11, 4, 4, 5, 4, 11: empat kunci menginginkan slot 11 dan tiga menginginkan slot 4, contoh yang sengaja dibuat padat.

15.6.2 *Linear probing*

Barisan *probe* yang paling sederhana mencoba slot berikutnya, lalu berikutnya lagi, dan membungkus kembali ke awal di ujung tabel:

$$h(k,i) = (h_1(k)+i) \bmod m. \tag{15.14}$$

Pada contoh kita 1003 mengambil slot 11, 1019 mendapati 11 terisi dan mengambil 12, 1035 mencoba 11 dan 12 lalu mengambil 13, dan 1051 kemudian memerlukan empat *probe* untuk mencapai 14. Kunci-kunci yang slot asalnya *berbeda* pun mulai saling mengganggu: 1013

menginginkan slot 5, yang sudah diambil 1028 ketika 1028 tergusur dari slot 4, sehingga 1013 pergi ke 6; lalu 1044, yang asalnya 4, harus menyusuri 4, 5, 6, dan 7. Slot-slot terisi bergabung menjadi deretan panjang, dan setiap kunci yang di-hash ke mana pun di dalam sebuah deretan harus berjalan ke ujungnya, sehingga deretan itu makin panjang. Inilah *primary clustering* (penggerombolan primer). Akibatnya *linear probing* cepat memburuk ketika tabel makin penuh (Subbab 15.7 mengukur 123 *probe* per pencarian gagal pada $\alpha = 0.95$), namun pada muatan sedang *linear probing* sering merupakan metode tercepat dalam praktik, karena *probe*-nya menyentuh memori berurutan yang sudah dimuat prosesor ke dalam *cache*-nya.

15.6.3 *Quadratic probing*

Quadratic probing melompat makin jauh pada setiap percobaan, sehingga kunci-kunci yang tergusur dari satu slot padat berpencar alih-alih menumpuk di sebelahnya. Dengan tabel berisi $m = 2^p$ slot, pilihan yang praktis memakai bilangan segitiga 0,1,3,6,10, … sebagai jarak:

$$h(k, i) = \left(h_1(k) + \frac{i(i+1)}{2}\right) \bmod m. \tag{15.15}$$

Untuk 1051, dengan slot asal 11, jarak 0, 1, 3, 6 memberikan slot 11, 12, 14, dan 17 mod 16 = 1. Apakah barisan ini sungguh mengunjungi setiap slot? Andaikan percobaan i dan j, dengan $0 \le i < j < m$, mengenai slot yang sama. Maka $m = 2^p$ membagi $\frac{j(j+1)}{2} - \frac{i(i+1)}{2} = \frac{(j-i)(j+i+1)}{2}$, jadi 2^{p+1} membagi $(j-i)(j+i+1)$. Kedua faktor berselisih $2i+1$, bilangan ganjil, sehingga salah satunya ganjil dan seluruh pangkat 2^{p+1} harus membagi faktor yang lain. Namun $0 < j - i < 2^p$ dan $0 < j + i + 1 < 2^{p+1}$, sehingga tidak satu pun habis dibagi 2^{p+1}, suatu kontradiksi. Jadi m percobaan pertama mengunjungi m slot yang berbeda, setiap slot sekali. (Dengan kuadrat biasa i^2 dan m prima, hanya sekitar separuh slot yang dijamin tercapai, dan karena itulah tabel yang memakai kuadrat harus tetap kurang dari setengah penuh.)

Quadratic probing menghilangkan *primary clustering* tetapi menyisakan *secondary clustering* (penggerombolan sekunder) yang lebih ringan: kunci-kunci dengan slot asal yang sama tetap mengikuti barisan yang sama. Pada jejak itu, 1003, 1019, 1035, dan 1051 semuanya bermula di 11 dan mengikuti 11, 12, 14, 1 satu demi satu.

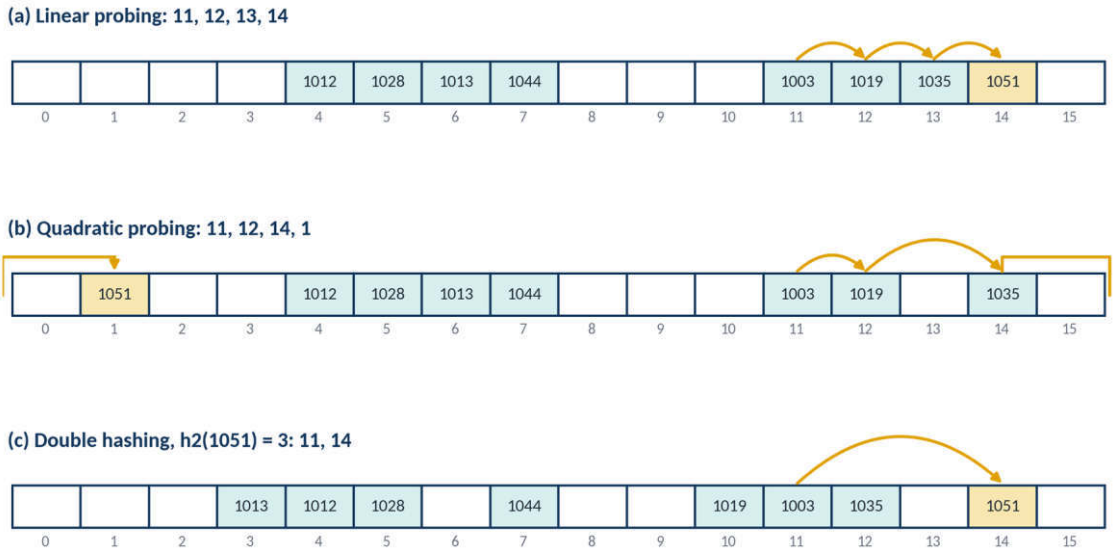
15.6.4 *Double hashing*

Double hashing menghilangkan penggerombolan terakhir itu dengan membuat *langkah*-nya bergantung pada kunci, melalui fungsi hash kedua h_2:

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m. \tag{15.16}$$

Barisan itu mengunjungi setiap slot tepat ketika $h_2(k)$ dan m tidak memiliki faktor bersama, menurut argumen Subbab 15.3.2 dengan langkah $d = h_2(k)$. Dengan $m = 2^p$ cukuplah membuat $h_2(k)$ ganjil; program-program kita memakai $h_2(k) = 1 + 2 \cdot (\lfloor k/m \rfloor \bmod (m/2))$, yang memakai bit-bit kunci yang diabaikan h_1. Untuk 1051, $\lfloor 1051/16 \rfloor = 65$, $65 \bmod 8 = 1$, jadi $h_2(1051) = 3$ dan *probe*-nya 11,14,17 mod 16 = 1, …; slot 14 bebas, dan 1051 tersimpan sesudah dua *probe*. Kunci-kunci yang berbagi slot asal kini memiliki langkah berbeda dan langsung berpisah: 1019 melangkah dengan $h_2 = 15$, yaitu -1 modulo 16, dan pergi ke slot 10; 1035 melangkah dengan 1 dan pergi ke slot 12. Dengan h_2 yang teraduk baik, *double hashing* berperilaku hampir seperti *uniform hashing* (hashing seragam) yang ideal, yang setiap barisan *probe*-nya berpeluang sama untuk menjadi salah satu dari $m!$ urutan slot.

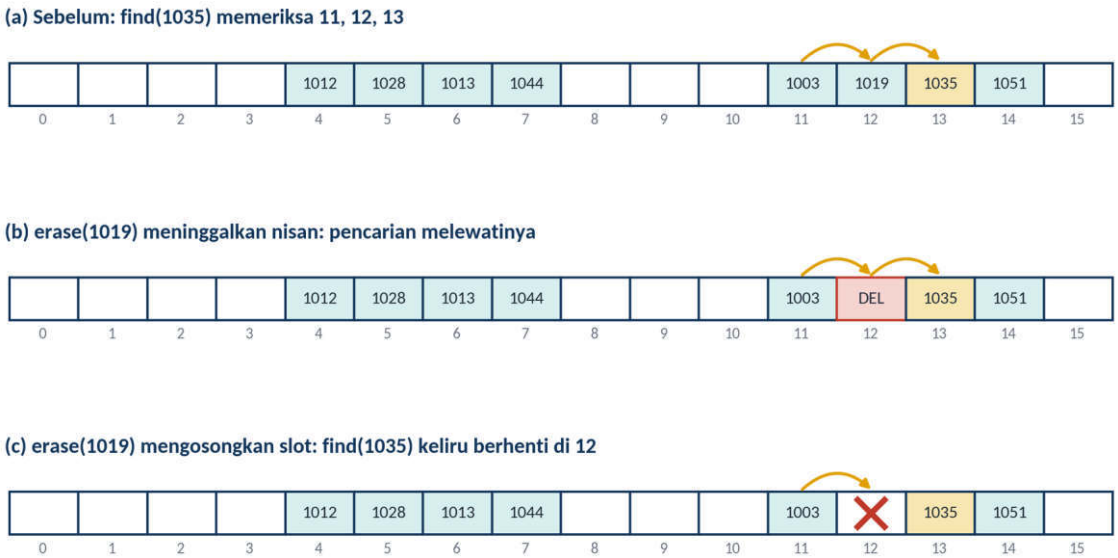
Gambar 15.4 memperlihatkan ketiga tabel sesudah delapan penyisipan, dengan lintasan *probe* 1051 digambar pada masing-masing.



Gambar 15.4 Delapan kunci yang sama dalam 16 slot dengan (a) *linear probing*, (b) *quadratic probing*, dan (c) *double hashing*. Panah-panahnya adalah *probe* yang dilakukan untuk menyisipkan (dan kelak menemukan) 1051: empat, empat, dan dua.

15.6.5 Penghapusan dan nisan

Menghapus dari tabel berpengalamatan terbuka lebih sulit daripada kelihatannya. Misalkan kita menghapus 1019 dari tabel *linear probing* dengan sekadar menandai slot 12 kosong. Pencarian 1035 berikutnya bermula dari slot asalnya 11, menemukan 1003, pindah ke 12, mendapatinya kosong, dan menyimpulkan bahwa 1035 tidak ada, padahal 1035 berada satu slot lebih jauh, di 13. Slot kosong itu telah memotong barisan *probe* yang dilalui 1035 ketika disisipkan. **Gambar 15.5** menggambarkan kegagalan persis ini, yang direproduksi bagian 4 keluaran **Kode 15.6** ("find 1035: not found, 2 probe(s)").



Gambar 15.5 Menghapus 1019 dari tabel *linear probing*. (a) Sebelumnya, pencarian 1035 memeriksa 11, 12, 13. (b) Dengan nisan di slot 12 pencarian melewatinya dan tetap menemukan 1035. (c) Bila slot 12 sekadar dikosongkan, pencarian berhenti di sana dan keliru melaporkan 1035 tidak ada.

Obat bakunya adalah *nisan* (*tombstone*): penanda khusus, DELETED, yang berarti "dulu ada kunci yang tinggal di sini". Pencarian memperlakukan nisan sebagai slot terisi dan melewatinya; penyisipan memperlakukannya sebagai slot bebas dan boleh memakainya kembali. **Pseudokode 15.3** memberikan ketiga operasi dengan tiga keadaan slot EMPTY, OCCUPIED, dan DELETED.

Pseudokode 15.3 Pencarian, penyisipan, dan penghapusan dengan pengalamatan terbuka dan nisan.

```
OPEN-SEARCH(T, k)
1  for i = 0 to m - 1
2      j = h(k, i)
3      if T[j].state == EMPTY
4          return NIL                // barisan probe berakhir: k tidak ada
5      if T[j].state == OCCUPIED and T[j].key == k
6          return j
7  return NIL                        // slot DELETED dilewati

OPEN-INSERT(T, k)
1  tomb = NIL
2  for i = 0 to m - 1
3      j = h(k, i)
4      if T[j].state == OCCUPIED and T[j].key == k
5          return j                  // sudah ada
6      if T[j].state == DELETED and tomb == NIL
7          tomb = j                  // ingat slot pertama yang dapat dipakai ulang
8      if T[j].state == EMPTY
9          if tomb != NIL then j = tomb
10         T[j].key = k; T[j].state = OCCUPIED
11         return j
12 error "tabel penuh"

OPEN-DELETE(T, k)
1  j = OPEN-SEARCH(T, k)
2  if j != NIL
3      T[j].state = DELETED          // tidak pernah EMPTY
```

Perhatikan baris 8 `OPEN-INSERT`. Bahkan sesudah menemukan nisan, penyisipan terus berjalan hingga mencapai slot EMPTY, karena baru saat itulah ia tahu bahwa k tidak tersimpan lebih jauh; berhenti di nisan dapat menyimpan salinan kedua k. Baris-baris terakhir keluaran memperlihatkan hal ini: menyisipkan 1067 menyusuri 11, 12, 13, 14, 15, lima *probe*, lalu menyimpan 1067 di nisan pada slot 12.

Nisan ada ongkosnya. Nisan tidak pernah memperpendek pencarian, sehingga tabel dengan banyak penghapusan melambat bahkan ketika isinya sedikit. Implementasi nyata menghitung nisan bersama kunci ketika menghitung muatan, dan membangun ulang tabel (*rehash* ke ukuran yang sama) ketika nisan menumpuk. *Linear probing* juga memungkinkan penghapusan tanpa nisan, *backward-shift deletion* (penghapusan dengan geser mundur), yang memindahkan kunci-kunci berikutnya dalam deretan ke celah itu; Soal Latihan 15.11 meminta Anda memikirkannya.

15.6.6 Programnya

Kode 15.6 mengimplementasikan ketiga barisan *probe* Persamaan (15.14), Persamaan (15.15), dan Persamaan (15.16) dalam satu kelas yang diparameterkan oleh modenya, dengan nisan dari **Pseudokode 15.3**. Anggota `verbose_` mencetak setiap barisan *probe* saat terjadi.

Kode 15.6 `open_addressing.cpp`: *linear probing*, *quadratic probing*, dan *double hashing* dalam tabel berisi 16 slot, dengan penghapusan melalui nisan.

```
// open_addressing.cpp -- open addressing with linear probing, quadratic probing
// and double hashing (Section 15.6), with tombstones for deletion.
#include <cstdio>
#include <vector>

enum Probe { LINEAR, QUADRATIC, DOUBLE };
enum State { EMPTY, OCCUPIED, DELETED };             // DELETED = tombstone

class OpenTable {
public:
    OpenTable(int m, Probe mode, bool verbose = true)
        : m_(m), mode_(mode), verbose_(verbose), key_(m, 0), state_(m, EMPTY) {}

    // The i-th slot tried for key k, i = 0, 1, 2, ... (m is a power of 2).
    int slot(long long k, int i) const {
        long long h1 = k % m_;
        switch (mode_) {
        case LINEAR:    return (int)((h1 + i) % m_);
        case QUADRATIC: return (int)((h1 + (long long)i * (i + 1) / 2) % m_);
        default: {
            long long h2 = 1 + 2 * ((k / m_) % (m_ / 2));   // odd, so coprime to m
            return (int)((h1 + i * h2) % m_);
        }
        }
    }

    void setVerbose(bool v) { verbose_ = v; }

    // Insert k; print the slots tried. Returns the number of probes.
    int insert(long long k) {
        int firstTomb = -1;
        if (verbose_) std::printf("  insert %lld:", k);
        for (int i = 0; i < m_; ++i) {
            int s = slot(k, i);
            if (verbose_) std::printf(" %d", s);
            if (state_[s] == OCCUPIED && key_[s] == k) {
                if (verbose_) std::printf("  already present\n");
                return i + 1;
            }
            if (state_[s] == DELETED && firstTomb < 0) firstTomb = s;
            if (state_[s] == EMPTY) {                // k is certainly absent
                int target = firstTomb >= 0 ? firstTomb : s;
                key_[target] = k;
                state_[target] = OCCUPIED;
                if (verbose_)
                    std::printf("  -> slot %d (%d probe%s)\n", target, i + 1,
```

Kode 15.6 (Lanjutan) *open_addressing.cpp: linear probing, quadratic probing, dan double hashing dalam tabel berisi 16 slot, dengan penghapusan melalui nisan.*

```
                                i == 0 ? "" : "s");
                return i + 1;
            }
        }
        if (verbose_) std::printf("  table full\n");
        return m_;
    }

    // Return the slot holding k, or -1; `probes` receives the probes used.
    int find(long long k, int &probes) const {
        probes = 0;
        for (int i = 0; i < m_; ++i) {
            int s = slot(k, i);
            probes = i + 1;
            if (state_[s] == EMPTY) return -1;       // the probe sequence ends here
            if (state_[s] == OCCUPIED && key_[s] == k) return s;
        }                                            // DELETED: keep going
        return -1;
    }

    bool erase(long long k) {                        // leave a tombstone
        int probes, s = find(k, probes);
        if (s < 0) return false;
        state_[s] = DELETED;
        return true;
    }

    bool eraseWrongly(long long k) {                 // just empties the slot: a bug
        int probes, s = find(k, probes);
        if (s < 0) return false;
        state_[s] = EMPTY;
        return true;
    }

    void show() const {
        for (int row = 0; row < m_; row += 8) {
            std::printf("  %2d-%-2d:", row, row + 7);
            for (int s = row; s < row + 8; ++s) {
                if (state_[s] == OCCUPIED) std::printf(" %4lld", key_[s]);
                else std::printf(" %4s", state_[s] == DELETED ? "DEL" : "--");
            }
            std::printf("\n");
        }
    }
```

Kode 15.6 (Lanjutan) *open_addressing.cpp: linear probing, quadratic probing, dan double hashing dalam tabel berisi 16 slot, dengan penghapusan melalui nisan.*

```
private:
    int m_;
    Probe mode_;
    bool verbose_;                                  // print each probe sequence
    std::vector<long long> key_;
    std::vector<State> state_;
};

void search(const OpenTable &t, long long k) {
    int probes, s = t.find(k, probes);
    if (s >= 0) std::printf("  find %lld: slot %d, %d probe(s)\n", k, s, probes);
    else std::printf("  find %lld: not found, %d probe(s)\n", k, probes);
}

int main() {
    const long long keys[8] = {1003, 1019, 1035, 1012, 1028, 1013, 1044, 1051};
    const char *name[3] = {"Linear probing: h(k,i) = (k + i) mod 16",
                           "Quadratic probing: h(k,i) = (k + i(i+1)/2) mod 16",
                           "Double hashing: h(k,i) = (k + i*h2(k)) mod 16"};
    for (int mode = LINEAR; mode <= DOUBLE; ++mode) {
        OpenTable t(16, (Probe)mode);
        std::printf("%d. %s\n", mode + 1, name[mode]);
        int total = 0;
        for (long long k : keys) total += t.insert(k);
        t.show();
        std::printf("  total probes for 8 insertions: %d\n", total);
        search(t, 1051);
        search(t, 1067);                            // absent, 1067 mod 16 = 11
        std::printf("\n");
    }

    std::printf("4. Deleting from the linear-probing table\n");
    OpenTable a(16, LINEAR, false), b(16, LINEAR, false);
    for (long long k : keys) {
        a.insert(k);
        b.insert(k);
    }
    std::printf("  with a tombstone: erase 1019\n");
    a.erase(1019);
    a.show();
    search(a, 1035);
    std::printf("  by emptying the slot (wrong): erase 1019\n");
    b.eraseWrongly(1019);
    b.show();
    search(b, 1035);
    std::printf("  a new key may reuse the tombstone:\n");
```

Kode 15.6 (Lanjutan) *open_addressing.cpp: linear probing, quadratic probing, dan double hashing dalam tabel berisi 16 slot, dengan penghapusan melalui nisan.*

```
    a.setVerbose(true);
    a.insert(1067);
    a.show();
    return 0;
}
```

```
1. Linear probing: h(k,i) = (k + i) mod 16
  insert 1003: 11  -> slot 11 (1 probe)
  insert 1019: 11 12  -> slot 12 (2 probes)
  insert 1035: 11 12 13  -> slot 13 (3 probes)
  insert 1012: 4  -> slot 4 (1 probe)
  insert 1028: 4 5  -> slot 5 (2 probes)
  insert 1013: 5 6  -> slot 6 (2 probes)
  insert 1044: 4 5 6 7  -> slot 7 (4 probes)
  insert 1051: 11 12 13 14  -> slot 14 (4 probes)
   0-7 :   --   --   --   -- 1012 1028 1013 1044
   8-15:   --   --   -- 1003 1019 1035 1051   --
  total probes for 8 insertions: 19
  find 1051: slot 14, 4 probe(s)
  find 1067: not found, 5 probe(s)

2. Quadratic probing: h(k,i) = (k + i(i+1)/2) mod 16
  insert 1003: 11  -> slot 11 (1 probe)
  insert 1019: 11 12  -> slot 12 (2 probes)
  insert 1035: 11 12 14  -> slot 14 (3 probes)
  insert 1012: 4  -> slot 4 (1 probe)
  insert 1028: 4 5  -> slot 5 (2 probes)
  insert 1013: 5 6  -> slot 6 (2 probes)
  insert 1044: 4 5 7  -> slot 7 (3 probes)
  insert 1051: 11 12 14 1  -> slot 1 (4 probes)
   0-7 :   -- 1051   --   -- 1012 1028 1013 1044
   8-15:   --   --   -- 1003 1019   -- 1035   --
  total probes for 8 insertions: 18
  find 1051: slot 1, 4 probe(s)
  find 1067: not found, 6 probe(s)

3. Double hashing: h(k,i) = (k + i*h2(k)) mod 16
  insert 1003: 11  -> slot 11 (1 probe)
  insert 1019: 11 10  -> slot 10 (2 probes)
  insert 1035: 11 12  -> slot 12 (2 probes)
  insert 1012: 4  -> slot 4 (1 probe)
  insert 1028: 4 5  -> slot 5 (2 probes)
  insert 1013: 5 4 3  -> slot 3 (3 probes)
  insert 1044: 4 7  -> slot 7 (2 probes)
  insert 1051: 11 14  -> slot 14 (2 probes)
   0-7 :   --   --   -- 1013 1012 1028   -- 1044
   8-15:   --   -- 1019 1003 1035   -- 1051   --
  total probes for 8 insertions: 15
  find 1051: slot 14, 2 probe(s)
  find 1067: not found, 2 probe(s)

4. Deleting from the linear-probing table
  with a tombstone: erase 1019
   0-7 :   --   --   --   -- 1012 1028 1013 1044
   8-15:   --   --   -- 1003  DEL 1035 1051   --
  find 1035: slot 13, 3 probe(s)
  by emptying the slot (wrong): erase 1019
   0-7 :   --   --   --   -- 1012 1028 1013 1044
   8-15:   --   --   -- 1003   -- 1035 1051   --
  find 1035: not found, 2 probe(s)
  a new key may reuse the tombstone:
  insert 1067: 11 12 13 14 15  -> slot 12 (5 probes)
   0-7 :   --   --   --   -- 1012 1028 1013 1044
   8-15:   --   --   -- 1003 1067 1035 1051   --
```

Jumlah-jumlahnya meringkas ketiga metode pada contoh padat ini: 19 *probe* untuk menyisipkan delapan kunci dengan *linear probing*, 18 dengan *quadratic probing*, 15 dengan *double hashing*. Kunci 1067 yang tidak ada, yang asalnya slot 11 yang padat, berongkos 5, 6, dan 2 *probe*. Delapan kunci tidak membuktikan apa pun secara statistik, tetapi peringkatnya sama dengan yang

dikonfirmasi Subbab 15.7 pada 16.384 slot. **Kode 15.7** dan **Kode 15.8** mencetak keluaran yang sama.

Kode 15.7 `OpenAddressing.java`: ketiga barisan *probe* dan nisan dalam Java.

```
// OpenAddressing.java -- open addressing with linear probing, quadratic probing
// and double hashing (Section 15.6), with tombstones for deletion.
public class OpenAddressing {
    enum Probe { LINEAR, QUADRATIC, DOUBLE }

    static final int EMPTY = 0, OCCUPIED = 1, DELETED = 2;   // DELETED = tombstone

    static class OpenTable {
        private final int m;
        private final Probe mode;
        private boolean verbose;                    // print each probe sequence
        private final long[] key;
        private final int[] state;

        OpenTable(int m, Probe mode, boolean verbose) {
            this.m = m;
            this.mode = mode;
            this.verbose = verbose;
            key = new long[m];
            state = new int[m];                     // all EMPTY
        }

        void setVerbose(boolean v) { verbose = v; }

        // The i-th slot tried for key k, i = 0, 1, 2, ... (m is a power of 2).
        int slot(long k, int i) {
            long h1 = k % m;
            switch (mode) {
                case LINEAR:
                    return (int) ((h1 + i) % m);
                case QUADRATIC:
                    return (int) ((h1 + (long) i * (i + 1) / 2) % m);
                default:
                    long h2 = 1 + 2 * ((k / m) % (m / 2));   // odd: coprime to m
                    return (int) ((h1 + i * h2) % m);
            }
        }
```

Kode 15.7 (Lanjutan) *OpenAddressing.java: ketiga barisan probe dan nisan dalam Java.*

```
    // Insert k; print the slots tried. Returns the number of probes.
    int insert(long k) {
        int firstTomb = -1;
        StringBuilder sb = new StringBuilder("  insert " + k + ":");
        for (int i = 0; i < m; i++) {
            int s = slot(k, i);
            sb.append(' ').append(s);
            if (state[s] == OCCUPIED && key[s] == k) {
                if (verbose) System.out.println(sb + "  already present");
                return i + 1;
            }
            if (state[s] == DELETED && firstTomb < 0) firstTomb = s;
            if (state[s] == EMPTY) {             // k is certainly absent
                int target = firstTomb >= 0 ? firstTomb : s;
                key[target] = k;
                state[target] = OCCUPIED;
                if (verbose)
                    System.out.printf("%s  -> slot %d (%d probe%s)%n", sb,
                                      target, i + 1, i == 0 ? "" : "s");
                return i + 1;
            }
        }
        if (verbose) System.out.println(sb + "  table full");
        return m;
    }

    // Return the slot holding k, or -1; probes[0] receives the probes used.
    int find(long k, int[] probes) {
        probes[0] = 0;
        for (int i = 0; i < m; i++) {
            int s = slot(k, i);
            probes[0] = i + 1;
            if (state[s] == EMPTY) return -1;    // the probe sequence ends here
            if (state[s] == OCCUPIED && key[s] == k) return s;
        }                                        // DELETED: keep going
        return -1;
    }

    boolean erase(long k) {                      // leave a tombstone
        int s = find(k, new int[1]);
        if (s < 0) return false;
        state[s] = DELETED;
        return true;
    }
```

Kode 15.7 (Lanjutan) *OpenAddressing.java: ketiga barisan probe dan nisan dalam Java.*

```
        boolean eraseWrongly(long k) {              // just empties the slot: a bug
            int s = find(k, new int[1]);
            if (s < 0) return false;
            state[s] = EMPTY;
            return true;
        }

        void show() {
            for (int row = 0; row < m; row += 8) {
                String label = String.format("  %2d-%-2d:", row, row + 7);
                StringBuilder sb = new StringBuilder(label);
                for (int s = row; s < row + 8; s++) {
                    String cell = state[s] == OCCUPIED ? Long.toString(key[s])
                                : state[s] == DELETED ? "DEL" : "--";
                    sb.append(String.format(" %4s", cell));
                }
                System.out.println(sb);
            }
        }
    }

    static void search(OpenTable t, long k) {
        int[] probes = new int[1];
        int s = t.find(k, probes);
        if (s >= 0)
            System.out.printf("  find %d: slot %d, %d probe(s)%n", k, s, probes[0]);
        else
            System.out.printf("  find %d: not found, %d probe(s)%n", k, probes[0]);
    }

    public static void main(String[] args) {
        long[] keys = {1003, 1019, 1035, 1012, 1028, 1013, 1044, 1051};
        String[] name = {"Linear probing: h(k,i) = (k + i) mod 16",
                         "Quadratic probing: h(k,i) = (k + i(i+1)/2) mod 16",
                         "Double hashing: h(k,i) = (k + i*h2(k)) mod 16"};
        Probe[] modes = Probe.values();
        for (int mode = 0; mode < 3; mode++) {
            OpenTable t = new OpenTable(16, modes[mode], true);
            System.out.printf("%d. %s%n", mode + 1, name[mode]);
            int total = 0;
            for (long k : keys) total += t.insert(k);
            t.show();
            System.out.printf("  total probes for 8 insertions: %d%n", total);
            search(t, 1051);
            search(t, 1067);                        // absent, 1067 mod 16 = 11
            System.out.println();
```

Kode 15.7 (Lanjutan) *OpenAddressing.java: ketiga barisan probe dan nisan dalam Java.*

```
        }

        System.out.println("4. Deleting from the linear-probing table");
        OpenTable a = new OpenTable(16, Probe.LINEAR, false);
        OpenTable b = new OpenTable(16, Probe.LINEAR, false);
        for (long k : keys) {
            a.insert(k);
            b.insert(k);
        }
        System.out.println("  with a tombstone: erase 1019");
        a.erase(1019);
        a.show();
        search(a, 1035);
        System.out.println("  by emptying the slot (wrong): erase 1019");
        b.eraseWrongly(1019);
        b.show();
        search(b, 1035);
        System.out.println("  a new key may reuse the tombstone:");
        a.setVerbose(true);
        a.insert(1067);
        a.show();
    }
}
```

Kode 15.8 `open_addressing.py`: ketiga barisan *probe* dan nisan dalam Python; `None` tidak diperlukan karena keadaan slot disimpan dalam list tersendiri.

```
# open_addressing.py -- open addressing with linear probing, quadratic probing
# and double hashing (Section 15.6), with tombstones for deletion.
LINEAR, QUADRATIC, DOUBLE = 0, 1, 2
EMPTY, OCCUPIED, DELETED = 0, 1, 2                    # DELETED = tombstone


class OpenTable:
    def __init__(self, m, mode, verbose=True):
        self.m, self.mode, self.verbose = m, mode, verbose
        self.key = [0] * m
        self.state = [EMPTY] * m

    def slot(self, k, i):
        """The i-th slot tried for key k, i = 0, 1, 2, ... (m is a power of 2)."""
        h1 = k % self.m
        if self.mode == LINEAR:
            return (h1 + i) % self.m
        if self.mode == QUADRATIC:
            return (h1 + i * (i + 1) // 2) % self.m
        h2 = 1 + 2 * ((k // self.m) % (self.m // 2))   # odd, so coprime to m
        return (h1 + i * h2) % self.m

    def insert(self, k):
        """Insert k; print the slots tried. Returns the number of probes."""
        first_tomb = -1
        tried = "  insert %d:" % k
        for i in range(self.m):
            s = self.slot(k, i)
            tried += " %d" % s
            if self.state[s] == OCCUPIED and self.key[s] == k:
                if self.verbose:
                    print(tried + "  already present")
                return i + 1
            if self.state[s] == DELETED and first_tomb < 0:
                first_tomb = s
            if self.state[s] == EMPTY:                # k is certainly absent
                target = first_tomb if first_tomb >= 0 else s
                self.key[target] = k
                self.state[target] = OCCUPIED
                if self.verbose:
                    print("%s  -> slot %d (%d probe%s)"
                          % (tried, target, i + 1, "" if i == 0 else "s"))
                return i + 1
        if self.verbose:
            print(tried + "  table full")
        return self.m
```

Kode 15.8 (Lanjutan) *open_addressing.py: ketiga barisan probe dan nisan dalam Python; None tidak diperlukan karena keadaan slot disimpan dalam list tersendiri.*

```
    def find(self, k):
        """Return (slot holding k or -1, probes used)."""
        probes = 0
        for i in range(self.m):
            s = self.slot(k, i)
            probes = i + 1
            if self.state[s] == EMPTY:              # the probe sequence ends here
                return -1, probes
            if self.state[s] == OCCUPIED and self.key[s] == k:
                return s, probes
        return -1, probes                           # DELETED: we kept going

    def erase(self, k):                             # leave a tombstone
        s, _ = self.find(k)
        if s < 0:
            return False
        self.state[s] = DELETED
        return True

    def erase_wrongly(self, k):                     # just empties the slot: a bug
        s, _ = self.find(k)
        if s < 0:
            return False
        self.state[s] = EMPTY
        return True

    def show(self):
        for row in range(0, self.m, 8):
            cells = []
            for s in range(row, row + 8):
                if self.state[s] == OCCUPIED:
                    cells.append(" %4d" % self.key[s])
                else:
                    mark = "DEL" if self.state[s] == DELETED else "--"
                    cells.append(" %4s" % mark)
            print("  %2d-%-2d:" % (row, row + 7) + "".join(cells))


def search(t, k):
    s, probes = t.find(k)
    if s >= 0:
        print("  find %d: slot %d, %d probe(s)" % (k, s, probes))
    else:
        print("  find %d: not found, %d probe(s)" % (k, probes))
```

Kode 15.8 (Lanjutan) *open_addressing.py: ketiga barisan probe dan nisan dalam Python; None tidak diperlukan karena keadaan slot disimpan dalam list tersendiri.*

```
def main():
    keys = [1003, 1019, 1035, 1012, 1028, 1013, 1044, 1051]
    name = ["Linear probing: h(k,i) = (k + i) mod 16",
            "Quadratic probing: h(k,i) = (k + i(i+1)/2) mod 16",
            "Double hashing: h(k,i) = (k + i*h2(k)) mod 16"]
    for mode in (LINEAR, QUADRATIC, DOUBLE):
        t = OpenTable(16, mode)
        print("%d. %s" % (mode + 1, name[mode]))
        total = sum(t.insert(k) for k in keys)
        t.show()
        print("  total probes for 8 insertions: %d" % total)
        search(t, 1051)
        search(t, 1067)                              # absent, 1067 mod 16 = 11
        print()

    print("4. Deleting from the linear-probing table")
    a, b = OpenTable(16, LINEAR, False), OpenTable(16, LINEAR, False)
    for k in keys:
        a.insert(k)
        b.insert(k)
    print("  with a tombstone: erase 1019")
    a.erase(1019)
    a.show()
    search(a, 1035)
    print("  by emptying the slot (wrong): erase 1019")
    b.erase_wrongly(1019)
    b.show()
    search(b, 1035)
    print("  a new key may reuse the tombstone:")
    a.verbose = True
    a.insert(1067)
    a.show()


if __name__ == "__main__":
    main()
```

15.6.7 *Chaining* atau pengalamatan terbuka?

Tabel 15.3 membandingkan kedua keluarga. Tidak ada yang menang di mana-mana, dan keduanya muncul dalam pustaka baku: `std::unordered_map` milik C++ dan `HashMap` milik Java memakai *chaining* (Subbab 15.9 dan 15.10), sedangkan `dict` milik Python memakai *probing*.

Tabel 15.3 *Separate chaining* dan pengalamatan terbuka dibandingkan.

	Separate chaining	Pengalamatan terbuka
Tempat kunci yang bertabrakan	di simpul list di luar larik	di slot lain dalam larik
Faktor muat	α berapa pun; kinerja menurun perlahan	$\alpha < 1$ mutlak; jaga paling tinggi sekitar 0,7–0,9
Memori per kunci	kunci, nilai, dan satu pointer, ditambah beban alokator	hanya kunci dan nilai, ditambah slot kosong
Perilaku *cache*	satu lompatan pointer per simpul	slot berurutan (linear) atau berpencar (ganda)
Penghapusan	lepaskan simpulnya, seperti di Bab 7	nisan atau geser mundur
Pointer ke elemen tersimpan	tetap sah ketika tabel tumbuh	berpindah ketika tabel tumbuh

Baris terakhir menjelaskan sebuah keputusan rancangan C++. Standar menjanjikan bahwa referensi ke elemen `unordered_map` tetap sah ketika tabel melakukan *rehash*, dan standar menyediakan antarmuka *bucket* (`bucket_count`, `bucket`, `begin(n)`); keduanya wajar untuk *chaining* dan nyaris mustahil untuk pengalamatan terbuka [4], [5]. Karena itulah tabel berpengalamatan terbuka yang lebih cepat hidup di luar pustaka baku.

15.7 Faktor Muat dan *Rehashing*, Diukur

15.7.1 Satu-satunya bilangan yang penting

Setiap ongkos yang telah kita turunkan bergantung pada tabel hanya melalui faktor muatnya,

$$\alpha = \frac{n}{m}, \tag{15.17}$$

yaitu banyaknya kunci tersimpan per slot. Untuk *chaining*, α adalah panjang rantai rata-rata dan boleh melebihi 1; untuk pengalamatan terbuka, faktor muat adalah pecahan slot yang terisi dan harus tetap di bawah 1. Janji $O(1)$ sebuah tabel hash sebenarnya adalah janji bahwa α tetap dibatasi oleh suatu konstanta, dan menepati janji itu adalah tanggung jawab tabel itu sendiri, karena pemakai hanya pernah menyisipkan.

15.7.2 Apa yang diramalkan teori

Untuk *chaining*, Persamaan (15.11) dan Persamaan (15.12) memberikan α dan $1 + \alpha/2$ perbandingan kunci.

Untuk pengalamatan terbuka kita lebih dulu menganalisis keadaan ideal, *uniform hashing*, yang setiap barisan *probe*-nya berupa urutan acak slot-slot, yaitu perilaku yang dihampiri *double hashing*. Pada pencarian gagal, *probe* pertama mengenai slot terisi dengan peluang $n/m = \alpha$; bila demikian, *probe* kedua mengenai salah satu dari $n-1$ slot terisi yang tersisa di antara $m-1$ slot yang tersisa dengan peluang $(n-1)/(m-1) \le \alpha$; dan seterusnya. Pencarian melakukan *probe* ke-i hanya bila $i-1$ *probe* pertama semuanya mengenai slot terisi, yang terjadi dengan peluang paling besar α^{i-1}. Dengan menjumlahkan peluang dilakukannya setiap *probe*,

$$E[\text{probe, gagal}] \le 1 + \alpha + \alpha^2 + \cdots = \frac{1}{1-\alpha}. \tag{15.18}$$

Pencarian berhasil untuk sebuah kunci mengikuti *probe* yang sama dengan penyisipan yang menyimpannya. Bila kunci itu disisipkan sebagai yang ke-$(i+1)$, tabel ketika itu berisi i kunci, dan menurut Persamaan (15.18) penyisipannya memerlukan paling banyak $1/(1 - i/m)$ *probe*. Dengan merata-ratakan atas n kunci dan mengganti jumlah dengan integral,

$$E[\text{probe, berhasil}] \le \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-i/m} \approx \frac{1}{\alpha}\int_0^{\alpha}\frac{dx}{1-x} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}. \tag{15.19}$$

Linear probing lebih buruk karena *primary clustering*, dan analisisnya adalah salah satu hasil terkenal di bidang ini. Knuth menurunkannya pada tahun 1962, dalam analisis algoritma pertamanya, dan menerbitkannya dalam *The Art of Computer Programming* [1]:

$$E[\text{probe, gagal}] \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right), \tag{15.20}$$

$$E[\text{probe, berhasil}] \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right). \tag{15.21}$$

Kuadrat pada ongkos pencarian gagal adalah *primary clustering* yang dinyatakan sebagai rumus. Pada $\alpha = 0.5$ ia tidak berbahaya, 2,5 *probe* dibandingkan 2 untuk *uniform hashing*; pada $\alpha = 0.9$ ia menjadi 50,5 *probe* dibandingkan 10.

15.7.3 Pengukurannya

Kode 15.9 menguji semua ramalan ini. Program itu mengisi tabel berisi $m = 2^{14} = 16384$ slot dengan kunci acak 62-bit hingga tiga belas faktor muat, mengukur rata-rata *probe* pencarian berhasil (untuk setiap kunci tersimpan) dan pencarian gagal (untuk 10.000 kunci baru), untuk *chaining* dan ketiga barisan *probe* Subbab 15.6, dan mencetak ramalan Persamaan (15.11), Persamaan (15.12), Persamaan (15.18), Persamaan (15.19), Persamaan (15.20), dan Persamaan (15.21) di sampingnya. Program itu lalu memeriksa sebaran panjang rantai, dan akhirnya menumbuhkan tabel berantai dari 8 *bucket* hingga 200.000 kunci sambil menghitung kunci yang dipindahkan oleh *rehashing*.

Kode 15.9 `load_factor.cpp`: *probe* per pencarian terhadap faktor muat, terukur dan teramalkan; sebaran panjang rantai; dan ongkos *rehashing* dengan penggandaan.

```cpp
// load_factor.cpp -- measuring what the load factor does (Section 15.7):
// probes per search for chaining and three kinds of open addressing, the
// distribution of chain lengths, and the cost of rehashing as a table grows.
#include <cmath>
#include <cstdint>
#include <cstdio>
#include <vector>

typedef std::uint64_t u64;

u64 rngState = 7;                                   // SplitMix64, fixed seed
u64 nextKey() {                                     // a random 62-bit key
    u64 z = (rngState += 0x9E3779B97F4A7C15ULL);
    z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9ULL;
    z = (z ^ (z >> 27)) * 0x94D049BB133111EBULL;
    return (z ^ (z >> 31)) >> 2;
}

const int P = 14, M = 1 << P;                       // m = 16384 slots

// ---------- open addressing: 0 = linear, 1 = quadratic, 2 = double hashing
u64 slotOf(u64 k, u64 i, int mode) {
    u64 h1 = k % M;
    if (mode == 0) return (h1 + i) % M;
    if (mode == 1) return (h1 + i * (i + 1) / 2) % M;
    u64 h2 = 1 + 2 * ((k >> P) % (M / 2));          // odd, so coprime to M
    return (h1 + i * h2) % M;
}

// Number of probes to find k (present) or to reach an empty slot (absent).
int probes(const std::vector<u64> &t, const std::vector<char> &used, u64 k,
           int mode) {
    for (u64 i = 0;; ++i) {
        u64 s = slotOf(k, i, mode);
        if (!used[s] || t[s] == k) return (int)i + 1;
    }
}
```

Kode 15.9 (Lanjutan) *load_factor.cpp: probe per pencarian terhadap faktor muat, terukur dan teramalkan; sebaran panjang rantai; dan ongkos rehashing dengan penggandaan.*

```
void measureOpen(int n, int mode, double &succ, double &unsucc) {
    std::vector<u64> t(M, 0), keys(n);
    std::vector<char> used(M, 0);
    for (int j = 0; j < n; ++j) {
        u64 k = keys[j] = nextKey();
        for (u64 i = 0;; ++i) {
            u64 s = slotOf(k, i, mode);
            if (!used[s]) { t[s] = k; used[s] = 1; break; }
        }
    }
    long long sum = 0;
    for (u64 k : keys) sum += probes(t, used, k, mode);
    succ = (double)sum / n;
    sum = 0;
    for (int j = 0; j < 10000; ++j) sum += probes(t, used, nextKey(), mode);
    unsucc = sum / 10000.0;
}

// ---------- separate chaining: count key comparisons
void measureChain(int n, double &succ, double &unsucc) {
    std::vector<std::vector<u64>> b(M);
    std::vector<u64> keys(n);
    for (int j = 0; j < n; ++j) b[(keys[j] = nextKey()) % M].push_back(keys[j]);
    long long sum = 0;
    for (u64 k : keys) {                              // position in its chain
        const std::vector<u64> &c = b[k % M];
        for (std::size_t p = 0; p < c.size(); ++p)
            if (c[p] == k) { sum += (long long)p + 1; break; }
    }
    succ = (double)sum / n;
    sum = 0;
    for (int j = 0; j < 10000; ++j) sum += (long long)b[nextKey() % M].size();
    unsucc = sum / 10000.0;
}

int main() {
    std::printf("1. Average probes (key comparisons for chaining), m = %d\n", M);
    std::printf("          chaining      linear     quadratic     double\n");
    std::printf("  alpha  succ unsucc  succ unsucc  succ unsucc  succ unsucc\n");
    const double alphas[13] = {0.10, 0.25, 0.50, 0.60, 0.70, 0.75, 0.80, 0.85,
                               0.90, 0.95, 1.50, 2.00, 3.00};
    for (double a : alphas) {
        int n = (int)std::lround(a * M);
        double s, u;
        measureChain(n, s, u);
        std::printf("  %4.2f %5.2f %6.2f", a, s, u);
```

Kode 15.9 (Lanjutan) *load_factor.cpp: probe per pencarian terhadap faktor muat, terukur dan teramalkan; sebaran panjang rantai; dan ongkos rehashing dengan penggandaan.*

```
        for (int mode = 0; mode < 3 && a < 1.0; ++mode) {
            measureOpen(n, mode, s, u);
            std::printf(" %5.2f %6.2f", s, u);
        }
        std::printf("\n");
    }

    std::printf("\n2. The same quantities predicted by the formulas of "
                "Section 15.7\n");
    std::printf("              chaining           linear          uniform\n");
    std::printf("  alpha   succ unsucc    succ   unsucc    succ   unsucc\n");
    for (double a : alphas) {
        std::printf("  %5.2f %6.3f %6.3f", a, 1.0 + a / 2.0, a);
        if (a < 1.0)
            std::printf(" %7.3f %8.3f %7.3f %8.3f", 0.5 * (1.0 + 1.0 / (1.0 - a)),
                        0.5 * (1.0 + 1.0 / ((1.0 - a) * (1.0 - a))),
                        std::log(1.0 / (1.0 - a)) / a, 1.0 / (1.0 - a));
        std::printf("\n");
    }

    std::printf("\n3. Chain lengths for n = m = %d random keys\n", M);
    std::vector<int> len(M, 0);
    for (int j = 0; j < M; ++j) ++len[nextKey() % M];
    std::vector<int> hist(9, 0);
    int longest = 0;
    for (int L : len) {
        ++hist[L < 8 ? L : 8];
        if (L > longest) longest = L;
    }
    std::printf("  length  buckets  Poisson(1) prediction\n");
    double fact = 1.0, tail = M;
    for (int L = 0; L <= 8; ++L) {
        if (L > 0) fact *= L;
        double expect = M * std::exp(-1.0) / fact;
        if (L < 8) {
            tail -= expect;
            std::printf("  %6d %8d %12.1f\n", L, hist[L], expect);
        } else {
            std::printf("  %5d+ %8d %12.1f\n", L, hist[L], tail);
        }
    }
    std::printf("  longest chain: %d\n", longest);
```

Kode 15.9 (Lanjutan) *load_factor.cpp: probe per pencarian terhadap faktor muat, terukur dan teramalkan; sebaran panjang rantai; dan ongkos rehashing dengan penggandaan.*

```
    std::printf("\n4. Growing a chained table from m = 8, doubling when "
                "n/m > 0.75\n");
    std::vector<std::vector<u64>> b(8);
    long long n = 0, moved = 0;
    int rehashes = 0;
    for (int j = 0; j < 200000; ++j) {
        u64 k = nextKey();
        b[k % b.size()].push_back(k);
        ++n;
        if ((double)n / b.size() > 0.75) {           // rehash: every key moves
            std::vector<std::vector<u64>> nb(2 * b.size());
            for (const std::vector<u64> &chain : b)
                for (u64 x : chain) {
                    nb[x % nb.size()].push_back(x);
                    ++moved;
                }
            b.swap(nb);
            ++rehashes;
        }
    }
    std::size_t maxChain = 0;
    for (const std::vector<u64> &chain : b)
        if (chain.size() > maxChain) maxChain = chain.size();
    std::printf("  keys inserted: %lld, rehashes: %d, final m = %zu, load = %.3f\n",
                n, rehashes, b.size(), (double)n / b.size());
    std::printf("  keys moved by all rehashes: %lld (%.3f per insertion)\n", moved,
                (double)moved / n);
    std::printf("  longest chain at the end: %zu\n", maxChain);
    return 0;
}
```

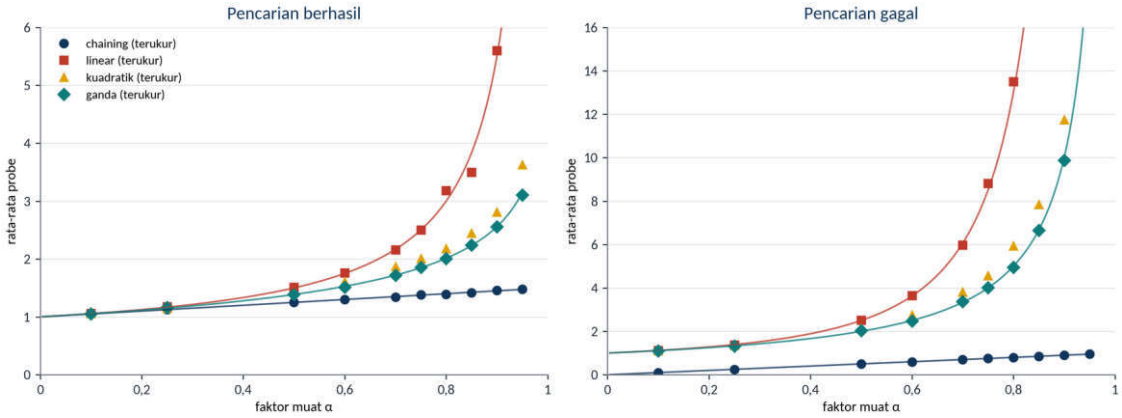
```
1. Average probes (key comparisons for chaining), m = 16384
         chaining      linear     quadratic     double
  alpha  succ unsucc  succ unsucc  succ unsucc  succ unsucc
  0.10  1.05   0.10  1.06   1.12  1.06   1.12  1.05   1.11
  0.25  1.13   0.24  1.18   1.38  1.15   1.36  1.16   1.33
  0.50  1.25   0.50  1.51   2.52  1.43   2.15  1.39   2.03
  0.60  1.30   0.60  1.76   3.65  1.61   2.76  1.51   2.48
  0.70  1.34   0.70  2.16   5.97  1.87   3.82  1.72   3.37
  0.75  1.38   0.76  2.50   8.81  2.01   4.56  1.86   4.01
  0.80  1.39   0.80  3.18  13.51  2.18   5.94  2.01   4.95
  0.85  1.42   0.85  3.50  20.41  2.45   7.85  2.24   6.66
  0.90  1.46   0.90  5.60  50.07  2.81  11.74  2.56   9.87
  0.95  1.48   0.95  8.45 123.04  3.63  23.92  3.11  19.93
  1.50  1.74   1.49
  2.00  2.02   2.02
  3.00  2.50   3.02

2. The same quantities predicted by the formulas of Section 15.7
              chaining           linear           uniform
  alpha   succ unsucc    succ   unsucc    succ   unsucc
   0.10  1.050  0.100   1.056    1.117   1.054    1.111
   0.25  1.125  0.250   1.167    1.389   1.151    1.333
   0.50  1.250  0.500   1.500    2.500   1.386    2.000
   0.60  1.300  0.600   1.750    3.625   1.527    2.500
   0.70  1.350  0.700   2.167    6.056   1.720    3.333
   0.75  1.375  0.750   2.500    8.500   1.848    4.000
   0.80  1.400  0.800   3.000   13.000   2.012    5.000
   0.85  1.425  0.850   3.833   22.722   2.232    6.667
   0.90  1.450  0.900   5.500   50.500   2.558   10.000
   0.95  1.475  0.950  10.500  200.500   3.153   20.000
   1.50  1.750  1.500
   2.00  2.000  2.000
   3.00  2.500  3.000

3. Chain lengths for n = m = 16384 random keys
  length  buckets  Poisson(1) prediction
       0     6108       6027.3
       1     5924       6027.3
       2     3008       3013.7
       3     1002       1004.6
       4      282        251.1
       5       51         50.2
       6        8          8.4
       7        1          1.2
      8+        0          0.2
  longest chain: 7

4. Growing a chained table from m = 8, doubling when n/m > 0.75
  keys inserted: 200000, rehashes: 16, final m = 524288, load = 0.381
  keys moved by all rehashes: 393226 (1.966 per insertion)
  longest chain at the end: 6
```

Gambar 15.6 menggambarkan dua tabel pertama: penanda adalah hasil pengukuran, kurva adalah rumusnya.



Gambar 15.6 Rata-rata *probe* per pencarian pada tabel berisi 16.384 slot, diukur oleh **Kode 15.9** (penanda) dan diramalkan oleh Persamaan (15.12), Persamaan (15.11), Persamaan (15.20), Persamaan (15.19), dan Persamaan (15.18) (kurva). *Chaining* tetap datar; *linear probing* meledak lebih dulu; *double hashing* mengikuti kurva *uniform hashing*.

Empat pelajaran tampak pada bilangan-bilangan itu.

- **Teorinya benar.** *Chaining* cocok dengan $1 + \alpha/2$ dan α dengan selisih paling banyak 0,02 pada setiap muatan. *Double hashing* cocok erat dengan rumus *uniform hashing* (1,86 terukur dibandingkan 1,848 teramalkan pada $\alpha = 0.75$; 9,87 dibandingkan 10,0 untuk pencarian gagal pada $\alpha = 0.9$). *Linear probing* cocok dengan Persamaan (15.20) hingga sekitar $\alpha = 0.9$ (50,07 dibandingkan 50,5).
- ***Linear probing* yang hampir penuh tidak dapat diramalkan.** Pada $\alpha = 0.95$ angka terukur 123 *probe* jauh di bawah ramalan 200,5, dan *seed* lain akan memberi angka yang cukup berbeda: rata-ratanya dikuasai oleh segelintir kunci tak ada yang jatuh di awal deretan terpanjang, dan Persamaan (15.20) sendiri adalah limit untuk tabel yang besarnya tak terhingga. Pelajarannya bukan angka persisnya melainkan peringatannya: jangan pernah menjalankan *linear probing* sepenuh itu.
- ***Quadratic probing* berada di tengah.** Ia menghilangkan *primary clustering*, sehingga mengikuti *double hashing* dengan erat hingga sekitar $\alpha = 0.8$, lalu membayar *secondary clustering*.
- ***Chaining* menurun dengan anggun.** Pada $\alpha = 3$, tiga kunci per *bucket*, pencarian berhasil tetap hanya berongkos 2,5 perbandingan. Pengalamatan terbuka bahkan tidak dapat mencapai $\alpha = 1$.

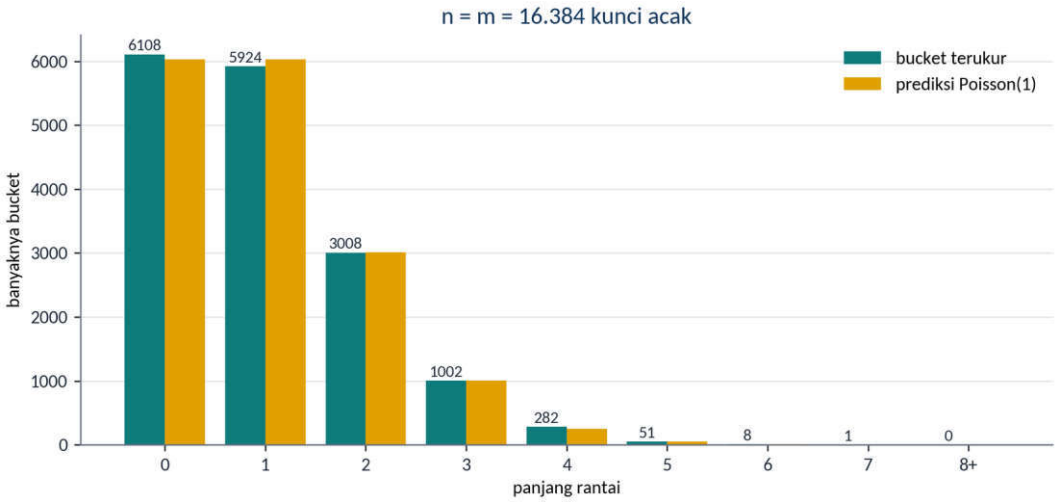
Karena itulah tabel nyata memilih faktor muat maksimum dan melakukan *rehash* ketika mencapainya: 1,0 untuk `std::unordered_map` [5], 0,75 untuk `HashMap` milik Java [6], 2/3 untuk `dict` milik Python [7], dan biasanya 0,5 sampai 0,875 untuk tabel berpengalamatan terbuka, bergantung pada barisan *probe*-nya.

15.7.4 Bagaimana panjang rantai tersebar

Panjang rantai *rata-rata* adalah α, tetapi seberapa sering sebuah *bucket* berisi 5 atau 7 kunci? Di bawah Persamaan (15.1) banyaknya kunci dalam *bucket* tertentu adalah jumlah dari n percobaan bebas yang masing-masing berhasil dengan peluang $1/m$: sebaran binomial. Bila n dan m besar dengan $n/m = \alpha$ tetap, sebaran itu mendekati *sebaran Poisson*:

$$\Pr[\text{sebuah bucket berisi tepat } j \text{ kunci}] \approx \frac{e^{-\alpha}\alpha^{j}}{j!}. \tag{15.22}$$

Untuk $\alpha = 1$ pecahannya $e^{-1} \approx 0.368$ untuk $j = 0$ dan untuk $j = 1$, $e^{-1}/2 \approx 0.184$ untuk $j = 2$, $e^{-1}/6 \approx 0.061$ untuk $j = 3$, dan sesudah itu turun lebih cepat daripada eksponensial. Bagian 3 keluaran menghitung *bucket* tabel dengan $n = m = 16384$ dan **Gambar 15.7** menjajarkannya dengan Persamaan (15.22): 6108 *bucket* kosong dibandingkan 6027 teramalkan, 3008 *bucket* berpanjang 2 dibandingkan 3014, dan rantai terpanjang 7, sedangkan ramalan untuk 8 atau lebih adalah 0,2 *bucket*.



Gambar 15.7 Panjang rantai untuk 16.384 kunci acak dalam 16.384 *bucket* (hijau toska) dibandingkan ramalan Poisson Persamaan (15.22) dengan $\alpha = 1$ (kuning). Lebih dari sepertiga *bucket* kosong; tidak ada yang berisi lebih dari 7 kunci.

Persamaan (15.22) menjelaskan sebuah rincian `HashMap` milik Java yang diuraikan Subbab 15.10: sebuah *bucket* diubah menjadi pohon hanya ketika mencapai 8 entri, karena dengan kode hash acak dan muatan bawaan paling tinggi 0,75, sebuah *bucket* berisi 8 entri berpeluang sekitar satu banding sepuluh juta. Komentar dalam kode sumber JDK mengajukan argumen persis ini [8].

15.7.5 *Rehashing*, dan mengapa penggandaan itu murah

Untuk menjaga α tetap terbatas, tabel yang melampaui muatan maksimumnya mengalokasikan larik yang lebih besar dan menyisipkan ulang setiap kunci, karena slot setiap kunci bergantung pada m melalui $h(k) \bmod m$. Inilah *rehashing*. Satu kali *rehash* berongkos $\Theta(n)$, yang terdengar mencemaskan: apakah kita kembali ke waktu linear?

Tidak, asalkan tabel tumbuh dengan suatu *faktor* tetap. Misalkan tabel berlipat dua setiap kali n melampaui $\alpha_{\max} m$, dimulai dari m_0 *bucket*. *Rehash* terjadi kira-kira pada ukuran $\alpha_{\max} m_0, 2\alpha_{\max} m_0, 4\alpha_{\max} m_0, \ldots$, masing-masing memindahkan semua kunci yang ada, dan yang terakhir terjadi dengan paling banyak n kunci. Dibaca mundur dari *rehash* terakhir, banyaknya kunci yang dipindahkan paling banyak $n, n/2, n/4, \ldots$, sehingga

$$\text{total kunci dipindahkan} \le n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) < 2n. \tag{15.23}$$

Tersebar atas n penyisipan, *rehashing* menambah kurang dari dua pemindahan per penyisipan: $O(1)$ *teramortisasi*, argumen yang sama yang membuat `std::vector::push_back` murah (Bab 1). Bagian 4 keluaran mengukurnya: 200.000 penyisipan, 16 kali *rehash*, 393.226 kunci dipindahkan, 1,966 pemindahan per penyisipan, tepat di bawah batas 2. Hitungan persisnya dapat diperiksa dengan tangan. Dengan $\alpha_{\max} = 0.75$ tabel berisi m *bucket* melakukan *rehash* ketika kunci ke-

$(0.75m + 1)$ tiba, sehingga tabel berukuran 8, 16, …, 262.144 *bucket* memindahkan $0.75(8 + 16 + \cdots + 262144) + 16 = 0.75 \cdot 524280 + 16 = 393226$ kunci.

Tumbuh dengan *jumlah* tetap, misalnya 1000 *bucket* setiap kali, akan menjadi bencana: tabel akan melakukan *rehash* setiap beberapa ratus penyisipan, dan total pekerjaannya menjadi $\Theta(n^2)$ (Soal Latihan 15.13). Dua penyempurnaan praktis lazim dipakai. Tabel yang tidak boleh berhenti lama (gelung permainan, *switch* jaringan) melakukan *rehash* secara *bertahap*, memindahkan beberapa *bucket* pada setiap operasi selama dua larik hidup berdampingan. Dan program yang tahu berapa banyak kunci yang akan datang sebaiknya menyatakannya sejak awal dengan `reserve(n)` di C++ atau argumen kapasitas `new HashMap<>(capacity)` di Java, sehingga tidak terjadi *rehash* sama sekali.

15.8 Kompleksitas, Kasus Terburuk, dan *Hash Flooding*

15.8.1 Tagihan lengkapnya

Tabel 15.4 menyatakan ongkos sebuah tabel hash, dengan anggapan fungsi hash dihitung dalam waktu $O(1)$ (untuk kunci *string* sepanjang L, tambahkan $O(L)$ pada setiap operasi, untuk meng-hash dan membandingkannya).

Tabel 15.4 Ongkos operasi tabel hash dengan faktor muat Persamaan (15.17) yang dijaga di bawah suatu konstanta; bandingkan dengan $\Theta(\lg n)$ pohon pencarian seimbang untuk semuanya.

Operasi	*Chaining*, rata-rata	Pengalamatan terbuka, rata-rata	Kasus terburuk (keduanya)
pencarian berhasil	$1 + \alpha/2$ perbandingan	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$ *probe* (seragam)	$\Theta(n)$
pencarian gagal	α perbandingan	$\frac{1}{1-\alpha}$ *probe* (seragam)	$\Theta(n)$
penyisipan (dengan pemeriksaan duplikat)	seperti pencarian gagal, lalu $O(1)$	seperti pencarian gagal	$\Theta(n)$
penghapusan	seperti pencarian berhasil, lalu $O(1)$	seperti pencarian berhasil	$\Theta(n)$
penyisipan, termasuk *rehashing*	$O(1)$ teramortisasi	$O(1)$ teramortisasi	$\Theta(n)$ untuk satu penyisipan
mendaftar semua kunci terurut	$\Theta(n \lg n)$ (urutkan dulu)	$\Theta(n \lg n)$	$\Theta(n \lg n)$
minimum, penerus, kueri rentang	$\Theta(n + m)$	$\Theta(m)$	$\Theta(n + m)$

Kasus terburuk selalu bergambar sama: setiap kunci di-hash ke slot yang sama. Tabel berantai merosot menjadi satu *linked list* Bab 7, tabel ber-*probing* menjadi satu deretan panjang, dan menyisipkan n kunci berongkos $0 + 1 + \cdots + (n-1) = n(n-1)/2$ perbandingan. Dengan fungsi hash tetap yang dirancang baik dan data biasa, hal ini praktis tidak pernah terjadi secara kebetulan. Bahayanya, hal itu dapat terjadi dengan sengaja.

15.8.2 *Hash flooding*

Bila penyerang mengetahui fungsi hash-nya, penyerang dapat memilih kunci-kunci yang semuanya bertabrakan lalu mengirimkannya ke program yang menyimpannya dalam tabel hash. Scott Crosby dan Dan Wallach menguraikan *serangan kompleksitas algoritmik* (*algorithmic complexity attack*) ini pada tahun 2003, dan memperlambat tabel hash bahasa Perl, *proxy* web Squid, dan sistem deteksi intrusi Bro hingga merangkak dengan masukan rekaan yang tidak

seberapa [9]. Pada Desember 2011 Alexander Klink dan Julian Wälde memperlihatkan bahwa hampir setiap *platform* web masa itu (PHP, ASP.NET, *server* aplikasi Java, Python, Ruby, dan lainnya) menyimpan parameter permintaan HTTP dalam tabel hash dengan fungsi hash publik tanpa kunci rahasia, sehingga satu permintaan rekaan dapat membuat prosesor sebuah *server* sibuk selama bermenit-menit [10]. Serangan itu dikenal sebagai *hash flooding* atau HashDoS.

Membangun kunci-kunci yang bertabrakan ternyata mudah untuk hash polinomial Persamaan (15.5) dengan $B = 31$. *String* dua huruf "Aa" dan "BB" bertabrakan: $65 \cdot 31 + 97 = 2112 = 66 \cdot 31 + 66$. Dan bila dua blok berpanjang sama bertabrakan, maka dua *string* apa pun yang dibangun dari blok-blok itu pada posisi yang sama juga bertabrakan, karena menurut Persamaan (15.5) $h(xy) = h(x) \cdot 31^{|y|} + h(y)$. Dengan 11 blok, masing-masing "Aa" atau "BB", kita memperoleh $2^{11} = 2048$ *string* 22 huruf yang berbeda dengan satu nilai hash tunggal. **Kode 15.10** membangunnya dan menyisipkannya ke tabel berantai berisi 4096 *bucket*, di samping 2048 kunci biasa, mula-mula dengan hash publik lalu dengan hash *berkunci* (*keyed*).

Kode 15.10 `flooding.cpp`: 2048 kunci yang semuanya bertabrakan di bawah hash polinomial publik berbasis 31, dan hash polinomial berkunci dengan basis rahasia yang kembali menyebarkannya.

```cpp
// flooding.cpp -- hash flooding (Section 15.8): keys built so that they all
// collide under a fixed, publicly known hash function, and a keyed (seeded)
// hash function that the same keys cannot defeat.
#include <cstdint>
#include <cstdio>
#include <string>
#include <vector>

// The 32-bit polynomial hash with base 31 (the formula of Java's String).
std::uint32_t hash31(const std::string &s) {
    std::uint32_t h = 0;
    for (unsigned char c : s) h = h * 31 + c;
    return h;
}

// A keyed polynomial hash modulo the prime 2^31 - 1, with a secret base r.
const std::uint64_t PRIME = 2147483647ULL;
std::uint64_t secretBase = 0;
std::uint32_t hashKeyed(const std::string &s) {
    std::uint64_t h = 0;
    for (unsigned char c : s) h = (h * secretBase + c) % PRIME;
    return (std::uint32_t)h;
}

// Insert every key into a chained table of m buckets (checking for duplicates,
// as put must), then search for every key. Report the work done.
void experiment(const char *label, const std::vector<std::string> &keys,
                std::uint32_t (*h)(const std::string &), std::size_t m) {
    std::vector<std::vector<std::string>> table(m);
    long long insertCmp = 0, searchCmp = 0;
    for (const std::string &k : keys) {
        std::vector<std::string> &chain = table[h(k) % m];
        bool present = false;
        for (const std::string &x : chain) {
            ++insertCmp;
            if (x == k) { present = true; break; }
        }
        if (!present) chain.push_back(k);
    }
    for (const std::string &k : keys) {
        for (const std::string &x : table[h(k) % m]) {
            ++searchCmp;
            if (x == k) break;
        }
    }
    std::size_t longest = 0;
```

Kode 15.10 (Lanjutan) *flooding.cpp: 2048 kunci yang semuanya bertabrakan di bawah hash polinomial publik berbasis 31, dan hash polinomial berkunci dengan basis rahasia yang kembali menyebarkannya.*

```
    for (const std::vector<std::string> &c : table)
        if (c.size() > longest) longest = c.size();
    std::printf("  %-28s %8zu %12lld %12lld\n", label, longest, insertCmp,
                searchCmp);
}

int main() {
    std::printf("1. Different strings, equal hash31 values\n");
    const char *pairs[6] = {"Aa", "BB", "AaAa", "AaBB", "BBAa", "BBBB"};
    for (const char *s : pairs)
        std::printf("  hash31(\"%s\") = %u\n", s, hash31(s));

    // 2^11 = 2048 attack keys: every string of 11 blocks, each "Aa" or "BB".
    std::vector<std::string> attack, ordinary;
    for (int mask = 0; mask < 2048; ++mask) {
        std::string s;
        for (int b = 10; b >= 0; --b) s += (mask >> b & 1) ? "BB" : "Aa";
        attack.push_back(s);
    }
    for (int i = 0; i < 2048; ++i) {
        char buf[24];
        std::snprintf(buf, sizeof buf, "reader%04d", i);
        ordinary.push_back(buf);
    }
    std::printf("  first attack keys: %s %s ...\n", attack[0].c_str(),
                attack[1].c_str());
    std::printf("  all 2048 attack keys hash to %u\n", hash31(attack[2047]));

    std::printf("\n2. n = 2048 keys into m = 4096 buckets\n");
    std::printf("  %-28s %8s %12s %12s\n", "keys and hash function", "longest",
                "insert cmp", "search cmp");
    experiment("ordinary keys, hash31", ordinary, hash31, 4096);
    experiment("attack keys, hash31", attack, hash31, 4096);
    secretBase = 1234567891ULL % PRIME;      // in real life: random at start
    experiment("attack keys, keyed hash", attack, hashKeyed, 4096);
    experiment("ordinary keys, keyed hash", ordinary, hashKeyed, 4096);
    return 0;
}
```

```
1. Different strings, equal hash31 values
  hash31("Aa") = 2112
  hash31("BB") = 2112
  hash31("AaAa") = 2031744
  hash31("AaBB") = 2031744
  hash31("BBAa") = 2031744
  hash31("BBBB") = 2031744
  first attack keys: AaAaAaAaAaAaAaAaAaAa AaAaAaAaAaAaAaAaAaBB ...
  all 2048 attack keys hash to 2738481856

2. n = 2048 keys into m = 4096 buckets
  keys and hash function        longest   insert cmp   search cmp
  ordinary keys, hash31               3          935         2983
  attack keys, hash31              2048      2096128      2098176
  attack keys, keyed hash             4          429         2477
  ordinary keys, keyed hash           3          752         2800
```

Kunci-kunci serangan membangun satu rantai berisi 2048 simpul, dan penyisipannya berongkos 2.096.128 perbandingan, tepat $2048 \cdot 2047/2$, dibandingkan 935 untuk kunci biasa sebanyak yang sama: lebih dari dua ribu kali lipat pekerjaan. Dengan sejuta kunci alih-alih dua ribu, rasionya menjadi setengah juta, dan tabel itu memerlukan berjam-jam untuk diisi.

15.8.3 Hashing teracak: pertahanannya

Pertahanannya adalah mengeluarkan fungsi hash dari pengetahuan penyerang. Alih-alih satu fungsi tetap, tabel memakai fungsi yang dipilih secara acak, ketika program mulai berjalan, dari suatu keluarga yang besar. Teori di baliknya adalah *universal hashing* (hashing universal) Carter dan Wegman tahun 1979 [11]:

> **DEFINISI.** Keluarga $\mathcal{H}$ berisi fungsi-fungsi hash dari U ke $\{0, \ldots, m-1\}$ disebut *universal* bila untuk setiap pasangan kunci berbeda $x \neq y$,

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}, \tag{15.24}$$

dengan peluang diambil atas pemilihan acak h dari $\mathcal{H}$. Ini adalah batas tabrakan hashing seragam sederhana, tetapi kini berlaku untuk *setiap* himpunan kunci, termasuk yang dipilih lawan yang mengetahui $\mathcal{H}$ tetapi tidak tahu anggota mana yang terpilih. Semua nilai rata-rata dalam bab ini, yang diturunkan di bawah Persamaan (15.1), lalu berlaku sebagai nilai harapan untuk setiap masukan [12].

Hash berkunci pada **Kode 15.10** adalah keluarga sederhana yang mendekati Persamaan (15.24). Hash itu menghitung nilai polinomial *string* pada suatu titik rahasia r modulo bilangan prima $p = 2^{31} - 1$:

$$h_r(s) = \left(\sum_{i=0}^{L-1} s_i \, r^{L-1-i} \right) \bmod p. \tag{15.25}$$

Untuk dua *string* berbeda x dan y berpanjang sama L, $h_r(x) - h_r(y)$ adalah polinomial tak nol dalam r berderajat paling tinggi $L-1$ atas lapangan bilangan bulat modulo p, dan polinomial semacam itu memiliki paling banyak $L-1$ akar. Jadi rahasia acak r membuat x dan y bertabrakan dengan peluang paling besar $(L-1)/p$: untuk kunci-kunci serangan 22 huruf, sekitar satu banding seratus juta. Kunci-kunci serangan itu dibangun untuk $r = 31$; dengan basis rahasia lain mereka berpencar, dan keluaran memperlihatkan rantai terpanjang 4 dan 429 perbandingan, lebih sedikit daripada untuk kunci biasa.

Sistem produksi memakai fungsi berkunci yang lebih kuat. SipHash, dirancang Jean-Philippe Aumasson dan Daniel J. Bernstein pada tahun 2012 sebagai jawaban langsung atas *hash flooding*, adalah fungsi pseudoacak berkunci yang cepat dengan kunci 128-bit [13]. Python mengadopsinya untuk *string* dan *bytes* pada versi 3.4 (PEP 456 [14]) dan beralih ke varian yang lebih cepat, SipHash-1-3, pada versi 3.11 [15]; kuncinya diambil secara acak setiap kali *interpreter* dimulai, dan karena itulah Subbab 15.10.2 mencetak tiga nilai hash berbeda untuk *string* yang sama. Java menempuh jalan lain, karena `String.hashCode` ditetapkan oleh spesifikasi bahasa dan banyak program bergantung pada nilainya: sejak Java 8 sebuah *bucket* `HashMap` yang mengumpulkan terlalu banyak entri diubah menjadi pohon seimbang, sehingga bahkan banjir kunci yang bertabrakan hanya berongkos $O(\lg n)$ per operasi alih-alih $O(n)$ [16]. Dan *server* web mulai membatasi banyaknya parameter yang boleh dibawa satu permintaan.

> **HASH PUSTAKA TIDAK SELALU BERKUNCI.** `std::hash<int>` milik GCC adalah fungsi identitas (Subbab 15.3.6), dan C++ tidak mewajibkan `std::hash` untuk *string* diacak. Program C++ yang menyimpan kunci pilihan orang asing (parameter permintaan, nama pengguna, nama berkas dari unggahan) dalam `unordered_map` sebaiknya menyediakan fungsi hash berkuncinya sendiri sebagai argumen templat ketiga, persis seperti yang dilakukan Subbab 15.9 untuk kunci dua medan.

15.9 `std::unordered_map` versus `std::map`

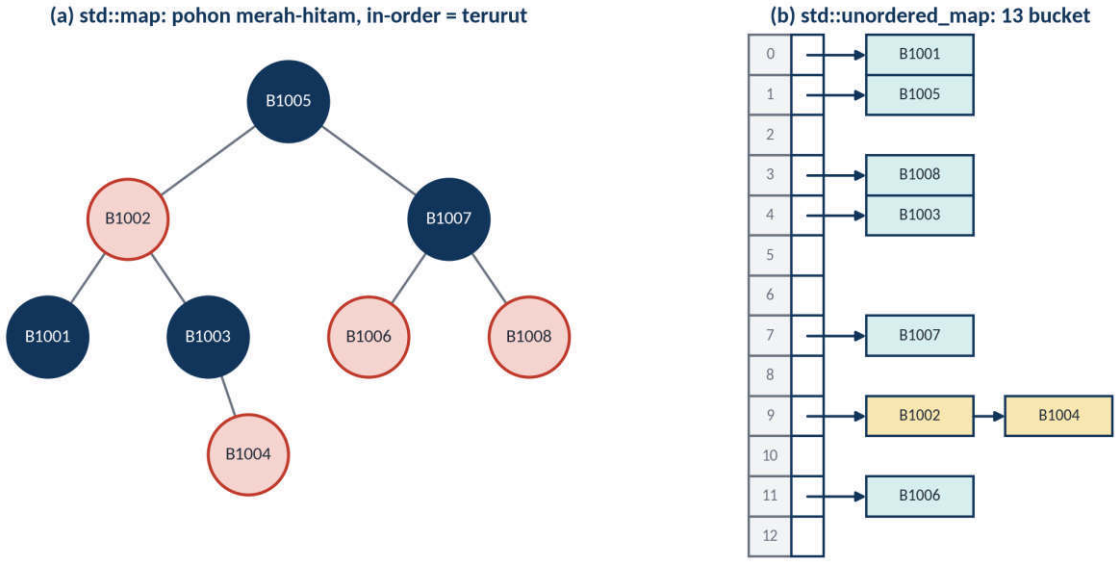
15.9.1 Antarmuka sama, mesin berbeda

C++ menawarkan dua kontainer asosiatif yang dalam kode tampak hampir identik. Keduanya menyimpan pasangan kunci-nilai, keduanya memiliki `operator[]`, `find`, `count`, `insert`, `erase`, dan `at`, dan sebuah program sering dapat berpindah dari yang satu ke yang lain dengan mengubah satu kata. Di dalamnya, keduanya sama sekali tidak memiliki kesamaan.

- `std::map` adalah *pohon pencarian biner seimbang*, dalam setiap implementasi utama berupa pohon merah-hitam (*red-black tree*) dari Subbab 11.9.3. Ia memerlukan kunci yang dapat dibandingkan dengan `<`. Ia menjaga kunci tetap terurut, sehingga iterasi mengunjunginya dari yang terkecil hingga terbesar, dan `lower_bound`, `upper_bound`, serta tetangga *in-order* tersedia. Setiap operasi $O(\lg n)$ pada kasus terburuk.
- `std::unordered_map` adalah *tabel hash dengan separate chaining*: larik *bucket*, masing-masing sebuah rantai simpul. Ia memerlukan fungsi hash (`std::hash<Key>` secara bawaan) dan `==`. Urutan iterasinya tidak ditentukan. Operasinya $O(1)$ rata-rata dan $O(n)$ pada kasus terburuk [4].

Nama-nama itu bersifat historis. `std::map` sudah menjadi bagian standar C++ pertama tahun 1998, ketika *map* berarti kontainer asosiatif apa pun; tabel hash baru ditambahkan pada C++11, dengan nama hati-hati *unordered* agar tidak merusak program-program yang sudah mendefinisikan `hash_map` sendiri. Akibatnya ada jebakan yang pernah dijumpai setiap pemrogram C++: program yang memakai `std::map` memakai *pohon*, bukan tabel hash, apa pun sebutan pernyataan masalahnya. Bab 16 menyusun kembali rencana perjalanan dari setumpuk tiket dengan sebuah *map* dari setiap kota asal ke tujuannya dan sebuah *map* terbalik dari setiap tujuan ke asalnya; ditulis dengan `std::map` program itu benar dan berjalan dalam $O(n \lg n)$, dan mengganti satu kata itu menjadi `std::unordered_map` menjadikannya solusi hashing dengan waktu harapan $O(n)$.

Gambar 15.8 memperlihatkan kedua kontainer sesudah delapan kode buku yang sama disisipkan dalam urutan acak yang sama.



Gambar 15.8 Delapan kode disisipkan dalam urutan B1005, B1001, B1008, B1003, B1002, B1007, B1004, B1006. (a) `std::map` menyimpan pohon merah-hitam, yang penelusuran *in-order*-nya terurut. (b) `std::unordered_map` menyimpan 13 *bucket*; B1002 dan B1004 berbagi *bucket* 9, dan tidak ada urutan yang bertahan.

15.9.2 Programnya

Kode 15.11 membuat perbedaan itu konkret: urutan iterasi, antarmuka *bucket*, pertanyaan yang hanya dapat dijawab pohon, kunci buatan pengguna, dan pengukur waktu.

Kode 15.11 `maps.cpp`: `std::map` versus `std::unordered_map` pada data yang sama, antarmuka *bucket*, kueri rentang, fungsi hash untuk kunci dua medan, dan pengukuran waktu untuk satu juta kunci.

```
// maps.cpp -- std::map (a balanced binary search tree) against
// std::unordered_map (a hash table): ordering, buckets, range queries,
// a user-defined key, and a stopwatch (Section 15.9).
#include <chrono>
#include <cstdint>
#include <cstdio>
#include <functional>
#include <map>
#include <random>
#include <string>
#include <unordered_map>
#include <vector>

struct Edition {                                    // a key made of two fields
    std::string title;
    int year;
    bool operator==(const Edition &o) const {
        return title == o.title && year == o.year;
    }
};

struct EditionHash {                                // how to hash an Edition
    std::size_t operator()(const Edition &e) const {
        std::size_t h = std::hash<std::string>{}(e.title);
        return h ^ (std::hash<int>{}(e.year) + 0x9E3779B97F4A7C15ULL + (h << 6)
                    + (h >> 2));                    // mix the two hashes
    }
};

template <typename F>
double ms(F f) {
    auto t0 = std::chrono::steady_clock::now();
    f();
    auto t1 = std::chrono::steady_clock::now();
    return std::chrono::duration<double, std::milli>(t1 - t0).count();
}
```

Kode 15.11 (Lanjutan) *maps.cpp: std::map versus std::unordered_map pada data yang sama, antarmuka bucket, kueri rentang, fungsi hash untuk kunci dua medan, dan pengukuran waktu untuk satu juta kunci.*

```
int main() {
    const char *code[8] = {"B1005", "B1001", "B1008", "B1003",
                           "B1002", "B1007", "B1004", "B1006"};
    const char *title[8] = {"Algoritma & Struktur Data", "Laskar Pelangi",
                            "Negeri 5 Menara", "Filosofi Kopi", "Bumi Manusia",
                            "Fisika Dasar", "Sejarah Nusantara",
                            "Cerita Rakyat Nusantara"};
    std::map<std::string, std::string> tree;
    std::unordered_map<std::string, std::string> hash;
    for (int i = 0; i < 8; ++i) {
        tree[code[i]] = title[i];
        hash[code[i]] = title[i];
    }

    std::printf("1. Inserted in the order B1005 B1001 B1008 B1003 B1002 B1007 "
                "B1004 B1006\n  std::map          :");
    for (const auto &kv : tree) std::printf(" %s", kv.first.c_str());
    std::printf("\n  std::unordered_map:");
    for (const auto &kv : hash) std::printf(" %s", kv.first.c_str());

    std::printf("\n\n2. Inside the unordered_map\n");
    std::printf("  bucket_count = %zu, load_factor = %.3f, "
                "max_load_factor = %.1f\n", hash.bucket_count(),
                hash.load_factor(), hash.max_load_factor());
    for (int i = 0; i < 8; ++i)
        std::printf("  %s -> bucket %zu%s", code[i], hash.bucket(code[i]),
                    i % 2 == 1 ? "\n" : "    ");
    std::printf("  std::hash<int>{}(42) = %zu\n", std::hash<int>{}(42));
    std::unordered_map<int, int> grow;
    std::size_t last = grow.bucket_count();
    std::printf("  bucket counts while inserting 1..1000:  %zu", last);
    for (int i = 1; i <= 1000; ++i) {
        grow[i] = i;
        if (grow.bucket_count() != last) {
            last = grow.bucket_count();
            std::printf(" %zu", last);
        }
    }
```

Kode 15.11 (Lanjutan) *maps.cpp: std::map versus std::unordered_map pada data yang sama, antarmuka bucket, kueri rentang, fungsi hash untuk kunci dua medan, dan pengukuran waktu untuk satu juta kunci.*

```
    std::printf("\n\n3. Questions only the ordered map can answer\n");
    std::printf("  first and last code: %s, %s\n", tree.begin()->first.c_str(),
                tree.rbegin()->first.c_str());
    std::printf("  codes from B1003 to B1006:");
    for (auto it = tree.lower_bound("B1003"); it != tree.upper_bound("B1006"); ++it)
        std::printf(" %s", it->first.c_str());
    std::printf("\n  first code after B1004: %s\n",
                tree.upper_bound("B1004")->first.c_str());

    std::printf("\n4. A key with two fields needs its own hash function\n");
    std::unordered_map<Edition, int, EditionHash> copies;
    copies[{"Laskar Pelangi", 2005}] = 3;
    copies[{"Laskar Pelangi", 2020}] = 2;
    copies[{"Bumi Manusia", 1980}] = 1;
    std::printf("  copies of Laskar Pelangi (2020): %d\n",
                copies.at({"Laskar Pelangi", 2020}));
    std::printf("  copies of Bumi Manusia (2005) recorded: %zu\n",
                copies.count({"Bumi Manusia", 2005}));

    std::printf("\n5. One million random integer keys (times vary by machine)\n");
    std::mt19937_64 gen(15);
    std::vector<std::uint64_t> keys(1000000);
    for (std::uint64_t &k : keys) k = gen();
    std::map<std::uint64_t, int> m;
    std::unordered_map<std::uint64_t, int> u;
    long long found = 0;
    double mi = ms([&] { for (std::uint64_t k : keys) m[k] = 1; });
    double mf = ms([&] { for (std::uint64_t k : keys) found += m.count(k); });
    double ui = ms([&] { for (std::uint64_t k : keys) u[k] = 1; });
    double uf = ms([&] { for (std::uint64_t k : keys) found += u.count(k); });
    std::printf("  %-20s %10s %10s\n", "", "insert", "find");
    std::printf("  %-20s %7.0f ms %7.0f ms\n", "std::map", mi, mf);
    std::printf("  %-20s %7.0f ms %7.0f ms\n", "std::unordered_map", ui, uf);
    std::printf("  (keys found: %lld)\n", found);
    return 0;
}
```

```
1. Inserted in the order B1005 B1001 B1008 B1003 B1002 B1007 B1004 B1006
  std::map          : B1001 B1002 B1003 B1004 B1005 B1006 B1007 B1008
  std::unordered_map: B1006 B1007 B1004 B1002 B1003 B1008 B1001 B1005

2. Inside the unordered_map
  bucket_count = 13, load_factor = 0.615, max_load_factor = 1.0
  B1005 -> bucket 1     B1001 -> bucket 0
  B1008 -> bucket 3     B1003 -> bucket 4
  B1002 -> bucket 9     B1007 -> bucket 7
  B1004 -> bucket 9     B1006 -> bucket 11
  std::hash<int>{}(42) = 42
  bucket counts while inserting 1..1000:  1 13 29 59 127 257 541 1109

3. Questions only the ordered map can answer
  first and last code: B1001, B1008
  codes from B1003 to B1006: B1003 B1004 B1005 B1006
  first code after B1004: B1005

4. A key with two fields needs its own hash function
  copies of Laskar Pelangi (2020): 2
  copies of Bumi Manusia (2005) recorded: 0

5. One million random integer keys (times vary by machine)
                          insert       find
  std::map                561 ms     621 ms
  std::unordered_map      153 ms      24 ms
  (keys found: 2000000)
```

Bagian 1 adalah perbedaan pokoknya: pohon mengembalikan kode-kode terurut, apa pun urutan kedatangannya; tabel hash mengembalikannya dalam urutan yang bukan urutan terurut dan bukan pula urutan penyisipan, dan yang dapat berubah sesudah *rehash* berikutnya atau dengan *compiler* lain. Bagian 2 membuka tabel hash. Tabel itu dimulai dengan 13 *bucket* untuk 8 kunci, faktor muat $8/13 \approx 0.615$, dan akan melakukan *rehash* ketika muatan melampaui `max_load_factor() = 1.0`, nilai bawaan yang ditetapkan standar [5]. Banyaknya *bucket* selama 1000 penyisipan, 1, 13, 29, 59, 127, 257, 541, 1109, adalah bilangan prima yang kira-kira dua kali ukuran sebelumnya: implementasi GCC memakai banyak *bucket* prima, dan dengan cara itulah ia bisa lolos dengan fungsi identitas sebagai `std::hash<int>` (Subbab 15.3.2 memperlihatkan bahwa m prima tahan terhadap kunci-kunci teratur). Bagian 3 mendaftar kueri yang memerlukan urutan: minimum dan maksimum, kode dalam suatu rentang, kode berikutnya sesudah kode tertentu. Pohon menjawab masing-masing dalam $O(\lg n)$ (ditambah ukuran jawabannya); tabel hash hanya dapat menjawabnya dengan memeriksa setiap kunci.

Bagian 4 memperlihatkan apa yang diperlukan tabel hash dari kunci buatan pengguna: `==` dan sebuah fungsi hash. `EditionHash` menggabungkan hash judul dan hash tahun dengan rumus yang diambil dari pustaka Boost, yang menggeser dan menambahkan konstanta rasio emas dari Persamaan (15.4) agar kedua bagian teraduk alih-alih saling meniadakan. Pohon akan memerlukan `operator<` sebagai gantinya. Bagian 5 adalah balapan sungguhan pada satu juta kunci acak 64-bit: dalam jalannya program ini tabel hash menyisipkan sekitar 4 kali lebih cepat dan mencari sekitar 25 kali lebih cepat. Sebagian besar waktu pohon habis untuk *cache miss*: setiap dari sekitar 20 tingkatnya adalah simpul di tempat lain dalam memori, sedangkan tabel hash menyentuh satu *bucket* dan biasanya satu simpul.

Tabel 15.5 meringkas pilihan itu.

Tabel 15.5 `std::map` dan `std::unordered_map` dibandingkan.

	`std::map`	`std::unordered_map`
Struktur	pohon merah-hitam (Subbab 11.9.3)	larik *bucket* berantai
Kebutuhan kunci	`operator<`	`std::hash<Key>` dan `operator==`
Urutan iterasi	terurut menurut kunci	tidak ditentukan, dapat berubah saat *rehash*
Pencarian, penyisipan, penghapusan	$O(\lg n)$ kasus terburuk	$O(1)$ rata-rata, $O(n)$ kasus terburuk
`lower_bound`, kueri rentang, min, maks	ya, $O(\lg n)$	tidak
Memori per elemen	simpul dengan tiga pointer dan satu warna	simpul dengan satu pointer, ditambah larik *bucket*
Pakailah bila	urutan, rentang, atau batas terjamin penting	hanya pencarian kunci persis yang penting

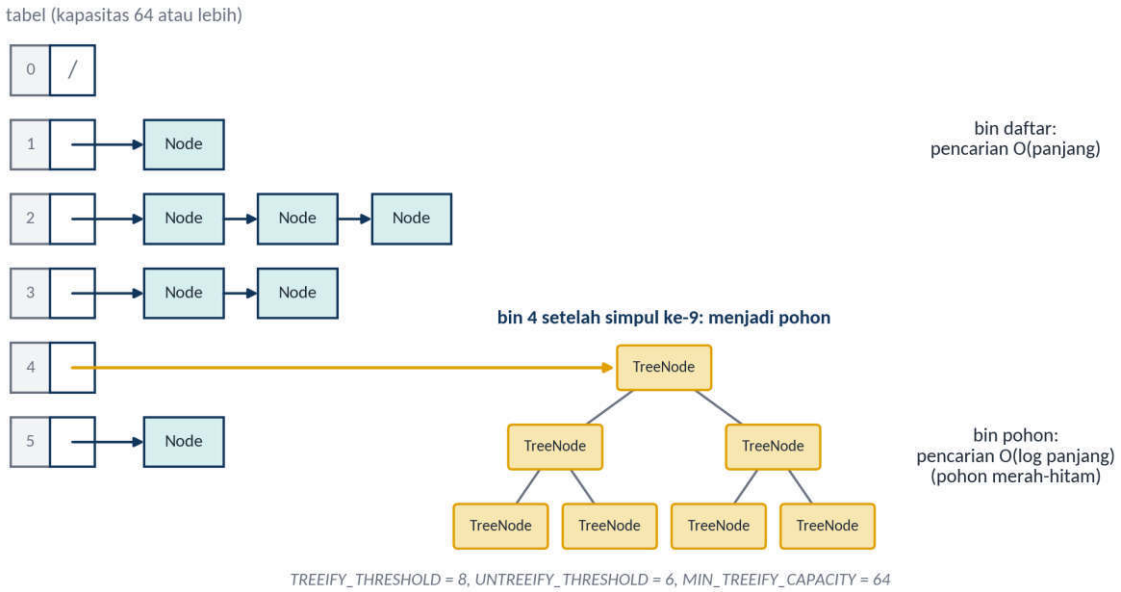
Pasangan yang sama ada untuk himpunan (`std::set`, `std::unordered_set`), dan Java memiliki pembagian yang sama antara `TreeMap` dan `HashMap`, yang diperkenalkan Bab 11 pada Subbab 11.11.2.

15.10 Di Dalam `HashMap` Milik Java dan `dict` Milik Python

15.10.1 Java: `HashMap` dan *bin*-nya yang menjadi pohon

`java.util.HashMap` milik Java adalah tabel berantai yang rancangannya dapat kita baca langsung dalam kode sumber JDK 25 [8], [6].

- **Tabelnya.** Larik berisi *bin* yang panjangnya selalu pangkat dua, bawaannya 16, dibuat pada penyisipan pertama. Faktor muat bawaannya 0,75, sehingga tabel berlipat dua ketika berisi lebih dari 12 entri, lalu 24, dan seterusnya. Menggandakan tabel berukuran pangkat dua memiliki sifat yang menyenangkan: entri-entri setiap *bin* tetap di indeks i atau pindah ke $i + m_{\text{lama}}$, ditentukan oleh satu bit hash, sehingga pengubahan ukuran membelah setiap rantai menjadi dua tanpa menghitung ulang satu hash pun.
- **Memilih *bin*.** Tabel berukuran pangkat dua mengambil bit-bit rendah hash, yang berbahaya (Subbab 15.3.2). Karena itu `HashMap` lebih dulu *menyebarkan* `hashCode()` kunci dengan melipat separuh atasnya ke separuh bawahnya, `h ^ (h >>> 16)`, lalu mengambil `(n - 1) & hash`, yaitu bit-bit rendahnya. **Kode 15.12** mencetak perhitungan ini untuk tiga kode.
- ***Bin* yang menjadi pohon.** Sebuah *bin* bermula sebagai *linked list* berisi objek `Node`. Ketika satu *bin* mencapai `TREEIFY_THRESHOLD = 8` entri, *bin* itu diubah menjadi pohon merah-hitam berisi objek `TreeNode`, asalkan tabel memiliki sedikitnya `MIN_TREEIFY_CAPACITY = 64` *bin*; tabel yang lebih kecil justru diperbesar, karena *bin* yang padat dalam tabel kecil lebih mungkin menandakan tabelnya terlalu kecil. *Bin* pohon menyusut kembali menjadi list pada `UNTREEIFY_THRESHOLD = 6`. *Bin* pohon diurutkan menurut nilai hash dan, di antara hash yang sama, menurut `compareTo` bila kuncinya `Comparable` (seperti `String`). **Gambar 15.9** menggambarkan tabel dengan satu *bin* yang telah menjadi pohon.



Gambar 15.9 Tabel `HashMap` dengan *bin* list dan satu *bin* yang telah menjadi pohon. Sebuah *bin* menjadi pohon merah-hitam berisi `TreeNode` ketika mengumpulkan 8 entri dalam tabel berisi sedikitnya 64 *bin*, sehingga pencarian di dalamnya memerlukan $O(\lg \ell)$ alih-alih $O(\ell)$ langkah untuk *bin* berisi ℓ entri.

Pengubahan menjadi pohon ditambahkan pada Java 8 melalui JEP 180 sebagai jawaban atas *hash flooding* [16]. Dengan kode hash acak, hal itu nyaris tidak pernah terpicu: menurut Persamaan (15.22) dengan $\alpha = 0.5$, muatan khas sebuah `HashMap`, sebuah *bin* tertentu berisi 8 entri dengan peluang $e^{-0.5}0.5^8/8! \approx 6 \times 10^{-8}$, angka yang dikutip dalam komentar kode sumbernya [8]. Tujuannya adalah membatasi kerusakan ketika kode hash buruk atau bermusuhan.

Kode 15.12 `LibraryMaps.java`: `HashMap`, `TreeMap`, dan `LinkedHashMap` pada data yang sama, cara memilih *bin*, *record* sebagai kunci, 65.536 kunci yang bertabrakan, dan pengukuran waktu.

```
// LibraryMaps.java -- java.util.HashMap against TreeMap and LinkedHashMap:
// ordering, how HashMap picks a bin, range queries, a record as a key,
// colliding keys, and a stopwatch (Section 15.10).
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.SplittableRandom;
import java.util.TreeMap;

public class LibraryMaps {
    record Edition(String title, int year) {}       // equals and hashCode for free

    static double ms(Runnable r) {
        long t0 = System.nanoTime();
        r.run();
        return (System.nanoTime() - t0) / 1e6;
    }

    public static void main(String[] args) {
        String[] code = {"B1005", "B1001", "B1008", "B1003",
                         "B1002", "B1007", "B1004", "B1006"};
        Map<String, Integer> hash = new HashMap<>();
        TreeMap<String, Integer> tree = new TreeMap<>();
        Map<String, Integer> linked = new LinkedHashMap<>();
        for (int i = 0; i < code.length; i++) {
            hash.put(code[i], i);
            tree.put(code[i], i);
            linked.put(code[i], i);
        }
        System.out.println("1. Inserted in the order B1005 B1001 B1008 B1003 B1002 "
                           + "B1007 B1004 B1006");
        System.out.println("  HashMap      : " + String.join(" ", hash.keySet()));
        System.out.println("  TreeMap      : " + String.join(" ", tree.keySet()));
        System.out.println("  LinkedHashMap: " + String.join(" ", linked.keySet()));

        System.out.println("\n2. How HashMap picks a bin (table of 16 bins)");
        for (String c : new String[] {"B1001", "B1002", "B1008"}) {
            int h = c.hashCode();
            int spread = h ^ (h >>> 16);             // HashMap.hash(key)
            System.out.printf("  %s  hashCode = %d  bin = %d%n", c, h, spread & 15);
        }
        System.out.printf("  \"Aa\".hashCode() = %d, \"BB\".hashCode() = %d%n",
                          "Aa".hashCode(), "BB".hashCode());
```

Kode 15.12 (Lanjutan) *LibraryMaps.java: HashMap, TreeMap, dan LinkedHashMap pada data yang sama, cara memilih bin, record sebagai kunci, 65.536 kunci yang bertabrakan, dan pengukuran waktu.*

```
        System.out.println("\n3. Questions only the TreeMap can answer");
        System.out.println("  first and last code: " + tree.firstKey() + ", "
                           + tree.lastKey());
        Map<String, Integer> range = tree.subMap("B1003", true, "B1006", true);
        System.out.println("  codes from B1003 to B1006: "
                           + String.join(" ", range.keySet()));
        System.out.println("  first code after B1004: " + tree.higherKey("B1004"));

        System.out.println("\n4. A record is a key with two fields");
        Map<Edition, Integer> copies = new HashMap<>();
        copies.put(new Edition("Laskar Pelangi", 2005), 3);
        copies.put(new Edition("Laskar Pelangi", 2020), 2);
        System.out.println("  copies of Laskar Pelangi (2020): "
                           + copies.get(new Edition("Laskar Pelangi", 2020)));

        System.out.println("\n5. 65,536 keys whose hashCode values are all equal "
                           + "(times vary)");
        List<String> attack = new ArrayList<>(), ordinary = new ArrayList<>();
        for (int mask = 0; mask < 65536; mask++) {
            StringBuilder sb = new StringBuilder();
            for (int b = 15; b >= 0; b--)
                sb.append((mask >> b & 1) != 0 ? "BB" : "Aa");
            attack.add(sb.toString());
            ordinary.add(String.format("reader%05d", mask));
        }
        long distinct = attack.stream().map(String::hashCode).distinct().count();
        System.out.println("  distinct hashCode values among the attack keys: "
                           + distinct);
        for (int round = 0; round < 2; round++) {    // round 0 warms up the JIT
            Map<String, Integer> a = new HashMap<>(), o = new HashMap<>();
            double ta = ms(() -> {
                for (String s : attack) a.put(s, 1);
                for (String s : attack) a.get(s);
            });
            double to = ms(() -> {
                for (String s : ordinary) o.put(s, 1);
                for (String s : ordinary) o.get(s);
            });
            if (round == 1)
                System.out.printf("  insert + find all: attack keys %.0f ms, "
                                  + "ordinary keys %.0f ms%n", ta, to);
        }
```

Kode 15.12 (Lanjutan) *LibraryMaps.java: HashMap, TreeMap, dan LinkedHashMap pada data yang sama, cara memilih bin, record sebagai kunci, 65.536 kunci yang bertabrakan, dan pengukuran waktu.*

```
        System.out.println("\n6. One million random long keys "
                           + "(times vary by machine)");
        SplittableRandom rnd = new SplittableRandom(15);
        long[] keys = new long[1_000_000];
        for (int i = 0; i < keys.length; i++) keys[i] = rnd.nextLong();
        Map<Long, Integer> tm = new TreeMap<>(), hm = new HashMap<>();
        long[] found = {0};
        double ti = ms(() -> { for (long k : keys) tm.put(k, 1); });
        double tf = ms(() -> {
            for (long k : keys) if (tm.containsKey(k)) found[0]++;
        });
        double hi = ms(() -> { for (long k : keys) hm.put(k, 1); });
        double hf = ms(() -> {
            for (long k : keys) if (hm.containsKey(k)) found[0]++;
        });
        System.out.printf("  %-10s %10s %10s%n", "", "insert", "find");
        System.out.printf("  %-10s %7.0f ms %7.0f ms%n", "TreeMap", ti, tf);
        System.out.printf("  %-10s %7.0f ms %7.0f ms%n", "HashMap", hi, hf);
        System.out.println("  (keys found: " + found[0] + ")");
    }
}
```

```
1. Inserted in the order B1005 B1001 B1008 B1003 B1002 B1007 B1004 B1006
  HashMap      : B1008 B1007 B1002 B1001 B1004 B1003 B1006 B1005
  TreeMap      : B1001 B1002 B1003 B1004 B1005 B1006 B1007 B1008
  LinkedHashMap: B1005 B1001 B1008 B1003 B1002 B1007 B1004 B1006

2. How HashMap picks a bin (table of 16 bins)
  B1001  hashCode = 62459810  bin = 11
  B1002  hashCode = 62459811  bin = 10
  B1008  hashCode = 62459817  bin = 0
  "Aa".hashCode() = 2112, "BB".hashCode() = 2112

3. Questions only the TreeMap can answer
  first and last code: B1001, B1008
  codes from B1003 to B1006: B1003 B1004 B1005 B1006
  first code after B1004: B1005

4. A record is a key with two fields
  copies of Laskar Pelangi (2020): 2

5. 65,536 keys whose hashCode values are all equal (times vary)
  distinct hashCode values among the attack keys: 1
  insert + find all: attack keys 122 ms, ordinary keys 9 ms

6. One million random long keys (times vary by machine)
                insert       find
  TreeMap        733 ms     572 ms
  HashMap        255 ms     140 ms
  (keys found: 2000000)
```

Keluaran itu mengulang pelajaran Subbab 15.9 dalam kosakata Java. `HashMap` beriterasi dalam urutan *bin*, `TreeMap` (pohon merah-hitam, Subbab 11.11.2) dalam urutan terurut, dan `LinkedHashMap`, tabel hash dengan *double linked list* yang dijalin melalui entri-entrinya (Subbab 9.12.2), dalam urutan penyisipan. Kode B1001 dan B1002, yang nilai `hashCode`-nya berselisih satu, jatuh di *bin* yang berbeda. Sebuah `record` memperoleh `equals` dan `hashCode` secara otomatis, sehingga dapat menjadi kunci tanpa kode tambahan. Bagian 5 adalah percobaan *flooding* Subbab 15.8 yang dijalankan terhadap pustaka sungguhan: 65.536 kunci dengan satu nilai `hashCode` tunggal semuanya disisipkan dan ditemukan dalam sekitar sepersepuluh detik, hanya sekitar tiga belas kali lebih lambat daripada kunci biasa, karena mereka tinggal dalam

pohon merah-hitam yang diurutkan oleh `compareTo`. Dengan *bin* list biasa, pekerjaan yang sama akan memerlukan sekitar $65536^2/2 \approx 2 \times 10^9$ perbandingan *string*. Pada bagian 6, `HashMap` mengalahkan `TreeMap` dengan faktor sekitar 3 sampai 4.

15.10.2 Python: `dict` yang ringkas dan berurutan menurut penyisipan

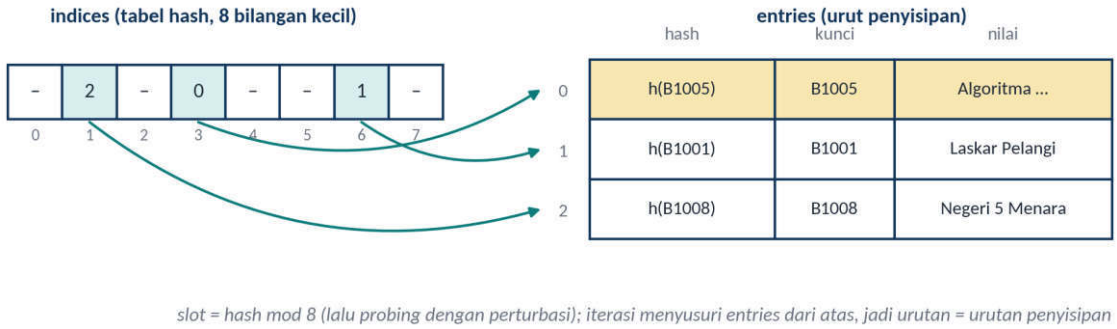
`dict` milik Python adalah tabel hash yang paling banyak dipakai dalam bahasa pengajaran paling populer di dunia, dan atribut setiap objek Python, variabel global setiap modul, dan metode setiap kelas tinggal di dalamnya. Rancangannya saat ini diuraikan panjang lebar dalam komentar berkas sumber CPython `Objects/dictobject.c` [7].

- **Pengalamatan terbuka dengan perturbasi.** `dict` adalah tabel ber-*probing* berisi 2^k slot, sedikitnya 8. *Probe* pertamanya adalah k bit rendah hash, dan ia berlanjut dengan rekurensi

$$j \leftarrow (5j + 1 + \text{perturb}) \bmod 2^k, \qquad \text{perturb} \leftarrow \text{perturb} \gg 5, \tag{15.26}$$

dengan *perturb* bermula sebagai nilai hash penuh. Bit-bit tinggi yang digeser masuk membuat barisan *probe* bergantung pada seluruh hash, seperti *double hashing*; begitu *perturb* mencapai nol, rekurensi $j \to 5j + 1 \bmod 2^k$ saja sudah mengunjungi setiap slot, sehingga slot kosong selalu ditemukan.

- **Dua larik, bukan satu.** Sejak Python 3.6 tabelnya *ringkas* (*compact*), gagasan yang diusulkan Raymond Hettinger pada tahun 2012 [17]. Tabel hash yang sebenarnya, `indices`, hanya berisi bilangan bulat kecil (masing-masing satu *byte* selama tabel memiliki paling banyak 128 slot); masing-masing menunjuk ke larik padat `entries` berisi tripel (hash, kunci, nilai) yang ditambahkan dalam urutan penyisipan. **Gambar 15.10** menggambarkannya. *Probing* terjadi dalam larik `indices` yang kecil, dan entri-entri yang besar tidak pernah digeser-geser oleh tabrakan.
- **Urutan secara cuma-cuma.** Iterasi cukup menyusuri `entries` dari yang pertama hingga yang terakhir, sehingga `dict` mengembalikan kuncinya dalam urutan penyisipan. Hal ini bermula sebagai efek samping tata letak ringkas pada 3.6 dan dinyatakan sebagai jaminan bahasa pada Python 3.7 [18]. `set` tidak memiliki larik `entries`, dan urutannya *bukan* urutan penyisipan.
- **Muatan dan pertumbuhan.** `dict` boleh mengisi paling banyak 2/3 slotnya (`USABLE_FRACTION`), dan ketika penuh ia tumbuh untuk menampung sekitar tiga kali banyaknya butir saat itu (`GROWTH_RATE`).
- **Nilai hash.** Bilangan bulat di-hash menjadi dirinya sendiri modulo bilangan prima $2^{61} - 1$ (kecuali bahwa `hash(-1)` bernilai -2, karena -1 menandakan galat dalam antarmuka C milik CPython). *String* dan *bytes* memakai SipHash-1-3 dengan kunci acak yang dipilih ketika *interpreter* dimulai [15], [14]; variabel lingkungan `PYTHONHASHSEED` menetapkan kunci itu agar jalannya program dapat diulang.



Gambar 15.10 Tata letak `dict` ringkas dengan tiga butir: tabel `indices` kecil berisi 8 slot satu-*byte* di-*probe* menurut hash, dan setiap slot terpakai menyimpan posisi sebuah entri dalam larik padat `entries`, yang dijaga dalam urutan penyisipan. (Posisi slot yang diperlihatkan hanya ilustrasi; nilai hash sebenarnya berubah dari satu jalannya program ke jalannya program berikutnya.)

Kode 15.13 mengamati setiap butir tersebut dari kode Python.

Kode 15.13 `library_maps.py`: urutan penyisipan, nilai hash bilangan bulat dan *string*, pengacakan hash antarjalannya *interpreter*, pertumbuhan `dict`, dan pengukuran waktu terhadap pencarian biner.

```
# library_maps.py -- Python's dict and set: insertion order, how small
# integers and strings hash, hash randomization between runs, how a dict
# grows, and a stopwatch against binary search (Section 15.10).
import bisect
import os
import random
import subprocess
import sys
import time

code = ["B1005", "B1001", "B1008", "B1003", "B1002", "B1007", "B1004", "B1006"]
catalog = {c: i for i, c in enumerate(code)}

print("1. Inserted in the order", " ".join(code))
print("  dict         :", " ".join(catalog))
print("  sorted(dict) :", " ".join(sorted(catalog)))

print("\n2. Hash values")
print("  hash(42) =", hash(42), "  hash(-1) =", hash(-1),
      "  hash(2**61) =", hash(2**61))
print("  sys.hash_info.algorithm =", sys.hash_info.algorithm,
      "  modulus = 2**61 - 1:", sys.hash_info.modulus == 2**61 - 1)
script = "print(hash('B1001'), ' '.join({'B1001', 'B1002', 'B1003', 'B1004'}))"
for seed in ("0", "1", "2"):                          # three separate interpreters
    env = dict(os.environ, PYTHONHASHSEED=seed)
    out = subprocess.run([sys.executable, "-c", script], env=env,
                         capture_output=True, text=True).stdout.strip()
    print("  PYTHONHASHSEED=%s: %s" % (seed, out))

print("\n3. Memory of a dict as it grows (sizes vary by Python version)")
d, last, steps = {}, sys.getsizeof({}), []
for i in range(1, 100):
    d[i] = i
    size = sys.getsizeof(d)
    if size != last:
        steps.append("%2d key(s) -> %4d bytes" % (i, size))
        last = size
print("  " + "\n  ".join(steps))
```

Kode 15.13 (Lanjutan) *library_maps.py: urutan penyisipan, nilai hash bilangan bulat dan string, pengacakan hash antarjalannya interpreter, pertumbuhan dict, dan pengukuran waktu terhadap pencarian biner.*

```
print("\n4. 200,000 random keys: dict against binary search (times vary)")
rng = random.Random(15)
keys = [rng.getrandbits(62) for _ in range(200_000)]
t0 = time.perf_counter()
table = {k: 1 for k in keys}
found = sum(1 for k in keys if k in table)
t1 = time.perf_counter()
ordered = sorted(keys)
found += sum(1 for k in keys if ordered[bisect.bisect_left(ordered, k)] == k)
t2 = time.perf_counter()
print("  dict (build + find all)           : %6.0f ms" % ((t1 - t0) * 1000))
print("  sorted list + bisect (sort + find): %6.0f ms" % ((t2 - t1) * 1000))
print("  (keys found: %d)" % found)
```

```
1. Inserted in the order B1005 B1001 B1008 B1003 B1002 B1007 B1004 B1006
  dict         : B1005 B1001 B1008 B1003 B1002 B1007 B1004 B1006
  sorted(dict) : B1001 B1002 B1003 B1004 B1005 B1006 B1007 B1008

2. Hash values
  hash(42) = 42   hash(-1) = -2   hash(2**61) = 1
  sys.hash_info.algorithm = siphash13   modulus = 2**61 - 1: True
  PYTHONHASHSEED=0: -1661493604806230238 B1004 B1001 B1002 B1003
  PYTHONHASHSEED=1: -7535737535382290098 B1002 B1003 B1001 B1004
  PYTHONHASHSEED=2: -4607229078860313946 B1002 B1003 B1004 B1001

3. Memory of a dict as it grows (sizes vary by Python version)
   1 key(s) ->  224 bytes
   6 key(s) ->  352 bytes
  11 key(s) ->  632 bytes
  22 key(s) -> 1168 bytes
  43 key(s) -> 2264 bytes
  86 key(s) -> 4688 bytes

4. 200,000 random keys: dict against binary search (times vary)
  dict (build + find all)           :     44 ms
  sorted list + bisect (sort + find):    164 ms
  (keys found: 400000)
```

Bagian 2 menjalankan tiga *interpreter* terpisah dengan tiga nilai `PYTHONHASHSEED`: hash "B1001" berbeda setiap kali, begitu pula urutan iterasi sebuah *set* berisi empat elemen, sedangkan `dict` tetap menjaga urutan penyisipan. Bagian 3 memperlihatkan tabel tumbuh ketika kunci ke-6, ke-11, ke-22, ke-43, dan ke-86 tiba, tepat ketika batas 2/3 tabel berisi 8, 16, 32, 64, dan 128 slot (ukuran terpakai 5, 10, 21, 42, dan 85) terlampaui; angka *byte*-nya milik Python 3.11 dan sedikit berbeda antarversi. Bagian 4 membandingkan pembangunan `dict` berisi 200.000 kunci dan pencarian setiap kunci dengan pengurutan list dan pencarian setiap kunci secara biner: tabel hash menang dengan faktor sekitar 4.

Tabel 15.6 menghimpun fakta-fakta Subbab 15.9 dan 15.10.

Tabel 15.6 Tabel hash dalam pustaka baku C++, Java, dan Python (GCC 13, JDK 25, CPython 3.14).

	C++ `std::unordered_map`	Java `HashMap`	Python `dict`
Penanganan tabrakan	*separate chaining*	*separate chaining*, *bin* menjadi pohon merah-hitam pada 8	pengalamatan terbuka, *probing* berperturbasi Persamaan (15.26)
Ukuran tabel	prima (GCC)	pangkat 2, mulai 16	pangkat 2, mulai 8
Muatan maksimum	1,0 (dapat diatur)	0,75 (dapat diatur)	2/3
Hash bawaan bilangan bulat	identitas (GCC)	`Integer.hashCode` = nilainya, lalu disebarkan	nilainya modulo $2^{61} - 1$
Hash bawaan *string*	hash *byte* dengan *seed* tetap	$\sum s_i 31^{L-1-i}$ dalam 32 bit, disimpan	SipHash-1-3, kunci acak per proses
Urutan iterasi	tidak ditentukan	tidak ditentukan (`LinkedHashMap`: penyisipan)	urutan penyisipan
Saudara terurut	`std::map`	`TreeMap`	tidak ada bawaan (urutkan, atau `bisect`)

15.11 Penerapan

Tabel hash layak dipakai setiap kali program terus bertanya "pernahkah aku melihat ini?" atau "apa yang kuketahui tentang kunci ini?". **Kode 15.14** menyelesaikan empat masalah semacam itu untuk Pustaka Ceria; `Applications.java` dan `applications.py` di folder kode bab ini mencetak keluaran yang sama.

Kode 15.14 `applications.cpp`: menghitung kata dalam katalog, mendeteksi pengembalian ganda, *two-sum* pada harga buku, dan *cache* LRU berisi rekaman katalog.

```cpp
// applications.cpp -- four everyday jobs for a hash table (Section 15.11):
// counting words, detecting duplicates, two-sum, and an LRU cache.
#include <algorithm>
#include <cctype>
#include <cstdio>
#include <list>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <utility>
#include <vector>

const std::vector<std::string> TITLES = {
    "Laskar Pelangi", "Sang Pemimpi", "Edensor", "Maryamah Karpov",
    "Bumi Manusia", "Anak Semua Bangsa", "Jejak Langkah", "Rumah Kaca",
    "Filosofi Kopi", "Perahu Kertas", "Negeri 5 Menara", "Ranah 3 Warna",
    "Rantau 1 Muara", "Cantik Itu Luka", "Lelaki Harimau", "Ronggeng Dukuh Paruk",
    "Sejarah Nusantara", "Cerita Rakyat Nusantara", "Algoritma & Struktur Data",
    "Struktur Data dengan C++", "Dasar-Dasar Pemrograman", "Fisika Dasar",
    "Kimia Dasar", "Matematika Diskrit", "Pengantar Basis Data",
    "Sistem Basis Data", "Cerita Anak Nusantara", "Dongeng Anak Indonesia"};

void countWords() {
    std::unordered_map<std::string, int> count;
    int total = 0;
    for (const std::string &t : TITLES) {
        std::string w;
        for (std::size_t i = 0; i <= t.size(); ++i) {
            if (i < t.size() && std::isalnum((unsigned char)t[i])) {
                w += (char)std::tolower((unsigned char)t[i]);
            } else if (!w.empty()) {                // a word just ended
                ++count[w];                         // absent key starts at 0
                ++total;
                w.clear();
            }
        }
    }
    std::vector<std::pair<std::string, int>> v(count.begin(), count.end());
    std::sort(v.begin(), v.end(), [](const auto &a, const auto &b) {
        return a.second != b.second ? a.second > b.second : a.first < b.first;
    });
    std::printf("1. Words in %zu titles: %d in all, %zu different\n", TITLES.size(),
                total, count.size());
    std::printf("   most frequent:");
    for (int i = 0; i < 6; ++i)
        std::printf(" %s (%d)", v[i].first.c_str(), v[i].second);
```

Kode 15.14 (Lanjutan) *applications.cpp: menghitung kata dalam katalog, mendeteksi pengembalian ganda, two-sum pada harga buku, dan cache LRU berisi rekaman katalog.*

```
    std::printf("\n");
}

void duplicates() {
    const std::vector<std::string> returned = {"B1003", "B1017", "B1008", "B1003",
                                               "B1022", "B1017", "B1030", "B1003"};
    std::unordered_set<std::string> seen, reported;
    std::printf("\n2. Books returned today:");
    for (const std::string &c : returned) std::printf(" %s", c.c_str());
    std::printf("\n");
    for (std::size_t i = 0; i < returned.size(); ++i) {
        const std::string &c = returned[i];
        if (!seen.insert(c).second && reported.insert(c).second)   // seen before?
            std::printf("  %s appears again at position %zu\n", c.c_str(), i);
    }
    std::printf("  %zu different books, %zu set lookups (a double loop would "
                "compare up to %zu pairs)\n", seen.size(), returned.size(),
                returned.size() * (returned.size() - 1) / 2);
}

void twoSum() {
    const std::vector<long> price = {45000, 72000, 38000, 91000,
                                     64000, 27000, 83000, 55000};
    const long budget = 119000;
    std::unordered_map<long, std::size_t> where;     // price -> index seen so far
    std::printf("\n3. Two books that cost exactly Rp%ld together\n", budget);
    for (std::size_t i = 0; i < price.size(); ++i) {
        auto it = where.find(budget - price[i]);
        if (it != where.end()) {
            std::printf("  books %zu and %zu: Rp%ld + Rp%ld (found after %zu "
                        "lookups)\n", it->second, i, price[it->second], price[i],
                        i + 1);
            return;
        }
        where[price[i]] = i;
    }
    std::printf("  no such pair\n");
}

class LruCache {                                     // a hash map + a linked list
public:
    explicit LruCache(std::size_t capacity) : capacity_(capacity) {}
```

Kode 15.14 (Lanjutan) *applications.cpp: menghitung kata dalam katalog, mendeteksi pengembalian ganda, two-sum pada harga buku, dan cache LRU berisi rekaman katalog.*

```
    bool get(const std::string &key) {              // true on a hit
        auto it = pos_.find(key);
        if (it != pos_.end()) {                     // hit: move to the front
            order_.splice(order_.begin(), order_, it->second);
            return true;
        }
        if (order_.size() == capacity_) {           // full: evict the back
            pos_.erase(order_.back());
            order_.pop_back();
        }
        order_.push_front(key);                     // miss: load, put in front
        pos_[key] = order_.begin();
        return false;
    }

    std::string contents() const {
        std::string s;
        for (const std::string &k : order_) s += (s.empty() ? "" : " ") + k;
        return s;
    }

private:
    std::size_t capacity_;
    std::list<std::string> order_;                  // most recent first
    std::unordered_map<std::string, std::list<std::string>::iterator> pos_;
};

void cache() {
    const char *req[10] = {"B1001", "B1002", "B1001", "B1003", "B1001",
                           "B1002", "B1004", "B1001", "B1002", "B1005"};
    LruCache c(3);
    int hits = 0;
    std::printf("\n4. An LRU cache of 3 catalog records (most recent first)\n");
    for (const char *r : req) {
        bool hit = c.get(r);
        hits += hit;
        std::printf("  get %s: %-4s  cache: %s\n", r,
                    hit ? "hit" : "miss", c.contents().c_str());
    }
    std::printf("  %d hits out of 10 requests\n", hits);
}
```

Kode 15.14 (Lanjutan) *applications.cpp: menghitung kata dalam katalog, mendeteksi pengembalian ganda, two-sum pada harga buku, dan cache LRU berisi rekaman katalog.*

```
int main() {
    countWords();
    duplicates();
    twoSum();
    cache();
    return 0;
}
```

```
1. Words in 28 titles: 70 in all, 57 different
  most frequent: dasar (4) data (4) anak (3) nusantara (3) basis (2) cerita (2)

2. Books returned today: B1003 B1017 B1008 B1003 B1022 B1017 B1030 B1003
  B1003 appears again at position 3
  B1017 appears again at position 5
  5 different books, 8 set lookups (a double loop would compare up to 28 pairs)

3. Two books that cost exactly Rp119000 together
  books 4 and 7: Rp64000 + Rp55000 (found after 8 lookups)

4. An LRU cache of 3 catalog records (most recent first)
  get B1001: miss  cache: B1001
  get B1002: miss  cache: B1002 B1001
  get B1001: hit   cache: B1001 B1002
  get B1003: miss  cache: B1003 B1001 B1002
  get B1001: hit   cache: B1001 B1003 B1002
  get B1002: hit   cache: B1002 B1001 B1003
  get B1004: miss  cache: B1004 B1002 B1001
  get B1001: hit   cache: B1001 B1004 B1002
  get B1002: hit   cache: B1002 B1001 B1004
  get B1005: miss  cache: B1005 B1002 B1001
  5 hits out of 10 requests
```

15.11.1 Menghitung kata dalam katalog

Perpustakaan ingin tahu kata mana yang paling sering muncul dalam judul-judulnya, untuk menyusun indeks subjek. `countWords` memecah setiap judul menjadi kata (huruf dan digit, huruf kecil) dan menambah satu pencacah per kata dalam `unordered_map<string, int>`. Baris `++count[w]` mengerjakan semuanya: `operator[]` menemukan pencacah kata itu, atau membuatnya dengan nilai 0 bila kata itu baru, lalu penambahan menyusul. Setiap kata berongkos satu pencarian harapan $O(1)$, sehingga menghitung W kata memerlukan waktu harapan $O(W)$, sedangkan mengurutkan kata-kata lebih dulu akan berongkos $O(W \lg W)$. Hanya *laporannya* yang memerlukan urutan: D kata berbeda disalin ke sebuah *vector* dan diurutkan menurut hitungannya, $O(D \lg D)$. Ke-28 judul memuat 70 kata, 57 di antaranya berbeda; *dasar* dan *data* masing-masing muncul 4 kali. Java menulis pembaruan pencacah sebagai `count.merge(w, 1, Integer::sum)`, dan Python memiliki *dictionary* penghitung yang siap pakai, `collections.Counter`.

15.11.2 Mendeteksi duplikat

Pada jam tutup meja pengembalian memindai setiap buku yang kembali, dan kode yang sama muncul dua kali berarti kesalahan pemindaian. Dengan `unordered_set` berisi kode-kode yang sudah terlihat, setiap pemindaian adalah satu penyisipan harapan $O(1)$; `insert` mengembalikan pasangan yang `.second`-nya `false` bila kode itu sudah ada, sehingga duplikat terdeteksi pada langkah yang sama. Himpunan kedua, `reported`, membuat setiap duplikat dilaporkan sekali. Untuk n pemindaian itu berarti waktu harapan $O(n)$, sedangkan membandingkan setiap pasangan memerlukan $n(n-1)/2$ perbandingan, 28 untuk kedelapan kode ini tetapi sekitar 500 juta untuk 32.000 kode. Inilah metode yang dipakai Subbab 8.7.3 untuk membuang duplikat dari *linked list*, dan metode ke-256 pencacah Subbab 4.4.3, kini untuk kunci yang bukan bilangan bulat kecil.

15.11.3 *Two-sum*

Seorang donatur menawarkan Rp119.000 untuk tepat dua buku. Dua harga mana yang jumlahnya sama dengan anggaran itu? Mencoba setiap pasangan berongkos $\Theta(n^2)$. Dengan *hash map*

pertanyaan untuk setiap buku menjadi satu pencarian: apakah *pelengkapnya*, `budget - price[i]`, adalah harga buku yang sudah kita lihat? `twoSum` menyimpan setiap harga beserta indeksnya sesudah memeriksa pelengkapnya, sehingga ia menemukan pasangan dalam satu lintasan, waktu harapan $O(n)$. Keluarannya memperlihatkan buku 4 dan 7, Rp64.000 dan Rp55.000, ditemukan pada pencarian kedelapan. Memeriksa sebelum menyisipkan juga mencegah sebuah buku dipasangkan dengan dirinya sendiri ketika harganya tepat separuh anggaran. Pola yang sama, "simpan yang sudah terlihat, cari yang diperlukan", menyelesaikan satu keluarga masalah: Soal Latihan 15.9 menghitung rentang hari yang jumlah peminjamannya sama dengan bilangan tertentu dengan menyimpan jumlah awalan (*prefix sum*).

15.11.4 *Cache*

Cache menyimpan hasil operasi yang mahal (kueri katalog yang lambat, halaman yang diambil lewat jaringan, skor yang telah dihitung) agar permintaan yang berulang langsung dijawab. *Cache* harus menjawab "apakah kunci ini ada di sini?" dalam $O(1)$, yang memerlukan *hash map*, dan harus memutuskan apa yang dibuang ketika penuh. Kebijakan *least recently used* (LRU) membuang entri yang paling lama tidak dipakai, yang memerlukan entri-entri terurut menurut pemakaian terakhir dan dipindahkan ke depan pada setiap akses, yang memerlukan *double linked list*. Subbab 9.9 membangun persis pasangan ini, list bersentinel ditambah *hash map* dari kunci ke simpul. `LruCache` pada **Kode 15.14** memakai versi pustakanya: `std::list` berisi kunci, yang terbaru di depan, dan `unordered_map` dari setiap kunci ke *iterator*-nya di dalam list. Pada *hit*, `splice` memindahkan simpul ke depan dalam $O(1)$ tanpa menyalinnya; pada *miss* dengan *cache* penuh, kunci di belakang dihapus dari kedua struktur; lalu kunci baru masuk di depan. Pada keluarannya, sepuluh permintaan ke *cache* berkapasitas tiga rekaman menghasilkan lima *hit*; *miss* untuk B1004 mengusir B1003, yang paling lama tidak dipakai. `LinkedHashMap` milik Java dalam urutan akses dan `OrderedDict` milik Python adalah kombinasi ini yang dikemas sebagai satu kelas, dan `functools.lru_cache` milik Python membungkusnya di sekitar sebuah fungsi.

15.11.5 *Map* terbalik

Kadang-kadang pertanyaannya berjalan mundur: bukan "tiket ini menuju ke mana?" melainkan "tiket mana yang tiba di sini?". Menyimpan *map* kedua dengan kunci dan nilai yang ditukar menjawabnya dalam $O(1)$ juga, dengan harga dua kali memori. Soal rencana perjalanan di Bab 16 memakai persis ini: satu-satunya kota yang tidak pernah muncul sebagai tujuan dalam *map* terbalik adalah tempat perjalanan dimulai.

15.12 Profitina dalam Praktik

Profitina adalah *platform* visibilitas AI (GEO): ia memindai situs web sebuah merek, menilai seberapa siap situs itu dikutip mesin penjawab AI, menjelaskan setiap sinyal, dan membantu pemiliknya memperbaiki apa yang ditemukan [19]. Profitina adalah aplikasi Python yang dibangun dengan FastAPI di atas PostgreSQL, dan tabel hash muncul di mana pun aplikasi itu harus mengenali sesuatu yang pernah dilihatnya. Kita menguraikan bentuk setiap masalah dan kelas struktur yang cocok untuknya; pilihan implementasi di dalam produk itu adalah urusannya sendiri.

- **Satu halaman, banyak alamat.** *Crawler* menjumpai halaman yang sama di bawah banyak URL: dengan dan tanpa `www.`, dengan `http` atau `https`, dengan garis miring di ujung, dengan parameter pelacak yang ditambahkan. Sebelum dipindai, setiap URL *dikanonikkan* (skema dan *host* dijadikan huruf kecil, *port* bawaan dan *fragment* dibuang, parameter pelacak yang dikenal

dihapus, parameter diurutkan), dan *string* kanonik itu masuk ke himpunan URL yang sudah dikunjungi. Uji "pernah dikunjungi?" lalu menjadi satu pencarian harapan $O(1)$, deteksi duplikat Subbab 15.11.2, dan itulah yang mencegah penelusuran situs dengan ribuan tautan mengunjungi ulang halaman tanpa akhir.

- **Halaman yang hampir sama.** Banyak situs menerbitkan salinan yang nyaris sama: artikel yang sama di bawah dua kategori, halaman produk yang berbeda satu baris. Membandingkan setiap pasangan halaman kata demi kata berongkos kuadratik. Sebagai gantinya, setiap halaman dapat dipotong menjadi jendela-jendela beberapa kata yang saling tumpang-tindih, setiap jendela di-hash dengan *rolling hash* (Subbab 15.3.5), dan halaman itu diwakili oleh himpunan kecil sidik jari tersebut. Dua halaman yang himpunan sidik jarinya banyak beririsan adalah nyaris-duplikat, dan irisan dua himpunan dihitung dengan pencarian di *hash set*.
- **Menyaring tautan.** Setiap tautan yang diunggah ke komunitas Profitina disaring oleh lapisan aturan yang memadukan beberapa sinyal. Beberapa sinyal itu adalah pertanyaan keanggotaan, misalnya apakah sebuah domain muncul dalam daftar pemendek URL yang dikenal atau *host* yang dikenal buruk, dan *hash set* menjawab masing-masing dalam waktu konstan betapa pun panjangnya daftar itu.
- **Menghitung per pengguna.** Pembatasan laju *login*, batas harian perolehan PROFIT, dan pencacah "N baru" pada kanal yang diikuti semuanya adalah pencacah yang dikunci oleh akun atau kanal: pola penghitungan kata Subbab 15.11.1, biasanya disimpan dalam *hash map* dengan waktu kedaluwarsa.
- ***Cache* untuk jawaban yang mahal.** Pemindaian dan model Qwen2.5 lokal itu mahal, dan pertanyaan yang sama berulang. *Cache* terbatas dengan kebijakan LRU Subbab 15.11.4 menghindari pengulangan pekerjaan.
- **Jenis hash yang lain.** Profitina menyimpan kata sandi dengan Argon2id, sebagaimana dituntut pekerjaan kepatuhannya terhadap UU PDP 27/2022. Itu adalah hash *kata sandi*, sengaja lambat, bergaram (*salted*), dan rakus memori, kebalikan dari hash tabel yang cepat dalam bab ini; peringatan pada Subbab 15.3.7 menjelaskan mengapa keduanya tidak boleh tertukar.

Versi kecil yang dapat dijalankan dari pola pertama merupakan latihan yang baik untuk studi kasus Profitina Laundry: simpan setiap pesanan di bawah kode pesanannya dalam sebuah `unordered_map`, tolak pesanan kedua dengan kode yang sama, dan hitung banyaknya pakaian per pelanggan dalam *map* kedua.

15.13 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Mengambil sisa pembagian dengan pangkat dua pada hash yang lemah. `k % 16` hanya menyisakan empat bit dari k; untuk harga bulat atau alamat yang sejajar, keempatnya bisa nol semua. Pakailah ukuran tabel prima dengan metode pembagian, atau aduklah hash lebih dulu (Subbab 15.3.2 dan 15.3.6).

Menjumlahkan karakter. Jumlah kode karakter mengabaikan urutan, sehingga anagram bertabrakan, dan menjejalkan *string* pendek ke rentang yang sempit. Pakailah hash polinomial Persamaan (15.5) atau hash pustaka.

Sisa pembagian negatif. Di C++ dan Java, `-7 % 5` bernilai -2, bukan 3, sehingga `h % m` dari hash negatif adalah indeks negatif. Lakukan hashing ke tipe tak bertanda, atau pakai `Math.floorMod` di Java; `%` milik Python sudah tak negatif.

Mengubah kunci selagi berada dalam tabel. Tabel mencatat kunci di bawah hash yang dimilikinya ketika disisipkan. Bila objek Java yang dipakai sebagai kunci mengubah medan yang

menjadi dasar `hashCode`, tabel mencari di *bucket* yang salah dan entri itu hilang untuk selamanya. Kunci harus tak dapat diubah (*immutable*), atau setidaknya tidak diubah selama tersimpan.

Mendefinisikan kesamaan tanpa hash yang sepadan. Dua kunci yang sama harus memiliki hash yang sama. Kelas Java yang menimpa `equals` tetapi tidak `hashCode`, atau *functor* hash C++ yang mengabaikan medan yang dibandingkan `==`, melanggar aturan ini, dan kunci-kunci yang sama berakhir di *bucket* berbeda. *Record* Java dan `@dataclass(frozen=True)` milik Python membangkitkan keduanya bersamaan.

Mengosongkan slot saat penghapusan pada pengalamatan terbuka. Hal itu memotong barisan *probe* kunci-kunci berikutnya, yang kemudian tidak dapat ditemukan (**Gambar 15.5**). Pakailah nisan atau penghapusan geser mundur.

Menghentikan penyisipan pada nisan pertama. Kunci itu mungkin sudah tersimpan lebih jauh di barisan *probe*-nya; teruslah berjalan hingga slot kosong, lalu pakai ulang nisan pertama (**Pseudokode 15.3**).

Membiarkan faktor muat membesar. Tabel berantai tanpa *rehashing* berubah menjadi m list panjang, dan tabel ber-*probing* yang mendekati $\alpha = 1$ mengubah setiap pencarian menjadi pemindaian. Lakukan *rehash* dengan *menggandakan*, jangan pernah dengan menambah konstanta (Persamaan (15.23)).

Mengandalkan urutan iterasi. `unordered_map` atau `HashMap` dapat mengembalikan kuncinya dalam urutan lain sesudah satu penyisipan lagi, pada *compiler* atau JDK lain, atau pada *set* Python dari satu jalannya program ke berikutnya. Bila keluaran harus terurut, urutkanlah, atau pakai `std::map`, `TreeMap`, atau `sorted(dict)`.

Menyebut `std::map` sebagai tabel hash. Itu adalah pohon merah-hitam dengan operasi $O(\lg n)$. Pilih `std::unordered_map` bila hanya pencarian persis yang penting, dan `std::map` bila urutan atau rentang penting (Subbab 15.9).

Memakai `operator[]` untuk menguji keberadaan. Di C++, `if (m[key] == 0)` *menyisipkan* `key` dengan nilai bawaan bila kunci itu belum ada. Pakailah `m.find(key) != m.end()`, `m.count(key)`, atau, pada C++20, `m.contains(key)`.

Memercayai hash publik untuk kunci pilihan orang asing. Siapa pun yang mengetahui fungsinya dapat membuat setiap kunci bertabrakan (**Kode 15.10**). Pakailah hash berkunci dengan rahasia acak, atau struktur dengan batas terjamin.

Memakai hash tabel untuk kata sandi, atau hash kata sandi untuk tabel. Keduanya dibangun untuk tujuan yang berlawanan (Subbab 15.3.7).

15.14 Ringkasan Bab

- **Hashing** menghitung slot dari kunci itu sendiri alih-alih membandingkan kunci, sehingga lolos dari batas bawah $\lceil \lg(n+1) \rceil$ pencarian berbasis perbandingan: pencarian, penyisipan, dan penghapusan rata-rata memerlukan waktu $O(1)$. Hashing melepaskan urutan dan jaminan kasus terburuk (**Tabel 15.1**).
- **Pengalamatan langsung** memakai kunci sebagai indeks; sempurna untuk semesta bilangan bulat kecil dan mustahil untuk semesta besar. **Fungsi hash** $h: U \to \{0, \ldots, m-1\}$ memampatkan semesta yang besar ke m slot; dua kunci dengan slot yang sama **bertabrakan**.
- **Fungsi hash yang baik** bersifat deterministik, cepat, seragam pada data nyata, dan memakai seluruh kunci. **Metode pembagian** $k \bmod m$ menghendaki m prima; **metode perkalian**

berlaku untuk $m = 2^p$ dan mengambil bit-bit *teratas* dari $k \cdot \lfloor 2^w A \rfloor$; *string* memakai **hash polinomial** yang dihitung dengan aturan Horner, yang juga dapat **bergulir** sepanjang teks dengan $O(1)$ per jendela. Uji **avalanche** memperlihatkan seberapa baik sebuah fungsi mengaduk: pengaduk seperti *finalizer* SplitMix64 membalik sekitar separuh bit keluaran untuk setiap perubahan satu bit.

- **Tabrakan datang lebih awal**: menurut **batas ulang tahun**, n kunci acak dalam m slot bertabrakan dengan peluang sekitar $1 - e^{-n(n-1)/2m}$, peluang seimbang pada $n \approx 1.1774\sqrt{m}$ (23 untuk 365, sekitar lima miliar untuk 2^{64}).
- ***Separate chaining*** menyimpan satu *linked list* per *bucket*: pencarian melakukan sekitar $1 + \alpha/2$ perbandingan bila berhasil dan α bila gagal, dengan $\alpha = n/m$ **faktor muat**; panjang rantai mengikuti sebaran Poisson.
- **Pengalamatan terbuka** menyimpan kunci di dalam larik dan mengikuti **barisan *probe***: **linear** (*primary clustering*, $\frac{1}{2}(1 + 1/(1-\alpha)^2)$ *probe* gagal), **kuadratik** (*secondary clustering*; jarak segitiga menjangkau setiap slot tabel berukuran pangkat dua), atau ***double hashing*** (dekat dengan *uniform hashing*, $1/(1-\alpha)$). Penghapusan memerlukan **nisan**.
- **Pengukuran** pada 16.384 slot cocok dengan semua rumus ini; *linear probing* yang mendekati $\alpha = 1$ lambat sekaligus tak menentu.
- ***Rehashing*** dengan penggandaan menjaga α tetap terbatas dengan ongkos teramortisasi kurang dari dua pemindahan per penyisipan (terukur: 1,966).
- **Kasus terburuk** adalah $\Theta(n)$ per operasi, dan ***hash flooding*** memancingnya dengan sengaja; hashing **universal** dan **berkunci** (SipHash di Python, berkunci sejak dimulai) serta ***bin* yang menjadi pohon** (sejak Java 8) adalah pertahanannya.
- **Pustaka**: `std::unordered_map` adalah tabel hash berantai, `std::map` pohon merah-hitam; `HashMap` milik Java merantai ke tabel berukuran pangkat dua dan mengubah *bin* yang padat menjadi pohon; `dict` milik Python melakukan *probing* pada larik indeks yang ringkas dan menjaga urutan penyisipan (**Tabel 15.6**).
- **Penerapan**: penghitungan, deteksi duplikat, *two-sum* dan varian jumlah awalannya, *map* terbalik, dan *cache* (*hash map* ditambah *double linked list* untuk LRU).

15.15 Soal Latihan

15.1 Sisipkan kunci 5, 28, 19, 15, 20, 33, 12, 17, 10, dengan urutan ini, ke tabel hash dengan *separate chaining*, $m = 9$ *bucket* dan $h(k) = k \bmod 9$, dengan menyisipkan setiap kunci di depan rantainya. Gambarlah tabelnya. Berapa faktor muatnya, dan berapa rata-rata perbandingan pada pencarian berhasil?

15.2 Sisipkan kunci 10, 22, 31, 4, 15, 28, 17, 88, 59, dengan urutan ini, ke tabel berpengalamatan terbuka berisi $m = 11$ slot dengan $h_1(k) = k \bmod 11$, memakai (a) *linear probing*; (b) *quadratic probing* dengan $h(k,i) = (h_1(k) + i + 3i^2) \bmod 11$; (c) *double hashing* dengan $h_2(k) = 1 + (k \bmod 10)$. Perlihatkan setiap tabel akhir dan hitung *probe*-nya.

15.3 Hitunglah hash metode perkalian untuk $k = 123456$ dengan $m = 2^{14}$ dan $A = (\sqrt{5} - 1)/2$, mula-mula dengan Persamaan (15.3), lalu dengan bentuk bilangan bulat Persamaan (15.4) untuk $w = 32$, dengan $s = \lfloor 2^{32} A \rfloor = 2654435769$.

15.4 Hitunglah hash polinomial *string* “abc” dengan basis 31 modulo 101 menggunakan aturan Horner, dengan mereduksi pada setiap langkah. Kode a, b, c adalah 97, 98, 99.

15.5 Sebuah perpustakaan memberikan pengenal acak 32-bit kepada anggotanya dan tidak pernah memeriksa duplikat. Dengan Persamaan (15.8), perkirakan berapa anggota yang dapat didaftarkan sebelum peluang dua di antaranya berbagi pengenal mencapai 1%. Bagaimana dengan pengenal 64-bit?

15.6 Tulislah fungsi yang mengembalikan kata pertama dalam sebuah list yang muncul tepat sekali di list itu, dalam waktu harapan $O(n)$. Ujilah pada "kopi teh kopi susu teh jahe susu".

15.7 Tulislah fungsi yang mengelompokkan kata-kata yang saling anagram, dengan menjaga urutan kelompok menurut kata pertamanya. (Petunjuk: dua kata adalah anagram tepat ketika huruf-huruf terurutnya sama.) Ujilah pada kasur, rusak, kita, kiat, sukar, taki, buku, kubu.

15.8 Diberikan larik bilangan bulat tak terurut, temukan panjang deretan bilangan bulat berurutan terpanjang di dalamnya (untuk 100, 4, 200, 1, 3, 2, 101, 102, 103, 5 jawabannya 5, yaitu deretan 1 sampai 5) dalam waktu harapan $O(n)$, tanpa mengurutkan.

15.9 Banyaknya buku yang dipinjam pada delapan hari berturut-turut, dengan pengembalian dihitung sebagai koreksi negatif, adalah 3, 4, 7, 2, −3, 1, 4, 2. Berapa banyak rentang hari berurutan yang jumlahnya tepat 7? Selesaikan dalam waktu harapan $O(n)$ dengan jumlah awalan dan *hash map*.

15.10 Sebuah tabel diperkirakan menampung 1500 kunci. (a) Berapa ukuran pangkat dua m terkecil yang menjaga $\alpha \leq 0.75$? (b) Pada faktor muat itu, berapa *probe* yang diramalkan Persamaan (15.20) untuk pencarian gagal dengan *linear probing*, dan Persamaan (15.18) dengan *double hashing*? (c) Berapa perbandingan dengan *chaining*?

15.11 *Penghapusan geser mundur.* Dengan *linear probing*, penghapusan dapat menghindari nisan: sesudah mengosongkan slot j, pindailah slot-slot berikutnya dalam deretan itu; setiap kunci yang slot asalnya tidak terletak secara siklis dalam $(j, i]$, dengan i slotnya saat ini, dipindahkan mundur ke slot j, dan pemindaian berlanjut dari posisi lamanya. Terapkan ini pada tabel *linear probing* **Gambar 15.5**(a) ketika 1019 dihapus, dan jelaskan mengapa setiap kunci yang tersisa tetap dapat ditemukan.

15.12 Untuk setiap tugas, pilih `std::map` atau `std::unordered_map`, dan jelaskan: (a) mencetak katalog menurut urutan kode buku; (b) memeriksa apakah kode yang dipindai termasuk katalog, sejuta kali sehari; (c) mendaftar semua peminjaman yang tanggal jatuh temponya berada dalam tujuh hari ke depan; (d) menghitung berapa kali setiap pembaca meminjam buku tahun ini, dicetak dalam urutan apa pun.

15.13 Sebuah tabel berantai dimulai dengan $m = 1$ dan melakukan *rehash* ketika $n > m$. (a) Tunjukkan bahwa bila tabel menggandakan m pada setiap *rehash*, total kunci yang dipindahkan selama menyisipkan n kunci adalah $O(n)$, kurang dari sekitar $2n$. (b) Tunjukkan bahwa bila tabel justru menambah 1000 *bucket* pada setiap *rehash*, totalnya $\Theta(n^2)$.

15.14 (a) Mengapa hash berkunci dengan rahasia yang dipilih saat program dimulai mematahkan serangan **Kode 15.10**, sedangkan rahasia yang ditanam di kode sumber tidak? (b) *Bin* Java yang menjadi pohon mengurutkan kunci-kunci yang bertabrakan dengan `compareTo`. Apa yang terjadi pada *bin* pohon yang kuncinya semua memiliki `hashCode` sama dan *tidak* mengimplementasikan `Comparable`?

15.15 Di Python, mengapa `hash(-1)` bernilai −2? Mengapa **Kode 15.13** mencetak *set* berisi empat kode dalam urutan berbeda di bawah setiap `PYTHONHASHSEED`, sedangkan `dict` berisi kode-kode yang sama selalu tercetak dalam urutan penyisipan?

15.16 Penyelesaian Soal Latihan

Soal hitung tangan 15.1 sampai 15.4 diperiksa oleh **Kode 15.15**, yang juga memuat penyelesaian soal pemrograman 15.6 sampai 15.9; `Exercises.java` dan `exercises.py` mencetak keluaran yang sama.

Kode 15.15 `exercises.cpp`: pemeriksaan Soal Latihan 15.1 sampai 15.4 dan penyelesaian Soal Latihan 15.6 sampai 15.9.

```cpp
// exercises.cpp -- checks for the hand exercises and solutions to the
// programming exercises of Chapter 15.
#include <algorithm>
#include <cmath>
#include <cstdint>
#include <cstdio>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <vector>

// Exercise 15.1: chaining, h(k) = k mod 9, insertion at the front of a chain.
void exercise1() {
    const int keys[9] = {5, 28, 19, 15, 20, 33, 12, 17, 10};
    std::vector<std::vector<int>> t(9);
    for (int k : keys) t[k % 9].insert(t[k % 9].begin(), k);
    std::printf("15.1 chaining, m = 9\n");
    for (int b = 0; b < 9; ++b) {
        std::printf("  %d:", b);
        for (int k : t[b]) std::printf(" %d", k);
        std::printf("\n");
    }
}

// Exercise 15.2: open addressing in m = 11 slots with three probe sequences.
int probe2(int k, int i, int mode) {
    if (mode == 0) return (k + i) % 11;                    // linear
    if (mode == 1) return (k + i + 3 * i * i) % 11;        // quadratic
    return (k + i * (1 + k % 10)) % 11;                    // double
}

void exercise2() {
    const int keys[9] = {10, 22, 31, 4, 15, 28, 17, 88, 59};
    const char *name[3] = {"linear   ", "quadratic", "double   "};
    std::printf("15.2 open addressing, m = 11 (- = empty)\n");
    for (int mode = 0; mode < 3; ++mode) {
        std::vector<int> t(11, -1);
        int probes = 0;
        for (int k : keys) {
            for (int i = 0; i < 11; ++i) {
                ++probes;
                int s = probe2(k, i, mode);
                if (t[s] < 0) { t[s] = k; break; }
            }
        }
        std::printf("  %s:", name[mode]);
```

Kode 15.15 (Lanjutan) *exercises.cpp: pemeriksaan Soal Latihan 15.1 sampai 15.4 dan penyelesaian Soal Latihan 15.6 sampai 15.9.*

```
        for (int x : t) {
            if (x < 0) std::printf("  -");
            else std::printf(" %2d", x);
        }
        std::printf("   (%d probes)\n", probes);
    }
}

// Exercise 15.3: the multiplication method for k = 123456, m = 2^14.
void exercise3() {
    const double A = (std::sqrt(5.0) - 1.0) / 2.0;
    double prod = 123456.0 * A;
    int h = (int)std::floor(16384.0 * (prod - std::floor(prod)));
    std::uint32_t s = 2654435769u;                          // floor(2^32 * A)
    std::uint32_t r0 = (std::uint32_t)123456u * s;          // low 32 bits of k*s
    std::printf("15.3 k*A = %.6f, h = %d; integer form: r0 = %u, h = %u\n", prod, h,
                r0, r0 >> 18);
}

// Exercise 15.4: Horner's rule for "abc", base 31, modulo 101.
void exercise4() {
    long h = 0;
    std::printf("15.4 ");
    for (char c : std::string("abc")) {
        h = (h * 31 + c) % 101;
        std::printf("%ld ", h);
    }
    std::printf("-> h(\"abc\") = %ld\n", h);
}

// Exercise 15.6: the first word that occurs exactly once.
std::string firstUnique(const std::vector<std::string> &w) {
    std::unordered_map<std::string, int> count;
    for (const std::string &x : w) ++count[x];              // pass 1: count
    for (const std::string &x : w)                          // pass 2: first with 1
        if (count[x] == 1) return x;
    return "";
}
```

Kode 15.15 (Lanjutan) *exercises.cpp: pemeriksaan Soal Latihan 15.1 sampai 15.4 dan penyelesaian Soal Latihan 15.6 sampai 15.9.*

```
// Exercise 15.7: group words that are anagrams of each other.
std::vector<std::vector<std::string>>
groupAnagrams(const std::vector<std::string> &w) {
    std::unordered_map<std::string, std::size_t> groupOf;   // letters -> group
    std::vector<std::vector<std::string>> groups;
    for (const std::string &x : w) {
        std::string key = x;
        std::sort(key.begin(), key.end());
        auto it = groupOf.find(key);
        if (it == groupOf.end()) {
            groupOf[key] = groups.size();
            groups.push_back({x});
        } else {
            groups[it->second].push_back(x);
        }
    }
    return groups;
}

// Exercise 15.8: the longest run of consecutive integers, in O(n) expected time.
int longestRun(const std::vector<int> &a) {
    std::unordered_set<int> s(a.begin(), a.end());
    int best = 0;
    for (int x : s) {
        if (s.count(x - 1)) continue;                  // x does not start a run
        int len = 1;
        while (s.count(x + len)) ++len;
        best = std::max(best, len);
    }
    return best;
}

// Exercise 15.9: how many contiguous stretches of days add up to exactly k?
int countSum(const std::vector<int> &a, int k) {
    std::unordered_map<long, int> seen{{0, 1}};             // prefix sum -> times
    long prefix = 0;
    int count = 0;
    for (int x : a) {
        prefix += x;
        auto it = seen.find(prefix - k);
        if (it != seen.end()) count += it->second;
        ++seen[prefix];
    }
    return count;
}
```

Kode 15.15 (Lanjutan) *exercises.cpp: pemeriksaan Soal Latihan 15.1 sampai 15.4 dan penyelesaian Soal Latihan 15.6 sampai 15.9.*

```
int main() {
    exercise1();
    exercise2();
    exercise3();
    exercise4();

    std::printf("15.6 first unique: %s\n",
                firstUnique({"kopi", "teh", "kopi", "susu", "teh", "jahe", "susu"})
                    .c_str());
    std::printf("15.7 anagram groups:");
    for (const auto &g : groupAnagrams({"kasur", "rusak", "kita", "kiat", "sukar",
                                        "taki", "buku", "kubu"})) {
        std::printf(" [");
        for (std::size_t i = 0; i < g.size(); ++i)
            std::printf("%s%s", i ? " " : "", g[i].c_str());
        std::printf("]");
    }
    std::printf("\n15.8 longest run: %d\n",
                longestRun({100, 4, 200, 1, 3, 2, 101, 102, 103, 5}));
    std::printf("15.9 stretches summing to 7: %d\n",
                countSum({3, 4, 7, 2, -3, 1, 4, 2}, 7));
    return 0;
}
```

```
15.1 chaining, m = 9
  0:
  1: 10 19 28
  2: 20
  3: 12
  4:
  5: 5
  6: 33 15
  7:
  8: 17
15.2 open addressing, m = 11 (- = empty)
  linear   : 22 88  -  -  4 15 28 17 59 31 10   (16 probes)
  quadratic: 22  - 88 17  4  - 28 59 15 31 10   (23 probes)
  double   : 22  - 59 17  4 15 28 88  - 31 10   (16 probes)
15.3 k*A = 76300.004115, h = 67; integer form: r0 = 17612864, h = 67
15.4 97 75 0 -> h("abc") = 0
15.6 first unique: jahe
15.7 anagram groups: [kasur rusak sukar] [kita kiat taki] [buku kubu]
15.8 longest run: 5
15.9 stretches summing to 7: 4
```

15.1 *Bucket* asalnya adalah 5 mod 9 = 5, 28 mod 9 = 1, 19 → 1, 15 → 6, 20 → 2, 33 → 6, 12 → 3, 17 → 8, 10 → 1. Dengan penyisipan di depan, *bucket* 1 berisi 10, 19, 28 (yang terakhir disisipkan paling depan), *bucket* 6 berisi 33, 15, dan *bucket* 2, 3, 5, 8 berisi 20, 12, 5, 17; *bucket* 0, 4, 7 kosong, seperti diperlihatkan keluaran. Faktor muatnya $9/9 = 1$. Pencarian berhasil membandingkan dengan kunci itu dan semua kunci di depannya: 1 untuk masing-masing dari enam kunci di depan sebuah rantai, 2 untuk 19 dan untuk 15, 3 untuk 28, seluruhnya $6 + 2 + 2 + 3 = 13$, rata-rata $13/9 \approx 1.44$, dibandingkan $1 + (n-1)/(2m) = 1 + 8/18 \approx 1.44$ dari Persamaan (15.12).

15.2 Slot asal modulo 11 adalah 10, 0, 9, 4, 4, 6, 6, 0, 4. (a) Linear: 10, 22, 31, 4 masuk ke slot asalnya; 15 mendapati 4 terisi dan mengambil 5; 28 mengambil 6; 17 mencoba 6 dan mengambil 7; 88 mencoba 0 dan mengambil 1; 59 mencoba 4, 5, 6, 7 dan mengambil 8. Tabel akhir, slot 0 sampai 10: 22, 88, –, –, 4, 15, 28, 17, 59, 31, 10; 16 *probe*. (b) Kuadratik, dengan jarak $i + 3i^2 =$ 0,4,14,30,52, …: 15 mencoba 4, lalu $4 + 4 = 8$, bebas; 28 pulang ke 6; 17 mencoba 6, 10, 9, lalu $6 + 30 = 36 \bmod 11 = 3$; 88 mencoba 0, 4, 3, 8, 8, 3, 4, 0 dan akhirnya 2; 59 mencoba 4, 8 dan

mengambil 7. Tabel akhir: 22, –, 88, 17, 4, –, 28, 59, 15, 31, 10; 23 *probe*. Pengulangan pada barisan 88 memperlihatkan bahwa barisan kuadratik ini tidak mengunjungi setiap slot tabel berisi 11 slot, sehingga dapat gagal menemukan slot bebas pada tabel yang lebih penuh. (c) Ganda: 15 memiliki $h_2 = 6$ dan mencoba 4, 10, lalu $16 \bmod 11 = 5$; 17 memiliki $h_2 = 8$ dan mencoba 6, lalu 3; 88 memiliki $h_2 = 9$ dan mencoba 0, 9, lalu $18 \bmod 11 = 7$; 59 memiliki $h_2 = 10$ dan mencoba 4, 3, lalu 2. Tabel akhir: 22, –, 59, 17, 4, 15, 28, 88, –, 31, 10; 16 *probe*. Ketiga tabel cocok dengan keluaran.

15.3 $kA = 123456 \times 0.6180339887\ldots = 76300.0041151\ldots$; bagian pecahannya $0.0041151\ldots$; $16384 \times 0.0041151\ldots = 67.42\ldots$, jadi $h(k) = 67$. Dalam bentuk bilangan bulat, $k \cdot s = 327706022297664 = 76300 \cdot 2^{32} + 17612864$, sehingga 32 bit rendahnya $r_0 = 17612864$, dan 14 bit teratasnya $\lfloor 17612864/2^{18} \rfloor = 67$. Kedua bentuk sepakat.

15.4 $h = 97 \bmod 101 = 97$; lalu $(97 \cdot 31 + 98) \bmod 101 = 3105 \bmod 101 = 75$; lalu $(75 \cdot 31 + 99) \bmod 101 = 2424 \bmod 101 = 0$. Jadi $h(\text{"abc"}) = 0$, karena $2424 = 24 \cdot 101$.

15.5 Tetapkan $1 - e^{-n^2/2m} = 0.01$, sehingga $n^2/(2m) = -\ln 0.99 \approx 0.01005$ dan $n \approx \sqrt{0.0201 \cdot 2^{32}} \approx \sqrt{8.63 \times 10^7} \approx 9300$. Peluang duplikat 1% sudah muncul sesudah sekitar 9300 anggota, jumlah yang mudah dilampaui perpustakaan universitas. Dengan pengenal 64-bit, $n \approx \sqrt{0.0201 \cdot 1.84 \times 10^{19}} \approx 6.1 \times 10^8$, lebih dari enam ratus juta anggota.

15.6 `firstUnique` melakukan dua lintasan: yang pertama menghitung setiap kata dalam *hash map*, yang kedua mengembalikan kata pertama yang hitungannya 1. Kedua lintasan berwaktu harapan $O(n)$. Untuk contohnya, hitungannya kopi 2, teh 2, susu 2, jahe 1, dan jawabannya *jahe*.

15.7 `groupAnagrams` memakai huruf-huruf kata dalam urutan terurut sebagai kunci: *kasur*, *rusak*, dan *sukar* semuanya menghasilkan "aksru". *Hash map* mengirim setiap kunci ke indeks kelompoknya dalam sebuah *vector*, yang menjaga urutan kemunculan pertama kelompok-kelompok. Mengurutkan kata sepanjang L berongkos $O(L \lg L)$, sehingga totalnya harapan $O(\sum L \lg L)$. Kelompoknya adalah [kasur rusak sukar] [kita kiat taki] [buku kubu].

15.8 Masukkan semua nilai ke *hash set*. Nilai x memulai sebuah deretan tepat ketika $x - 1$ tidak ada dalam himpunan; hanya dari awal seperti itulah `longestRun` menghitung naik selama $x +$ len ada. Setiap nilai dihitung dalam paling banyak satu penelusuran naik (yang dimulai dari awal deretannya), sehingga total pekerjaan harapan $O(n)$, walaupun ada dua gelung bersarang. Untuk contohnya, awal-awalnya adalah 100, 200, dan 1, dengan deretan sepanjang 4, 1, dan 5.

15.9 Misalkan P_j jumlah j hari pertama, dengan $P_0 = 0$. Hari $i + 1$ sampai j berjumlah $P_j - P_i$, yang sama dengan 7 tepat ketika $P_i = P_j - 7$. Jadi telusuri hari-harinya, simpan *hash map* dari setiap jumlah awalan ke banyaknya kemunculannya, dan pada setiap j tambahkan banyaknya jumlah awalan sebelumnya yang sama dengan $P_j - 7$. Jumlah-jumlah awalannya 0, 3, 7, 14, 16, 13, 14, 18, 20, dan keempat rentangnya adalah hari 1 sampai 2 (3 + 4), hari 3 (7), hari 3 sampai 6 (7 + 2 − 3 + 1), dan hari 6 sampai 8 (1 + 4 + 2). Nilai negatif adalah alasan mengapa jendela dua pointer tidak berlaku di sini sedangkan *hash map* berlaku.

15.10 (a) $1500/0.75 = 2000$, jadi $m = 2048$, sehingga $\alpha = 1500/2048 \approx 0.732$. (b) *Linear probing*: $\frac{1}{2}(1 + 1/(1 - 0.7324)^2) = \frac{1}{2}(1 + 1/0.0716) \approx 7.48$ *probe*; *double hashing*: $1/(1 - 0.7324) \approx 3.74$ *probe*. (c) *Chaining*: sekitar $\alpha \approx 0.73$ perbandingan, ditambah perhitungan hash $O(1)$.

15.11 Pada **Gambar 15.5**(a) deretannya adalah 1003, 1019, 1035, 1051 di slot 11 sampai 14, semuanya dengan slot asal 11. Menghapus 1019 mengosongkan slot 12. Pemindaian memeriksa

slot 13: 1035 berasal dari 11, yang tidak berada dalam selang siklis (12,13], jadi 1035 pindah mundur ke 12, dan slot 13 menjadi celah. Di slot 14, 1051 berasal dari 11, tidak dalam (13,14], jadi ia pindah ke 13 dan slot 14 menjadi celah. Slot 15 kosong, yang mengakhiri deretan. Hasil akhirnya: 1003, 1035, 1051 di slot 11 sampai 13. Sebuah kunci dipindahkan mundur hanya bila slot asalnya terletak di atau sebelum celah, sehingga sesudah dipindahkan ia tetap terjangkau dari slot asalnya melalui slot-slot terisi yang berurutan; kunci yang slot asalnya terletak sesudah celah tetap di tempatnya, karena pencarian terhadapnya tidak pernah melewati celah itu. Jadi tidak ada barisan pencarian yang terpotong, dan tidak diperlukan nisan. (Metode ini bergantung pada langkah tetap *linear probing*; ia tidak berlaku untuk *quadratic probing* atau *double hashing*.)

15.12 (a) `std::map`: iterasi *in-order*-nya adalah urutan kode, tanpa pengurutan. (b) `std::unordered_map` (atau `unordered_set`): hanya keanggotaan persis yang diperlukan, dalam harapan $O(1)$. (c) `std::map` yang dikunci tanggal jatuh tempo (atau `std::multimap` bila tanggal berulang): tujuh hari ke depan adalah sebuah rentang, ditemukan dengan `lower_bound` dan `upper_bound`. (d) `std::unordered_map<reader, int>`: penghitungan dengan kunci persis, urutan keluaran tidak penting.

15.13 (a) *Rehash* terjadi ketika tabel berisi 2^j *bucket* menerima kunci ke-$(2^j + 1)$, yaitu ketika n mencapai 2,3,5,9,17, ..., dan *rehash* itu memindahkan $2^j + 1$ kunci tersebut. Bila 2^J adalah ukuran tabel terbesar yang pernah meluap, maka $2^J < n$ dan totalnya $\sum_{j=0}^{J}(2^j + 1) = 2^{J+1} - 1 + (J + 1) < 2n + \lg n + 1$, yaitu $O(n)$, kurang dari sekitar dua pemindahan per penyisipan, seperti pada Persamaan (15.23). (b) Dengan m bertambah 1000, *rehash* terjadi kira-kira setiap 1000 penyisipan, dengan *rehash* ke-t memindahkan sekitar $1000t$ kunci; mencapai n kunci memerlukan sekitar $n/1000$ kali *rehash*, yang seluruhnya memindahkan sekitar $1000(1 + 2 + \cdots + n/1000) \approx n^2/2000$ kunci, yaitu $\Theta(n^2)$: rata-rata $\Theta(n)$ pemindahan per penyisipan.

15.14 (a) Kunci-kunci serangan dihitung untuk satu fungsi yang diketahui. Rahasia yang dipilih secara acak ketika program dimulai (atau per tabel) tidak diketahui penyerang, dan menurut Persamaan (15.25) setiap pasangan kunci tetap bertabrakan dengan peluang paling besar $(L - 1)/p$ atas pemilihan acak rahasia itu, sehingga daftar kunci yang dihitung sebelumnya tidak berguna. Rahasia yang ditulis di kode sumber sama di setiap salinan program; siapa pun yang membaca kode atau berkas binernya dapat menghitung tabrakan untuknya, dan serangan itu kembali berlaku. (b) Tanpa `compareTo`, pohon tidak dapat mengurutkan kunci-kunci dengan hash yang sama. Penyisipan lalu mengurutkannya dengan pemecah seri yang sembarang (nama kelas dan kode hash identitasnya), yang tidak dapat direproduksi pencarian dari kuncinya, sehingga pencarian mungkin harus memeriksa kedua subpohon pada setiap simpul, dan *bin* itu kembali berperilaku seperti list: $O(\ell)$ untuk *bin* berisi ℓ entri. Pertahanan itu bekerja sepenuhnya hanya untuk kunci seperti `String` yang `Comparable`.

15.15 Dalam antarmuka C milik CPython, fungsi hash mengembalikan -1 untuk menandakan terjadinya galat, sehingga tidak ada objek yang boleh memiliki hash -1; bilangan bulat -1 diberi -2 sebagai gantinya. `set` adalah tabel ber-*probing* biasa tanpa larik `entries` terpisah, sehingga iterasinya menyusuri slot-slot menurut urutan indeks, dan slot-slot itu bergantung pada nilai hash, yang untuk *string* berubah bersama kunci acaknya; `dict` beriterasi atas larik `entries`-nya yang berurutan menurut penyisipan, yang sama sekali tidak bergantung pada nilai hash (**Gambar 15.10**).

Daftar Pustaka

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998, sec. 6.4.

[2] W. W. Peterson, "Addressing for random-access storage," *IBM Journal of Research and Development*, vol. 1, no. 2, pp. 130–146, Apr. 1957.

[3] Oracle, "Class String: hashCode()," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/String.html. [Accessed: 24-Sep-2026].

[4] cppreference.com, "std::unordered_map." [Online]. Available: https://en.cppreference.com/w/cpp/container/unordered_map. [Accessed: 24-Sep-2026].

[5] ISO/IEC 14882:2017, *Programming Languages — C++*, 5th ed. Geneva, Switzerland: ISO, 2017, sec. 26.2.7 (unordered associative containers).

[6] Oracle, "Class HashMap<K,V>," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/HashMap.html. [Accessed: 24-Sep-2026].

[7] Python Software Foundation, "Objects/dictobject.c," CPython source repository. [Online]. Available: https://github.com/python/cpython/blob/main/Objects/dictobject.c. [Accessed: 24-Sep-2026].

[8] OpenJDK, "src/java.base/share/classes/java/util/HashMap.java," JDK 25 update repository. [Online]. Available: https://github.com/openjdk/jdk25u/blob/master/src/java.base/share/classes/java/util/HashMap.java. [Accessed: 24-Sep-2026].

[9] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *Proc. 12th USENIX Security Symposium*, Washington, DC, USA, Aug. 2003, pp. 29–44.

[10] A. Klink and J. Wälde, "Efficient denial of service attacks on web application platforms," presented at the 28th Chaos Communication Congress (28C3), Berlin, Germany, Dec. 2011. [Online]. Available: https://fahrplan.events.ccc.de/congress/2011/Fahrplan/attachments/2007_28C3_Effective_DoS_on_web_application_platforms.pdf

[11] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, Apr. 1979.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 11.

[13] J.-P. Aumasson and D. J. Bernstein, "SipHash: A fast short-input PRF," in *Progress in Cryptology – INDOCRYPT 2012*, LNCS, vol. 7668. Berlin, Germany: Springer, 2012, pp. 489–508.

[14] C. Heimes, "PEP 456 – Secure and interchangeable hash algorithm," Python Enhancement Proposals, 2013. [Online]. Available: https://peps.python.org/pep-0456/. [Accessed: 24-Sep-2026].

[15] Python Software Foundation, "What's New In Python 3.11: Other language changes (siphash13, bpo-29410)," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/whatsnew/3.11.html. [Accessed: 24-Sep-2026].

[16] M. Duigou and B. Christian, "JEP 180: Handle frequent HashMap collisions with balanced trees," OpenJDK, delivered in JDK 8. [Online]. Available: https://openjdk.org/jeps/180. [Accessed: 24-Sep-2026].

[17] R. Hettinger, "More compact dictionaries with faster iteration," python-dev mailing list, Dec. 10, 2012. [Online]. Available: https://mail.python.org/pipermail/python-dev/2012-December/123028.html

[18] Python Software Foundation, "What's New In Python 3.7: Summary — release highlights (insertion-order preservation of dict objects)," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/whatsnew/3.7.html. [Accessed: 24-Sep-2026].

[19] Profitina, "Profitina — AI Visibility (GEO) platform." [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

Bab 16

Evaluasi dan Pembahasan IV: Graf, Heap, dan Tabel Hash dalam Praktik

Bab terakhir ini tidak menambahkan struktur baru. Bab ini mengumpulkan semua alat yang ada di dalam buku, dengan titik berat pada tiga yang paling baru, yaitu graf dari Bab 13, *heap* dan *priority queue* dari Bab 14, serta tabel hash dari Bab 15, lalu mempekerjakannya pada sembilan soal. Setiap soal diselesaikan dari halaman kosong, selangkah kecil demi selangkah kecil, dalam C++, Java, dan Python. Dua di antaranya membentuk satu pasangan: melengkapi kelas *max-heap* dengan lima operasi baru, dan menyusun setumpuk tiket perjalanan menurut urutan pemakaiannya dengan *hash map*. Tujuh soal lainnya baru, dan masing-masing membidik satu gagasan yang sering terasa licin bagi pemula: jumlah lompatan paling sedikit antara dua tempat, urutan membaca topik-topik yang saling bergantung, rute tercepat ketika jalan memiliki panjang, *k* nilai terbesar dari sebuah aliran data yang panjang, nilai tengah dari aliran data yang terus bertambah, tabel hash yang dibangun dengan tangan lengkap dengan rantainya, serta urusan pelik menghapus dari tabel yang menyimpan kuncinya di dalam larik itu sendiri. Untuk setiap soal Anda akan menggambar strukturnya sebelum menulis kode, menelusuri program dengan tangan, membaca tiga program lengkap beserta keluaran yang benar-benar dicetaknya, mempelajari satu penyelesaian keliru yang menggoda dan melihat persis bagaimana penyelesaian itu gagal, lalu menutup dengan varian *uji diri* beserta jawabannya. Bab ini diakhiri dengan pertanyaan yang telah dipersiapkan oleh seluruh buku: berhadapan dengan tugas nyata, *struktur data mana yang sebaiknya saya pilih?* Kita menjawabnya dengan tabel keputusan yang merangkum kelima belas bab dan dengan pengukuran setiap struktur pada sepuluh ribu buku yang sama dari katalog Pustaka Ceria.

16.1 Titik Tolak Kita

Lima belas bab telah kita lalui, dan ada baiknya kita menyebut apa yang diberikan masing-masing, karena separuh dari menyelesaikan soal adalah mengenali alat apa yang diminta soal itu.

Bagian pertama buku membahas larik dan apa yang dapat dikerjakan dengannya. Bab 1 menyimpan katalog Pustaka Ceria di dalam larik dan menunjukkan mengapa membaca elemen ke-i berongkos sama untuk setiap i, sedangkan menyisipkan di tengah menuntut penggeseran semua elemen sesudahnya. Bab 2 membahas pencarian: pencarian sekuensial menyusuri larik, pencarian biner membelah dua larik yang *terurut* pada setiap langkah dan memerlukan paling banyak $\lfloor \lg n \rfloor + 1$ kali pemeriksaan. Bab 3 membahas pengurutan, mulai dari pengurutan elementer, *merge sort*, *quicksort*, perkenalan pertama dengan *heap sort*, sampai pengurutan dengan pencacahan. Bab 4 mempekerjakan gagasan-gagasan itu.

Bagian kedua mengganti “sebaris sel” dengan “kotak-kotak yang menunjuk ke kotak lain”. Bab 5 membangun *stack* (terakhir masuk, pertama keluar) dan *queue* (pertama masuk, pertama keluar). Bab 6 menjelaskan pointer, dan Bab 7 memakainya untuk membangun *singly linked list*; Bab 8 mempekerjakan *stack*, pointer, dan *list*.

Bagian ketiga menambahkan arah dan hierarki. Bab 9 memberi setiap simpul tautan kedua, ke belakang, dan menutup *list* menjadi lingkaran. Bab 10 mengajarkan rekursi, yaitu fungsi yang menyelesaikan soal dengan menyelesaikan salinan yang lebih kecil dari soal itu. Bab 11 menumbuhkan pohon pencarian biner (*binary search tree*, BST): setiap kunci di subpohon kiri sebuah simpul lebih kecil daripada kunci simpul itu dan setiap kunci di subpohon kanannya lebih besar, sehingga pencarian menuruni satu lintasan alih-alih menyusuri seluruh koleksi. Bab 12 mempekerjakan semuanya.

Bagian keempat, yang menjadi pusat perhatian bab ini, memberi kita tiga struktur yang bersama-sama menggerakkan sebagian besar perangkat lunak di dunia.

- **Bab 13, graf.** *Graf* adalah himpunan *simpul* (*vertex*) yang dihubungkan oleh *sisi* (*edge*); sisi dapat berarah dan dapat berbobot. Kita menyimpan graf sebagai matriks ketetanggaan (*adjacency matrix*) atau sebagai *adjacency list*, lalu menjelajahinya dengan dua cara: penelusuran melebar (*breadth-first search*, BFS), yang mengunjungi simpul-simpul dalam cincin-cincin berjarak makin jauh dengan bantuan *queue* sehingga menemukan rute dengan sisi paling sedikit, dan penelusuran mendalam (*depth-first search*, DFS), yang masuk sedalam mungkin sebelum mundur, dengan bantuan *stack* atau rekursi. Di atas kedua penelusuran itu kita membangun komponen terhubung, pengurutan topologis untuk graf tanpa siklus berarah, dan versi pertama yang lembut dari algoritma Dijkstra untuk rute terpendek ketika sisi memiliki panjang.
- **Bab 14, *heap* dan *priority queue*.** *Priority queue* selalu menyerahkan butir yang paling mendesak lebih dahulu. *Binary heap* mewujudkannya di dalam larik biasa yang kita baca sebagai pohon biner *complete*: anak-anak indeks i berada di $2i + 1$ dan $2i + 2$, dan setiap induk paling sedikit sebesar anak-anaknya (*max-heap*) atau paling banyak sebesar anak-anaknya (*min-heap*). Dua operasi perbaikan, *sift-up* dan *sift-down*, masing-masing berongkos paling banyak setinggi pohon, yaitu $\lfloor \lg n \rfloor$; sebuah *heap* utuh dapat dibangun dari larik tak terurut dalam waktu linear; dan pustaka standar ketiga bahasa menyediakannya.
- **Bab 15, hashing.** *Fungsi hash* mengubah kunci menjadi indeks larik, sehingga *tabel hash* dapat menemukan kunci dengan menghitung di mana kunci itu harus berada alih-alih mencarinya. Dua kunci yang jatuh di tempat yang sama disebut *bertabrakan* (*collide*); *separate chaining* menyimpan sebuah *linked list* kecil di setiap slot, sedangkan *open addressing* memindahkan kunci kedua ke slot lain di larik yang sama. Selama tabel dijaga tidak terlalu penuh, pencarian, penyisipan, dan penghapusan masing-masing berongkos konstan secara rata-rata.

Bab ini memakai semua alat tersebut, sering kali di dalam satu program. **Tabel 16.1** mendaftar kesembilan soalnya, subbab tempat setiap soal diselesaikan, dan bab yang memasok gagasan utamanya. Subbab 16.8 kemudian mengambil jarak dan membandingkan semua struktur di dalam buku pada satu tugas nyata.

Tabel 16.1 Sembilan soal dalam bab ini, subbab yang menyelesaikan masing-masing, dan bab yang memasok gagasan utamanya.

Soal	Subbab	Yang diminta	Alat utama dari
16.1	16.3	Melengkapi kelas *max-heap*: `extractMax`, `increaseKey`, `getMin`, `buildMaxHeap`, `isMaxHeap`	Bab 3, 11, 14
16.2	16.4	Menyusun setumpuk tiket perjalanan menurut urutannya, dengan *hash map*	Bab 7, 15
16.3	16.5	Lompatan paling sedikit antarcabang perpustakaan	Bab 5, 13
16.4	16.5	Urutan membaca topik-topik yang saling bergantung	Bab 5, 13
16.5	16.5	Rute tercepat untuk mobil buku	Bab 13, 14
16.6	16.6	*k* nilai terbesar dari daftar yang panjang, dengan *heap* berukuran *k*	Bab 14
16.7	16.6	Median sejauh ini, dengan dua *heap*	Bab 14
16.8	16.7	Tabel hash dengan *separate chaining*, dibangun dengan tangan	Bab 7, 15
16.9	16.7	*Open addressing*, dan mengapa penghapusan memerlukan *tombstone*	Bab 15

Contoh berjalan kita masih perpustakaan kecil *Pustaka Ceria*. Buku-bukunya memiliki kode seperti `B1001`, perpustakaan ini memiliki cabang di beberapa kawasan Surabaya yang dihubungkan oleh mobil buku, dan, untuk pengukuran di Subbab 16.6 dan 16.8, memiliki katalog nyata berisi sepuluh ribu buku yang diperkenalkan Bab 2.

> **WAWASAN KUNCI: SATU GAGASAN DI BALIK HAMPIR SETIAP SOAL DI SINI.** Bab 8 mendapati bahwa sebagian besar galat pointer melanggar satu aturan: pertahankan jalan menuju semua yang masih diperlukan sebelum mengubah sebuah tautan. Struktur-struktur dalam bab ini memiliki aturan sejenis, yaitu: *setiap struktur menjanjikan sesuatu, sebuah invarian, dan setiap operasi harus membiarkan janji itu tetap benar. Heap* menjanjikan bahwa tidak ada anak yang lebih besar daripada induknya; `extractMax`, `increaseKey`, dan `buildMaxHeap` masing-masing adalah perbaikan kecil atas janji yang dirusak oleh satu perubahan. BFS menjanjikan bahwa simpul keluar dari *queue* menurut urutan jaraknya, sehingga saat pertama kali melihat sebuah simpul kita sudah tahu jaraknya. Dijkstra menjanjikan bahwa simpul yang diambil dari *heap* telah memperoleh jarak akhirnya. Tabel hash menjanjikan bahwa sebuah kunci berada di slot yang ditunjuk hash-nya atau lebih jauh di sepanjang lintasan yang akan diikuti pencarian. Bila sebuah program dalam bab ini keliru, hampir selalu penyebabnya adalah sebuah operasi yang meninggalkan janji itu dalam keadaan rusak, dan penyelesaian-penyelesaian keliru di bawah merusaknya dengan semua cara tersebut.

16.2 Cara Menghadapi Soal tentang Sebuah Struktur

Metode dari Bab 8, yaitu menyatakan ulang, memilih, menggambar, menulis invarian, mendaftar kasus tepi, mengubah dalam urutan yang aman, lalu menguji, berlaku tanpa perubahan. Struktur-struktur dalam bab ini menambahkan dua kebiasaan tersendiri, sehingga metodenya bertambah dua langkah menjadi sembilan.

Langkah 1 — Nyatakan ulang soal dengan kata-kata sendiri, disertai satu contoh. Tuliskan apa yang masuk dan apa yang keluar, lalu kerjakan satu masukan kecil dengan tangan. Jika Anda belum dapat mengerjakan contohnya dengan tangan, Anda belum siap menulis kode.

Langkah 2 — Pilih strukturnya, dan sebutkan alasannya dalam satu kalimat. “Selalu layani yang terbesar” berarti *heap*. “Cari berdasarkan nama” berarti tabel hash. “Benda-benda yang

terhubung dengan benda lain" berarti graf. Kalimat itu memberi tahu operasi apa yang diperlukan, dan Subbab 16.8 mengubah kalimat-kalimat semacam itu menjadi sebuah tabel.

Langkah 3 — Gambarlah. *Heap* digambar dua kali: sebagai pohon, tempat janjinya mudah dilihat, dan sebagai larik yang sebenarnya, dengan indeks di bawah setiap sel. Graf digambar sebagai lingkaran dan garis, dengan *adjacency list* ditulis di sampingnya dalam urutan yang disimpan program. Tabel hash digambar sebagai sekolom slot beserta rantai atau lintasan *probe*-nya.

Langkah 4 — Tuliskan invariannya. Untuk *max-heap*: "`heap[(i-1)/2] >= heap[i]` untuk setiap $i \geq 1$". Untuk BFS: "*queue* memuat simpul-simpul menurut urutan jarak, dan jarak-jarak di dalamnya berselisih paling banyak satu". Untuk tabel hash berantai: "kunci k, jika ada, berada di rantai slot $h(k)$".

Langkah 5 — Daftarkan kasus tepi. Untuk *heap*: *heap* kosong, satu elemen, kunci yang sama, indeks di luar larik. Untuk graf: simpul terpencil, simpul yang tidak dapat dicapai, siklus, sisi dari sebuah simpul ke dirinya sendiri. Untuk tabel hash: kunci yang tidak ada, dua kunci yang bertabrakan, kunci yang disisipkan dua kali, penghapusan yang diikuti pencarian.

Langkah 6 — Atur urutan perubahan dengan aman. Simpan akar sebelum menimpanya; tandai simpul sebagai sudah terlihat sebelum ia menunggu di dalam *queue*; lepaskan simpul dari rantainya sebelum membebaskannya.

Langkah 7 — Hitung ongkosnya. Langkah ini baru dalam bab ini, karena seluruh alasan keberadaan ketiga struktur ini adalah kecepatan. Tuliskan berapa kali setiap gelung berjalan dalam kasus terburuk, lalu bandingkan dengan penyelesaian lambat yang paling jelas. Jika struktur yang cepat tidak lebih cepat di atas kertas, ia tidak akan lebih cepat di komputer.

Langkah 8 — Tentukan apa yang terjadi pada masukan yang melanggar janji soal. Tumpukan tiket yang membentuk putaran, daftar prasyarat yang bersiklus, indeks yang melewati ujung *heap*. Pilih sebuah perilaku (pesan, *exception*, atau nilai kembalian khusus) lalu ujilah.

Langkah 9 — Uji, termasuk dengan pemeriksa memori, dan bandingkan dengan versi yang lambat tetapi jelas benar. Jalankan contohnya, lalu setiap kasus tepi. Di C++, kompilasi sekali lagi dengan *sanitizer* dinyalakan. Dan setiap kali sebuah algoritma cepat memiliki kembaran sederhana yang lambat, mengurutkan semuanya alih-alih memelihara *heap*, memindai semuanya alih-alih memakai hash, jalankan keduanya dan bandingkan; beberapa program di bawah melakukan persis hal itu dan mencetak hasil perbandingannya.

Tabel 16.2 memadatkan metode ini dalam satu halaman. Label bercetak tebal di dalam setiap soal di bawah mengikuti urutannya.

Tabel 16.2 Metode sembilan langkah untuk soal tentang sebuah struktur data. Langkah 1 sampai 3 memahami soal, Langkah 4 sampai 6 merancang operasi, Langkah 7 memeriksa bahwa rancangan itu sepadan, dan Langkah 8 serta 9 memeriksa programnya.

Langkah	Tulis di kertas	Pertanyaan yang dijawabnya
1	Soal dengan kata-kata sendiri, dan satu contoh dengan tangan	Tahukah saya keluaran yang semestinya?
2	Strukturnya, dan satu kalimat alasan	*Heap*, tabel hash, graf, pohon, *list*, atau larik?
3	Gambarnya: pohon dan larik, lingkaran dan *list*, slot dan rantai	Seperti apa struktur itu saat ini?
4	Invariannya	Apa yang dijanjikan struktur itu?
5	Kasus tepinya	Kosong, satu elemen, kunci tak ada, tabrakan, siklus?
6	Perubahan-perubahannya, diberi nomor	Adakah yang hilang atau terlambat terlihat dalam urutan ini?
7	Ongkosnya, dan ongkos cara yang lambat	Apakah strukturnya sepadan?
8	Perlakuan terhadap masukan yang melanggar aturan	Pesan, *exception*, atau nilai khusus?
9	Pengujian, pemeriksa memori, dan kembaran yang lambat	Apakah program bekerja, atau hanya *tampak* bekerja?

Bagaimana program-program bab ini diperiksa. Setiap program di bawah lengkap: Anda dapat menyalinnya ke sebuah berkas, mengompilasinya, dan menjalankannya. Program C++ dikompilasi dengan GCC dalam mode C++17 dengan `-Wall -Wextra` tanpa satu peringatan pun; masing-masing dikompilasi sekali lagi dengan AddressSanitizer dan UndefinedBehaviorSanitizer [1], [2] dan dijalankan di bawah Valgrind [3], dan setiap jalannya bersih, tanpa akses tak sah dan tanpa satu bita pun yang bocor. Program Java dikompilasi dengan `javac` dengan semua peringatan *lint* dinyalakan dan tidak menghasilkan satu pun; program Python berjalan di Python 3.10 atau yang lebih baru. Semuanya ditulis untuk JDK 25 dan Python 3.14 tetapi tidak memakai apa pun yang lebih baru daripada JDK 21 dan Python 3.10. Untuk setiap soal, ketiga program mencetak baris-baris yang persis sama, yang diperiksa dengan membandingkan keluarannya bita demi bita; satu-satunya pengecualian adalah pengukuran di Subbab 16.8, yang waktunya memang berbeda menurut bahasa. Perintah-perintahnya sama dengan Bab 8:

```
g++ -std=c++17 -O2 -Wall -Wextra -o p1601 p1601_max_heap.cpp
./p1601
g++ -std=c++17 -g -Wall -Wextra -fsanitize=address,undefined -o p1601_asan
p1601_max_heap.cpp
./p1601_asan                                    keluaran sama, plus laporan bila memori
disalahgunakan
valgrind --leak-check=full ./p1601              Linux: pemeriksaan yang sama tanpa
kompilasi ulang
javac P1601MaxHeap.java && java P1601MaxHeap
python3 p1601_max_heap.py
```

Tiga struktur, tiga bahasa. *Heap*, tabel hash, dan graf tersedia di pustaka standar setiap bahasa dalam buku ini, dengan nama berbeda dan nilai bawaan yang berbeda pula. **Tabel 16.3** adalah kamus kecil; dua barisnya menimbulkan lebih banyak galat daripada gabungan semua baris lain, dan soal-soal di bawah akan kembali ke keduanya.

Tabel 16.3 Struktur-struktur Bab 13 sampai 15 dalam tiga bahasa buku ini. *Priority queue* C++ secara bawaan adalah *max-heap*, sedangkan milik Java dan Python adalah *min-heap*; dan di C++ kata `map` berarti pohon pencarian, bukan tabel hash.

Gagasan	C++	Java	Python
Priority queue	`std::priority_queue<T>`	`PriorityQueue<T>`	fungsi-fungsi `heapq` pada sebuah `list`
Ujung mana yang keluar lebih dahulu	yang **terbesar**	yang **terkecil**	yang **terkecil**
Min-heap di C++	`priority_queue<T, vector<T>, greater<T>>`	(bawaan)	(bawaan)
Max-heap	(bawaan)	`new PriorityQueue<>(Collections.reverseOrder())`	simpan `-x`, atau `heapq.heappush_max` (Python 3.14)
Tabel hash (*map*)	`std::unordered_map<K, V>`	`HashMap<K, V>`	`dict`
Hash set	`std::unordered_set<K>`	`HashSet<K>`	`set`
Map terurut (pohon pencarian seimbang)	`std::map<K, V>`	`TreeMap<K, V>`	tidak ada di pustaka standar
Queue FIFO untuk BFS	`std::queue<int>`	`ArrayDeque<Integer>`	`collections.deque`
Adjacency list	`vector<vector<int>>`	`List<List<Integer>>`	`list` berisi `list`

Baris berbahaya yang pertama adalah arah *priority queue*: `std::priority_queue` menyerahkan elemen terbesar, sedangkan `PriorityQueue` dan `heapq` menyerahkan yang terkecil, sehingga kode yang disalin dari satu bahasa ke bahasa lain diam-diam berubah menjadi kebalikannya (Soal 16.6 memperlihatkan akibatnya). Baris kedua adalah kata *map*: di Java, `HashMap` menyebutkan jati dirinya, tetapi di C++ nama pendek yang wajar, `std::map`, dimiliki oleh pohon pencarian biner yang seimbang, sedangkan tabel hash memakai nama yang lebih panjang, `std::unordered_map` [4], [5], [6], [7]. Soal 16.2 adalah tempat perbedaan itu menjadi penting.

16.3 Soal 16.1 — Melengkapi Kelas Max-Heap

16.3.1 Soalnya

> **SOAL.** A C++ class `MaxHeap` stores a max-heap of integers in a `vector<int> heap` and already provides `insert`, `deleteRoot`, `getMax`, `heapifyArray`, `heapSort`, `printHeap` and two private helpers, `heapifyDown(index)` and `heapifyUp(index)`. Add five member functions and a `clear()` that empties the heap: (1) `int extractMax()`, which removes and returns the maximum element; (2) `void increaseKey(int index, int newValue)`, which increases the

value of the key at a given index; (3) `int getMin()`, which returns the minimum element of the max-heap without removing it; (4) `void buildMaxHeap(const vector<int>& arr)`, which builds a max-heap from an unsorted array; (5) `bool isMaxHeap(const vector<int>& arr)`, which checks whether a given array is a max-heap. The driver clears the heap, inserts 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, calls `extractMax()`, `increaseKey(2, 8)` and `getMin()`, and then, for each of the arrays {10, 7, 8, 5, 2, 3, 1} and {10, 7, 6, 5, 1, 8, 2}, calls `buildMaxHeap` and `isMaxHeap` and prints the results.

Dengan kata lain: kelas C++ `MaxHeap` menyimpan *max-heap* bilangan bulat di dalam `vector<int> heap` dan sudah memiliki `insert`, `deleteRoot`, `getMax`, `heapifyArray`, `heapSort`, `printHeap`, serta dua fungsi bantu privat `heapifyDown` dan `heapifyUp`. Tambahkan lima fungsi anggota dan `clear()`: `extractMax` mengambil dan mengembalikan elemen terbesar; `increaseKey` menaikkan nilai kunci pada indeks tertentu; `getMin` mengembalikan elemen terkecil tanpa mengambilnya; `buildMaxHeap` membangun *max-heap* dari larik tak terurut; `isMaxHeap` memeriksa apakah sebuah larik merupakan *max-heap*. *Driver*-nya menyisipkan sebelas nilai, memanggil ketiga operasi pertama, lalu menguji dua larik dengan dua operasi terakhir.

Kelas yang harus dilengkapi adalah kelas yang dibangun lengkap oleh Bab 14, dan ada gunanya mengingat kembali dua fungsi bantu privatnya secara persis, karena setiap operasi baru tersusun darinya. `heapifyDown(i)` melihat elemen di indeks i beserta kedua anaknya; jika ada anak yang lebih besar, elemen itu ditukar dengan anak yang *lebih besar* dan proses berlanjut dari sana, sehingga elemen yang terlalu kecil tenggelam sampai kedua anaknya tidak lebih besar lagi. `heapifyUp(i)` membandingkan elemen dengan induknya; jika elemen lebih besar, keduanya ditukar dan proses berlanjut dari induk, sehingga elemen yang terlalu besar naik sampai induknya tidak lebih kecil lagi. Bab 14 menyebut keduanya *sift-down* dan *sift-up*; kelas ini mempertahankan nama `heapifyDown` dan `heapifyUp`, dan kita pun demikian.

16.3.2 Dengan kata-kata kita sendiri

Kita memiliki larik `heap` yang elemen ke-0-nya adalah akar sebuah pohon biner *complete*, dan kerabat elemen di indeks i ditemukan dengan aritmetika:

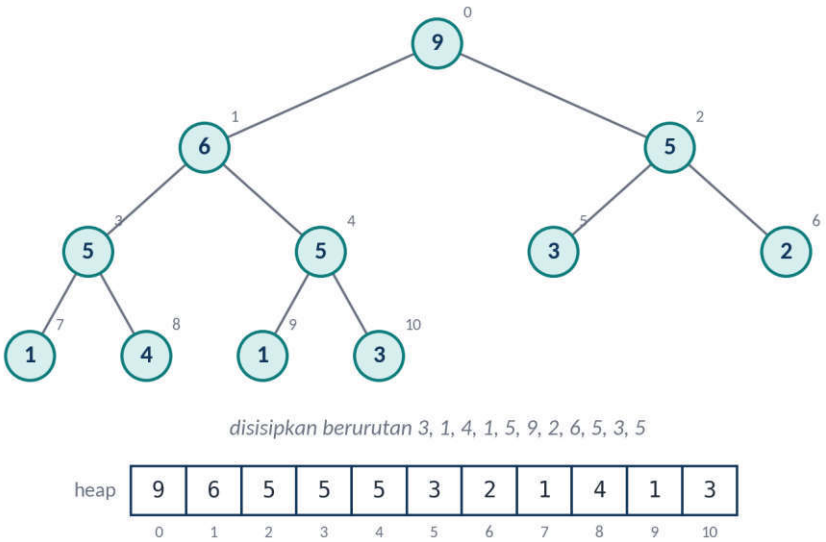
$$\text{parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor, \qquad \text{left}(i) = 2i + 1, \qquad \text{right}(i) = 2i + 2. \tag{16.1}$$

DEFINISI. *Invarian max-heap*: untuk setiap indeks i dengan $1 \le i < n$, berlaku `heap[parent(i)] >= heap[i]`. Dengan kata lain, tidak ada elemen yang lebih besar daripada induknya. Akibatnya, dengan menaiki pohon dari elemen mana pun sampai akar, `heap[0]` adalah elemen terbesar di dalam *heap*.

Setiap fungsi dari kelima fungsi itu memiliki tugas satu kalimat, dan setiap kalimat dapat dibaca dari invarian. `extractMax` harus mengeluarkan akar dan meninggalkan sebuah *heap*. `increaseKey` memperbesar satu elemen, yang hanya dapat mengusik induknya, tidak pernah anak-anaknya, sehingga memerlukan perbaikan *ke atas*. `getMin` harus menemukan elemen terkecil, yang oleh invarian tidak ditempatkan di indeks tetap mana pun, tetapi, seperti akan kita buktikan, selalu ditempatkan di antara daun-daun. `buildMaxHeap` harus mengubah larik sembarang menjadi *heap*, lebih cepat daripada menyisipkan elemennya satu per satu. `isMaxHeap` harus memeriksa invarian untuk setiap i, dan itu cukup dengan satu gelung.

16.3.3 Dengan tangan, selangkah demi selangkah

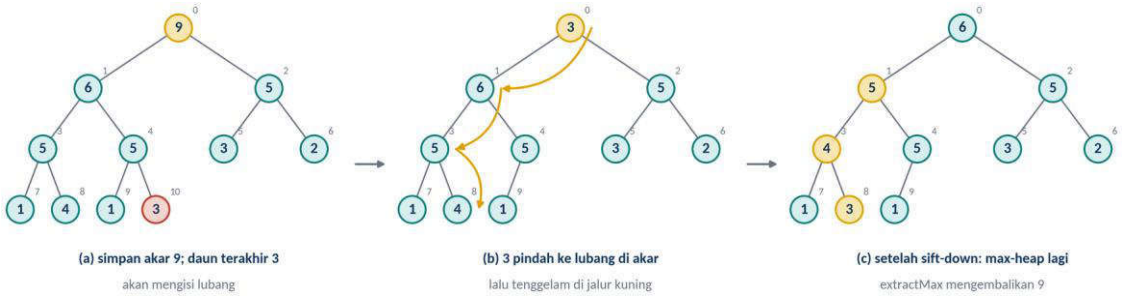
Membangun *heap* dengan penyisipan. *Driver* menyisipkan sebelas nilai. Setiap penyisipan menaruh nilai di ujung larik, yaitu posisi bebas berikutnya pada pohon *complete*, lalu membiarkannya naik. Ikuti tiga penyisipan yang memindahkan sesuatu. Menyisipkan 4 ke `[3, 1]` menaruhnya di indeks 2, yang induknya indeks $\lfloor 1/2 \rfloor = 0$ berisi 3; karena $4 > 3$ keduanya ditukar: `[4, 1, 3]`. Menyisipkan 5 ke `[4, 1, 3, 1]` menaruhnya di indeks 4, induknya indeks 1 berisi 1: tukar; kini di indeks 1, induknya indeks 0 berisi 4: tukar lagi, dan 5 menjadi akar: `[5, 4, 3, 1, 1]`. Menyisipkan 9 menaruhnya di indeks 5, induknya indeks 2 berisi 3: tukar; di indeks 2, induknya indeks 0 berisi 5: tukar; `[9, 4, 5, 1, 1, 3]`. Delapan penyisipan lainnya berjalan serupa, dan setelah kesebelasnya larik menjadi seperti pada **Gambar 16.1**.



Gambar 16.1 *Max-heap* setelah sebelas penyisipan oleh *driver*, digambar sebagai pohon (angka abu-abu kecil adalah indeks larik) dan sebagai larik yang sebenarnya.

Periksa invariannya pada gambar: 9 berada di atas 6 dan 5; 6 di atas 5 dan 5; 5 di indeks 3 di atas 1 dan 4; 5 di indeks 4 di atas 1 dan 3; 5 di indeks 2 di atas 3 dan 2. Setiap induk paling sedikit sebesar anak-anaknya. Perhatikan bahwa larik itu *tidak* terurut, dan memang tidak perlu: 4 di indeks 8 lebih besar daripada 1 di indeks 7 dan lebih besar daripada 3 di indeks 5, dan tidak ada yang keberatan, karena mereka bukan induk dan anak.

1. `extractMax`. Gambar 16.2 memperlihatkan ketiga langkahnya. Jawabannya adalah `heap[0]`, yaitu 9, dan hal pertama yang harus dilakukan adalah *menyimpannya di variabel lokal*, karena langkah berikutnya menimpanya. Pohon harus menyusut satu elemen, dan satu-satunya elemen yang dapat hilang tanpa meninggalkan lubang di tengah pohon *complete* adalah elemen terakhir, indeks 10 berisi 3. Jadi 3 itu kita salin ke indeks 0 dan sel terakhir kita buang. Pohon kembali *complete*, tetapi akarnya, 3, lebih kecil daripada kedua anaknya, 6 dan 5; invarian rusak tepat di satu tempat, yaitu puncak, dan janji yang rusak di puncak adalah tepat yang diperbaiki `heapifyDown(0)`. Anak yang lebih besar adalah 6 di indeks 1: tukar. Kini 3 berada di indeks 1 dengan anak 5 (indeks 3) dan 5 (indeks 4); yang lebih besar adalah 5 yang pertama (kode memilih anak kiri bila keduanya sama): tukar. Kini 3 berada di indeks 3 dengan anak 1 dan 4: tukar dengan 4. Di indeks 8 ia tidak lagi memiliki anak, dan perbaikan berhenti. Fungsi mengembalikan 9 yang disimpan tadi, dan lariknya menjadi `[6, 5, 5, 4, 5, 3, 2, 1, 3, 1]`.



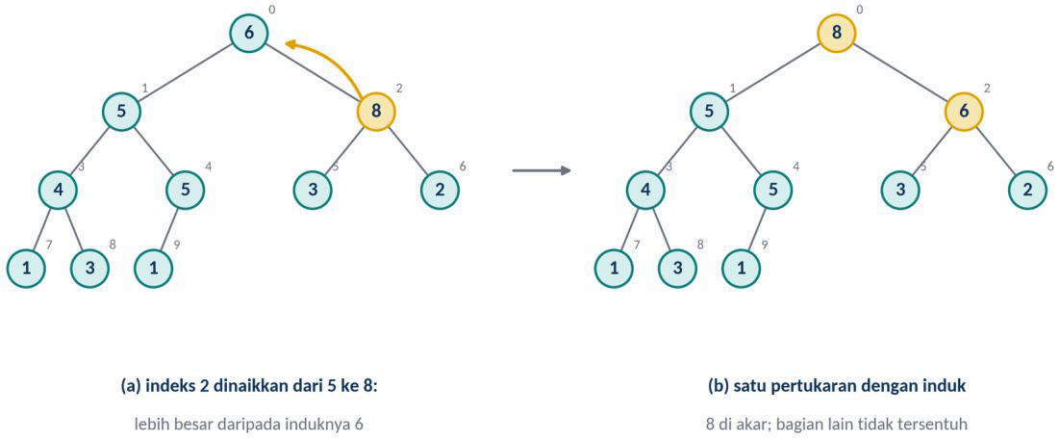
Gambar 16.2 `extractMax` dalam tiga langkah: simpan akar, pindahkan daun terakhir ke akar, lalu biarkan ia tenggelam di sepanjang lintasan anak yang lebih besar sampai invarian *max-heap* berlaku kembali.

Berapa banyak pertukaran yang mungkin terjadi? Setiap pertukaran menurunkan elemen yang tenggelam satu tingkat, dan pohon *complete* berisi n elemen hanya memiliki sekian tingkat. Tingkat d dapat memuat 2^d elemen, sehingga tingkat 0 sampai h bersama-sama memuat paling banyak $2^{h+1} - 1$ elemen, dan tingkat terakhir harus memuat paling sedikit satu. Karena itu $2^h \leq n$, dan

$$h = \lfloor \lg n \rfloor. \tag{16.2}$$

Menurut Persamaan (16.2), *heap* berisi 10 elemen memiliki tinggi 3, dan memang 3 tadi mengalami tepat tiga pertukaran. Karena itu setiap operasi yang menenggelamkan atau menaikkan elemen di sepanjang satu lintasan berongkos $O(\lg n)$.

2. `increaseKey(2, 8)`**.** Elemen di indeks 2 adalah 5, dan menjadi 8. Nilai yang lebih besar tidak dapat merusak hubungan dengan anak-anaknya, yang paling banyak 5 dan tentu paling banyak 8. Ia hanya dapat merusak hubungan dengan induknya: indeks $\lfloor (2-1)/2 \rfloor = 0$ berisi 6, dan $8 > 6$. Jadi perbaikan berjalan *ke atas*, dengan `heapifyUp(2)`: satu pertukaran, dan 8 menjadi akar. **Gambar 16.3** memperlihatkannya. Hasilnya `[8, 5, 6, 4, 5, 3, 2, 1, 3, 1]`.



Gambar 16.3 `increaseKey(2, 8)`: kunci yang dinaikkan hanya dapat menjadi terlalu besar bagi induknya, sehingga ia naik; satu pertukaran memulihkan invarian.

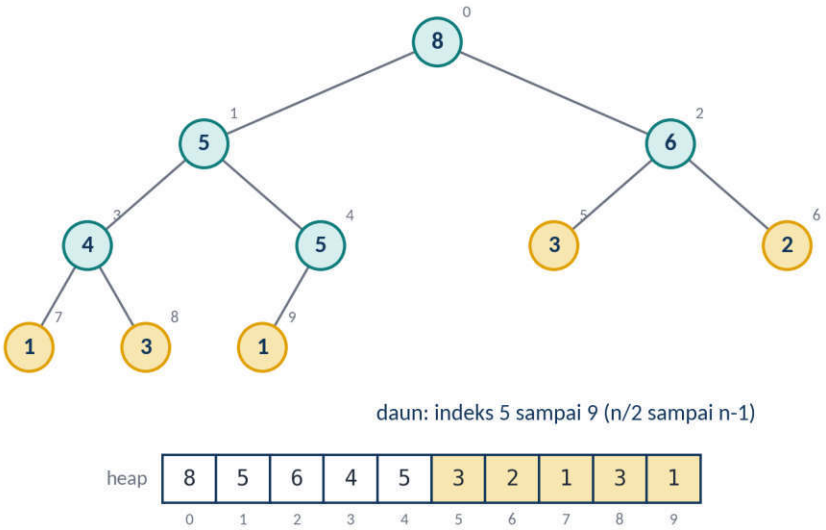
Bagaimana jika pemanggil memberikan nilai yang *lebih kecil*? Elemen itu dapat menjadi lebih kecil daripada anak-anaknya, dan naik adalah perbaikan yang salah. Nama fungsinya menjanjikan kenaikan, sehingga versi kita menolak penurunan dengan *exception* alih-alih diam-diam mengerjakan hal yang salah; Soal Latihan 16.1 menulis fungsi cerminannya, `decreaseKey`, yang perbaikannya berjalan ke bawah.

3. `getMin`**.** Di mana elemen terkecil dapat berada? Bukan di indeks yang memiliki anak, karena anak itu paling banyak sebesar dirinya, sehingga elemen tersebut tidak lebih kecil daripada

semua yang lain. Jadi minimum adalah sebuah *daun*, elemen tanpa anak. Indeks mana saja yang merupakan daun? Indeks i memiliki anak kiri tepat ketika $2i + 1 \le n - 1$, yaitu ketika $i \le (n - 2)/2$. Jadi daun adalah semua indeks mulai $\lfloor n/2 \rfloor$ ke atas:

$$\text{daun} = \{ \lfloor n/2 \rfloor,\ \lfloor n/2 \rfloor + 1,\ \ldots,\ n - 1 \}, \qquad \text{sekitar } n/2 \text{ buah.} \tag{16.3}$$

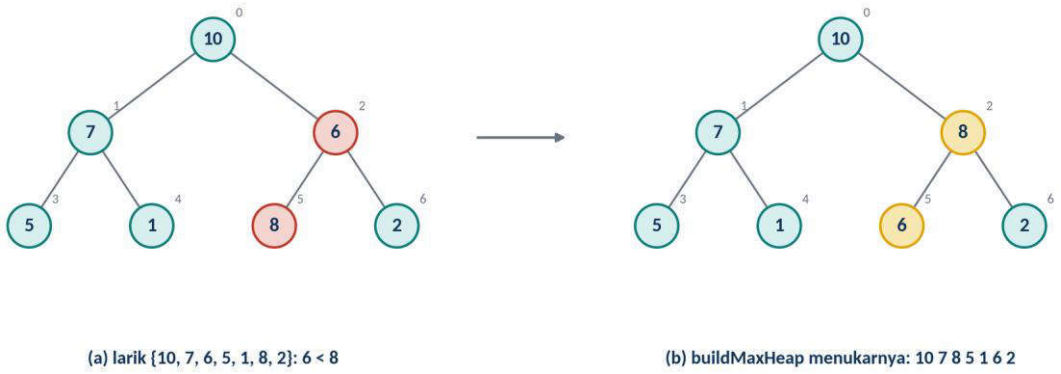
Gambar 16.4 menandai daun-daun itu untuk *heap* kita dengan $n = 10$: indeks 5 sampai 9, berisi 3, 2, 1, 3, 1. Yang terkecil adalah 1. Menurut Persamaan (16.3), `getMin` memeriksa $\lfloor n/2 \rfloor$ elemen alih-alih n: separuh pekerjaan pemindaian biasa, tetapi tetap $\Theta(n)$. *Max-heap* memang tidak menyimpan minimumnya di tempat istimewa mana pun. Jika kedua ekstrem diperlukan dengan cepat, orang memelihara dua *heap*, seperti yang dilakukan Soal 16.7 untuk alasan lain.



Gambar 16.4 Minimum sebuah *max-heap* selalu berupa daun, dan daun-daun itu adalah separuh akhir larik: indeks $\lfloor n/2 \rfloor$ sampai $n - 1$.

4. `buildMaxHeap`. Cara paling sederhana untuk membangun *heap* dari larik adalah menyisipkan elemennya satu per satu, dengan ongkos sampai lg$n$ pertukaran untuk masing-masing, $O(n \lg n)$ seluruhnya. Metode *bottom-up* Floyd [8] lebih cepat. Salin larik apa adanya, lalu panggil `heapifyDown` pada setiap elemen yang memiliki anak, *dari elemen terakhir semacam itu mundur sampai akar*. Mengapa urutannya demikian? `heapifyDown(i)` mengandaikan bahwa kedua subpohon di bawah i sudah merupakan *heap*; bekerja dari bawah ke atas menjaminnya, karena setiap subpohon di bawah i sudah diperbaiki sebelum i. Menurut Persamaan (16.3), elemen yang memiliki anak adalah indeks $\lfloor n/2 \rfloor - 1$ turun sampai 0.

Ambil larik kedua dari *driver*, {10, 7, 6, 5, 1, 8, 2}, $n = 7$, yang digambar pada **Gambar 16.5**. Indeks-indeks internalnya 2, 1, 0. Di indeks 2 elemen 6 memiliki anak 8 dan 2; 8 lebih besar: tukar. Di indeks 1 elemen 7 memiliki anak 5 dan 1: tidak ada yang dikerjakan. Di indeks 0 elemen 10 memiliki anak 7 dan 8: tidak ada yang dikerjakan. Hasilnya `[10, 7, 8, 5, 1, 6, 2]`. Larik pertama, {10, 7, 8, 5, 2, 3, 1}, sudah merupakan *heap*, dan setiap pemanggilan tidak menemukan apa pun untuk dikerjakan.



Gambar 16.5 `buildMaxHeap` pada {10, 7, 6, 5, 1, 8, 2}: hanya indeks 2 yang melanggar invarian, dan satu pertukaran memperbaikinya.

Mengapa ini linear? Sebagian besar elemen berada di dekat dasar, tempat tenggelam itu murah. Paling banyak $n/2^{j+1}$ elemen berada di ketinggian j, dan masing-masing tenggelam paling banyak j tingkat, sehingga jumlah seluruh pertukaran paling banyak

$$\sum_{j=0}^{\lfloor \lg n \rfloor} \frac{n}{2^{j+1}}\, j \;=\; \frac{n}{2}\sum_{j\geq 0} \frac{j}{2^{j}} \;\leq\; \frac{n}{2}\cdot 2 \;=\; n. \tag{16.4}$$

Jumlah $\sum_{j\geq 0} j\,/2^{j}$ sama dengan 2 (Bab 14 menurunkannya), sehingga Persamaan (16.4) membatasi pekerjaan `buildMaxHeap` dengan $O(n)$ [9].

5. `isMaxHeap`. Invarian berlaku untuk setiap i dari 1 sampai $n-1$, sehingga pemeriksaannya berupa satu gelung yang membandingkan setiap elemen dengan induknya dan berhenti pada pelanggar pertama. Untuk {10, 7, 8, 5, 2, 3, 1}: $7 \leq 10$, $8 \leq 10$, $5 \leq 7$, $2 \leq 7$, $3 \leq 8$, $1 \leq 8$: sebuah *heap*. Untuk {10, 7, 6, 5, 1, 8, 2}: di $i = 5$, indeks induk 2 berisi 6 dan $8 > 6$: bukan *heap*. Perhatikan apa yang ditanyakan *driver*. Ia memberikan larik *asli* kepada `isMaxHeap`, bukan *heap* yang dibangun darinya; jadi jawaban kedua adalah "No", padahal `buildMaxHeap` baru saja menghasilkan *heap* yang sempurna dari larik itu. Program kita menanyakan keduanya, agar perbedaannya kelihatan. Larik kosong dan larik berisi satu elemen tidak memiliki i sama sekali dalam rentang $1 \leq i < n$, sehingga jawabannya "Yes" tanpa satu perbandingan pun, sebagaimana mestinya: tidak ada yang dapat dilanggar.

16.3.4 Penyelesaian C++

Kode 16.1 adalah kelas yang telah lengkap. Operasi-operasi yang sudah ada dipertahankan dengan namanya; kita hanya mengubah perilakunya pada *heap* kosong (`getMax` melempar *exception* alih-alih mengembalikan -1 karangan, dengan alasan yang sama seperti *stack* di Bab 8: -1 adalah kunci yang sah) dan memakai `size_t` untuk indeks agar kompilator tidak perlu memperingatkan perbandingan *signed* dengan *unsigned*. `main` menjalankan *driver* yang diberikan, memberi label "The five new operations" pada paruh keduanya, dan menambahkan kasus-kasus tepi di bagian akhir.

Kode 16.1 `p1601_max_heap.cpp`: kelas *max-heap* Soal 16.1 dengan `extractMax`, `increaseKey`, `getMin`, `buildMaxHeap`, `isMaxHeap`, dan `clear`.

```cpp
// p1601_max_heap.cpp - Problem 16.1: a max-heap class with five new operations.
#include <iostream>
#include <stdexcept>
#include <vector>
using namespace std;

class MaxHeap {
private:
    vector<int> heap;   // heap[0] is the root; the children of i are 2i+1 and 2i+2

    // Moves heap[index] down until it is at least as large as both children.
    void heapifyDown(size_t index) {
        size_t largest = index;
        size_t left = 2 * index + 1;
        size_t right = 2 * index + 2;
        if (left < heap.size() && heap[left] > heap[largest]) largest = left;
        if (right < heap.size() && heap[right] > heap[largest]) largest = right;
        if (largest != index) {
            swap(heap[index], heap[largest]);
            heapifyDown(largest);
        }
    }

    // Moves heap[index] up until its parent is at least as large as it.
    void heapifyUp(size_t index) {
        if (index > 0 && heap[(index - 1) / 2] < heap[index]) {
            swap(heap[index], heap[(index - 1) / 2]);
            heapifyUp((index - 1) / 2);
        }
    }

public:
    // ---------------- the operations that were already there ----------------
    void insert(int key) {
        heap.push_back(key);
        heapifyUp(heap.size() - 1);
    }
```

Kode 16.1 (Lanjutan) *p1601_max_heap.cpp: kelas max-heap Soal 16.1 dengan extractMax, increaseKey, getMin, buildMaxHeap, isMaxHeap, dan clear.*

```
    void deleteRoot() {
        if (heap.empty()) {
            cout << "Heap is empty\n";
            return;
        }
        heap[0] = heap.back();
        heap.pop_back();
        if (!heap.empty()) heapifyDown(0);
    }

    int getMax() const {
        if (heap.empty()) throw underflow_error("getMax on an empty heap");
        return heap[0];
    }

    void heapifyArray(const vector<int>& arr) {
        heap = arr;
        for (size_t i = heap.size() / 2; i-- > 0;) heapifyDown(i);
    }

    vector<int> heapSort() {             // empties the heap, largest first
        vector<int> sorted;
        while (!heap.empty()) {
            sorted.push_back(getMax());
            deleteRoot();
        }
        return sorted;
    }

    void printHeap() const {
        if (heap.empty()) cout << "(empty)";
        for (size_t i = 0; i < heap.size(); i++) cout << (i ? " " : "") << heap[i];
        cout << "\n";
    }

    // ---------------- the five new operations, and clear ----------------
    // 1. Removes and returns the largest element.
    int extractMax() {
        if (heap.empty()) throw underflow_error("extractMax on an empty heap");
        int maxVal = heap[0];            // SAVE the answer first
        heap[0] = heap.back();           // the last leaf fills the hole at the root
        heap.pop_back();
        if (!heap.empty()) heapifyDown(0);
        return maxVal;
    }
```

Kode 16.1 (Lanjutan) *p1601_max_heap.cpp: kelas max-heap Soal 16.1 dengan extractMax, increaseKey, getMin, buildMaxHeap, isMaxHeap, dan clear.*

```
    // 2. Raises heap[index] to newValue; a larger key can only move UP.
    void increaseKey(int index, int newValue) {
        if (index < 0 || static_cast<size_t>(index) >= heap.size())
            throw out_of_range("increaseKey: no element at that index");
        if (newValue < heap[index])
            throw invalid_argument("increaseKey: new value is smaller");
        heap[index] = newValue;
        heapifyUp(index);
    }

    // 3. The minimum of a max-heap is a leaf; the leaves are heap[n/2 .. n-1].
    int getMin() const {
        if (heap.empty()) throw underflow_error("getMin on an empty heap");
        size_t n = heap.size();
        int minVal = heap[n / 2];
        for (size_t i = n / 2 + 1; i < n; i++)
            if (heap[i] < minVal) minVal = heap[i];
        return minVal;
    }

    // 4. Floyd's bottom-up construction: sift down every internal node, last first.
    void buildMaxHeap(const vector<int>& arr) {
        heap = arr;
        for (size_t i = heap.size() / 2; i-- > 0;) heapifyDown(i);
    }

    // 5. Every element except the root must be <= its parent.
    static bool isMaxHeap(const vector<int>& arr) {
        for (size_t i = 1; i < arr.size(); i++)
            if (arr[(i - 1) / 2] < arr[i]) return false;
        return true;
    }

    void clear() { heap.clear(); }

    const vector<int>& data() const { return heap; }
};

static const char* yesNo(bool b) { return b ? "Yes" : "No"; }
```

Kode 16.1 (Lanjutan) *p1601_max_heap.cpp: kelas max-heap Soal 16.1 dengan extractMax, increaseKey, getMin, buildMaxHeap, isMaxHeap, dan clear.*

```
int main() {
    MaxHeap h;
    for (int x : {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5}) h.insert(x);
    cout << "Max-Heap array: ";
    h.printHeap();
    cout << "The max element is " << h.getMax() << "\n";
    h.deleteRoot();
    cout << "After deleting the root, heap array: ";
    h.printHeap();
    h.heapifyArray({10, 20, 15, 30, 40});
    cout << "Heapified array: ";
    h.printHeap();
    cout << "Heap sorted array:";
    for (int x : h.heapSort()) cout << " " << x;
    cout << "\n";

    cout << "\n=== The five new operations ===\n\n";
    cout << "Clearing the existing heap ...\n";
    h.clear();
    cout << "Heap after clearing: ";
    h.printHeap();
    for (int x : {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5}) h.insert(x);
    cout << "Max-Heap array: ";
    h.printHeap();

    cout << "\n1. Extracted Max: " << h.extractMax() << "\n";
    cout << "Max-Heap array: ";
    h.printHeap();

    h.increaseKey(2, 8);
    cout << "\n2. Heap after increasing key at index 2 to 8: ";
    h.printHeap();

    cout << "\n3. Minimum value in the heap: " << h.getMin() << "\n\n";

    vector<int> a1 = {10, 7, 8, 5, 2, 3, 1};
    cout << "4a. BuildMaxHeap()\nArray: {10, 7, 8, 5, 2, 3, 1}\n";
    h.clear();
    h.buildMaxHeap(a1);
    cout << "Max-Heap built from unsorted array: ";
    h.printHeap();
    cout << "\n5a. IsMaxHeap()\n";
    cout << "Is the array {10, 7, 8, 5, 2, 3, 1} a Max-Heap? "
         << yesNo(MaxHeap::isMaxHeap(a1)) << "\n";
```

Kode 16.1 (Lanjutan) *p1601_max_heap.cpp: kelas max-heap Soal 16.1 dengan extractMax, increaseKey, getMin, buildMaxHeap, isMaxHeap, dan clear.*

```
    vector<int> a2 = {10, 7, 6, 5, 1, 8, 2};
    cout << "\n4b. BuildMaxHeap()\nArray: {10, 7, 6, 5, 1, 8, 2}\n";
    h.clear();
    h.buildMaxHeap(a2);
    cout << "Max-Heap built from unsorted array: ";
    h.printHeap();
    cout << "\n5b. IsMaxHeap()\n";
    cout << "Is the array {10, 7, 6, 5, 1, 8, 2} a Max-Heap? "
         << yesNo(MaxHeap::isMaxHeap(a2)) << "\n";
    cout << "Is the heap built from it a Max-Heap? "
         << yesNo(MaxHeap::isMaxHeap(h.data())) << "\n";

    cout << "\n=== Edge cases ===\n";
    cout << "isMaxHeap({}) -> " << yesNo(MaxHeap::isMaxHeap({})) << ", "
         << "isMaxHeap({7}) -> " << yesNo(MaxHeap::isMaxHeap({7})) << "\n";
    try {
        h.increaseKey(0, 1);
    } catch (const invalid_argument& e) {
        cout << "caught: " << e.what() << "\n";
    }
    try {
        h.increaseKey(7, 99);
    } catch (const out_of_range& e) {
        cout << "caught: " << e.what() << "\n";
    }
    h.clear();
    try {
        h.extractMax();
    } catch (const underflow_error& e) {
        cout << "caught: " << e.what() << "\n";
    }
    return 0;
}
```

```
Max-Heap array: 9 6 5 5 5 3 2 1 4 1 3
The max element is 9
After deleting the root, heap array: 6 5 5 4 5 3 2 1 3 1
Heapified array: 40 30 15 10 20
Heap sorted array: 40 30 20 15 10

=== The five new operations ===

Clearing the existing heap ...
Heap after clearing: (empty)
Max-Heap array: 9 6 5 5 5 3 2 1 4 1 3

1. Extracted Max: 9
Max-Heap array: 6 5 5 4 5 3 2 1 3 1

2. Heap after increasing key at index 2 to 8: 8 5 6 4 5 3 2 1 3 1

3. Minimum value in the heap: 1

4a. BuildMaxHeap()
Array: {10, 7, 8, 5, 2, 3, 1}
Max-Heap built from unsorted array: 10 7 8 5 2 3 1

5a. IsMaxHeap()
Is the array {10, 7, 8, 5, 2, 3, 1} a Max-Heap? Yes

4b. BuildMaxHeap()
Array: {10, 7, 6, 5, 1, 8, 2}
Max-Heap built from unsorted array: 10 7 8 5 1 6 2

5b. IsMaxHeap()
Is the array {10, 7, 6, 5, 1, 8, 2} a Max-Heap? No
Is the heap built from it a Max-Heap? Yes

=== Edge cases ===
isMaxHeap({}) -> Yes, isMaxHeap({7}) -> Yes
caught: increaseKey: new value is smaller
caught: increaseKey: no element at that index
caught: extractMax on an empty heap
```

Baris demi baris. Lima baris pertama berasal dari operasi yang sudah ada: kesebelas penyisipan menghasilkan larik pada **Gambar 16.1**; menghapus akar menghasilkan larik yang sama dengan hasil `extractMax` nanti; `heapifyArray` mengubah {10, 20, 15, 30, 40} menjadi `40 30 15 10 20`; dan *heap sort*, yang berulang kali mengambil maksimum, mencetak kelima nilai dari yang terbesar. Paruh kedua mengikuti Subbab 16.3.3 dengan persis: 9 diambil, 8 di akar setelah `increaseKey(2, 8)`, 1 sebagai minimum, dua *heap* yang dibangun, dan dua jawaban `isMaxHeap`. Baris tambahan "Is the heap built from it a Max-Heap? Yes" memastikan bahwa `buildMaxHeap` menjalankan tugasnya pada larik yang bukan *heap*. Kasus-kasus tepi memperlihatkan bahwa larik kosong dan larik satu elemen dihitung sebagai *heap*, bahwa penurunan yang diberikan kepada `increaseKey` ditolak, bahwa indeks 7 ditolak pada *heap* berisi tujuh elemen (indeks sah adalah 0 sampai 6), dan bahwa `extractMax` pada *heap* kosong melempar *exception* alih-alih membaca `heap[0]` yang tidak ada.

Tiga rincian layak mendapat satu kalimat masing-masing. Gelung `for (size_t i = heap.size() / 2; i-- > 0;)` menghitung mundur dari $\lfloor n/2 \rfloor - 1$ sampai 0 dengan pencacah *unsigned*; pengujian `i-- > 0` memeriksa sebelum mengurangi, sehingga badan gelung menerima $\lfloor n/2 \rfloor - 1, \ldots, 0$ dan gelung berakhir tanpa pernah menghitung -1, yang tidak dapat ditampung variabel *unsigned*. `isMaxHeap` bersifat `static` karena memeriksa larik yang diberikan kepadanya, bukan *heap* milik objek itu sendiri. Dan `extractMax` memanggil `heapifyDown(0)` hanya jika *heap* tidak

kosong setelah pengambilan, karena *heap* yang tadinya berisi satu elemen kini kosong dan tidak memiliki indeks 0.

Ongkos. Menurut Persamaan (16.2), `extractMax`, `increaseKey`, dan `insert` berongkos $O(\lg n)$; `getMin` dan `isMaxHeap` berongkos $\Theta(n)$ dalam kasus terburuk; `buildMaxHeap` berongkos $O(n)$ menurut Persamaan (16.4); `clear` berongkos $O(n)$ karena `vector::clear` menghancurkan setiap elemen (untuk `int`, kompilator membuatnya praktis gratis).

16.3.5 Penyelesaian Java dan Python

Kode 16.2 dan **Kode 16.3** mencetak baris-baris yang persis sama. Java menyimpan *heap* di dalam `ArrayList<Integer>` dan melempar `NoSuchElementException` pada *heap* kosong, *exception* yang dipakai koleksinya sendiri; Python menyimpan sebuah `list`, melempar `IndexError` seperti `list.pop()` miliknya, dan menulis `getMin` sebagai `min(self.heap[len(self.heap) // 2:])`, yaitu irisan menurut Persamaan (16.3). Dalam program sungguhan Anda akan memakai *heap* pustaka pada **Tabel 16.3**; tujuan di sini adalah menulis operasinya sendiri sekali.

Kode 16.2 `P1601MaxHeap.java`: Soal 16.1 dalam Java.

```java
// P1601MaxHeap.java - Problem 16.1: a max-heap class with five new operations.
import java.util.ArrayList;
import java.util.List;
import java.util.NoSuchElementException;

public class P1601MaxHeap {

    static class MaxHeap {
        // heap.get(0) is the root; the children of i are 2i+1 and 2i+2
        private final ArrayList<Integer> heap = new ArrayList<>();

        private void swap(int i, int j) {
            int t = heap.get(i);
            heap.set(i, heap.get(j));
            heap.set(j, t);
        }

        private void heapifyDown(int index) {
            int largest = index, left = 2 * index + 1, right = 2 * index + 2;
            int n = heap.size();
            if (left < n && heap.get(left) > heap.get(largest)) largest = left;
            if (right < n && heap.get(right) > heap.get(largest)) largest = right;
            if (largest != index) {
                swap(index, largest);
                heapifyDown(largest);
            }
        }

        private void heapifyUp(int index) {
            if (index > 0 && heap.get((index - 1) / 2) < heap.get(index)) {
                swap(index, (index - 1) / 2);
                heapifyUp((index - 1) / 2);
            }
        }

        // ---------------- the operations that were already there ----------------
        void insert(int key) {
            heap.add(key);
            heapifyUp(heap.size() - 1);
        }
```

Kode 16.2 (Lanjutan) *P1601MaxHeap.java: Soal 16.1 dalam Java.*

```
    void deleteRoot() {
        if (heap.isEmpty()) {
            System.out.println("Heap is empty");
            return;
        }
        heap.set(0, heap.get(heap.size() - 1));
        heap.remove(heap.size() - 1);
        if (!heap.isEmpty()) heapifyDown(0);
    }

    int getMax() {
        if (heap.isEmpty())
            throw new NoSuchElementException("getMax on an empty heap");
        return heap.get(0);
    }

    void heapifyArray(List<Integer> arr) {
        buildMaxHeap(arr);
    }

    List<Integer> heapSort() {             // empties the heap, largest first
        List<Integer> sorted = new ArrayList<>();
        while (!heap.isEmpty()) {
            sorted.add(getMax());
            deleteRoot();
        }
        return sorted;
    }

    void printHeap() {
        System.out.println(heap.isEmpty() ? "(empty)" : join(heap));
    }

    // ---------------- the five new operations, and clear ----------------
    int extractMax() {
        if (heap.isEmpty())
            throw new NoSuchElementException("extractMax on an empty heap");
        int maxVal = heap.get(0);                   // SAVE the answer first
        heap.set(0, heap.get(heap.size() - 1));     // last leaf fills the hole
        heap.remove(heap.size() - 1);
        if (!heap.isEmpty()) heapifyDown(0);
        return maxVal;
    }
```

Kode 16.2 (Lanjutan) *P1601MaxHeap.java: Soal 16.1 dalam Java.*

```
        void increaseKey(int index, int newValue) {
            if (index < 0 || index >= heap.size())
                throw new IndexOutOfBoundsException(
                        "increaseKey: no element at that index");
            if (newValue < heap.get(index))
                throw new IllegalArgumentException(
                        "increaseKey: new value is smaller");
            heap.set(index, newValue);
            heapifyUp(index);
        }

        int getMin() {                  // the minimum is a leaf: heap[n/2 .. n-1]
            if (heap.isEmpty())
                throw new NoSuchElementException("getMin on an empty heap");
            int n = heap.size();
            int minVal = heap.get(n / 2);
            for (int i = n / 2 + 1; i < n; i++)
                minVal = Math.min(minVal, heap.get(i));
            return minVal;
        }

        void buildMaxHeap(List<Integer> arr) {
            heap.clear();
            heap.addAll(arr);
            for (int i = heap.size() / 2 - 1; i >= 0; i--) heapifyDown(i);
        }

        static boolean isMaxHeap(List<Integer> arr) {
            for (int i = 1; i < arr.size(); i++)
                if (arr.get((i - 1) / 2) < arr.get(i)) return false;
            return true;
        }

        void clear() {
            heap.clear();
        }

        List<Integer> data() {
            return heap;
        }
    }
```

Kode 16.2 (Lanjutan) *P1601MaxHeap.java: Soal 16.1 dalam Java.*

```
    static String join(List<Integer> xs) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < xs.size(); i++)
            sb.append(i > 0 ? " " : "").append(xs.get(i));
        return sb.toString();
    }

    static String yesNo(boolean b) {
        return b ? "Yes" : "No";
    }

    public static void main(String[] args) {
        MaxHeap h = new MaxHeap();
        for (int x : new int[] {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5}) h.insert(x);
        System.out.print("Max-Heap array: ");
        h.printHeap();
        System.out.println("The max element is " + h.getMax());
        h.deleteRoot();
        System.out.print("After deleting the root, heap array: ");
        h.printHeap();
        h.heapifyArray(List.of(10, 20, 15, 30, 40));
        System.out.print("Heapified array: ");
        h.printHeap();
        System.out.println("Heap sorted array: " + join(h.heapSort()));

        System.out.println("\n=== The five new operations ===\n");
        System.out.println("Clearing the existing heap ...");
        h.clear();
        System.out.print("Heap after clearing: ");
        h.printHeap();
        for (int x : new int[] {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5}) h.insert(x);
        System.out.print("Max-Heap array: ");
        h.printHeap();

        System.out.println("\n1. Extracted Max: " + h.extractMax());
        System.out.print("Max-Heap array: ");
        h.printHeap();

        h.increaseKey(2, 8);
        System.out.print("\n2. Heap after increasing key at index 2 to 8: ");
        h.printHeap();

        System.out.println("\n3. Minimum value in the heap: " + h.getMin() + "\n");
```

Kode 16.2 (Lanjutan) *P1601MaxHeap.java: Soal 16.1 dalam Java.*

```
        List<Integer> a1 = List.of(10, 7, 8, 5, 2, 3, 1);
        System.out.println("4a. BuildMaxHeap()\nArray: {10, 7, 8, 5, 2, 3, 1}");
        h.clear();
        h.buildMaxHeap(a1);
        System.out.print("Max-Heap built from unsorted array: ");
        h.printHeap();
        System.out.println("\n5a. IsMaxHeap()");
        System.out.println("Is the array {10, 7, 8, 5, 2, 3, 1} a Max-Heap? "
                + yesNo(MaxHeap.isMaxHeap(a1)));

        List<Integer> a2 = List.of(10, 7, 6, 5, 1, 8, 2);
        System.out.println("\n4b. BuildMaxHeap()\nArray: {10, 7, 6, 5, 1, 8, 2}");
        h.clear();
        h.buildMaxHeap(a2);
        System.out.print("Max-Heap built from unsorted array: ");
        h.printHeap();
        System.out.println("\n5b. IsMaxHeap()");
        System.out.println("Is the array {10, 7, 6, 5, 1, 8, 2} a Max-Heap? "
                + yesNo(MaxHeap.isMaxHeap(a2)));
        System.out.println("Is the heap built from it a Max-Heap? "
                + yesNo(MaxHeap.isMaxHeap(h.data())));

        System.out.println("\n=== Edge cases ===");
        System.out.println("isMaxHeap({}) -> " + yesNo(MaxHeap.isMaxHeap(List.of()))
                + ", isMaxHeap({7}) -> " + yesNo(MaxHeap.isMaxHeap(List.of(7))));
        try {
            h.increaseKey(0, 1);
        } catch (IllegalArgumentException e) {
            System.out.println("caught: " + e.getMessage());
        }
        try {
            h.increaseKey(7, 99);
        } catch (IndexOutOfBoundsException e) {
            System.out.println("caught: " + e.getMessage());
        }
        h.clear();
        try {
            h.extractMax();
        } catch (NoSuchElementException e) {
            System.out.println("caught: " + e.getMessage());
        }
    }
}
```

Kode 16.3 `p1601_max_heap.py`: Soal 16.1 dalam Python.

```python
# p1601_max_heap.py - Problem 16.1: a max-heap class with five new operations.


class MaxHeap:
    def __init__(self):
        self.heap = []              # heap[0] is the root; children of i: 2i+1, 2i+2

    def _heapify_down(self, index):
        largest, left, right = index, 2 * index + 1, 2 * index + 2
        h = self.heap
        if left < len(h) and h[left] > h[largest]:
            largest = left
        if right < len(h) and h[right] > h[largest]:
            largest = right
        if largest != index:
            h[index], h[largest] = h[largest], h[index]
            self._heapify_down(largest)

    def _heapify_up(self, index):
        h = self.heap
        if index > 0 and h[(index - 1) // 2] < h[index]:
            p = (index - 1) // 2
            h[index], h[p] = h[p], h[index]
            self._heapify_up(p)

    # ---------------- the operations that were already there ----------------
    def insert(self, key):
        self.heap.append(key)
        self._heapify_up(len(self.heap) - 1)

    def delete_root(self):
        if not self.heap:
            print("Heap is empty")
            return
        self.heap[0] = self.heap[-1]
        self.heap.pop()
        if self.heap:
            self._heapify_down(0)

    def get_max(self):
        if not self.heap:
            raise IndexError("getMax on an empty heap")
        return self.heap[0]
```

Kode 16.3 (Lanjutan) *p1601_max_heap.py: Soal 16.1 dalam Python.*

```
    def heapify_array(self, arr):
        self.build_max_heap(arr)

    def heap_sort(self):                    # empties the heap, largest first
        out = []
        while self.heap:
            out.append(self.get_max())
            self.delete_root()
        return out

    def print_heap(self):
        print(" ".join(map(str, self.heap)) if self.heap else "(empty)")

    # ---------------- the five new operations, and clear ----------------
    def extract_max(self):
        if not self.heap:
            raise IndexError("extractMax on an empty heap")
        max_val = self.heap[0]              # SAVE the answer first
        self.heap[0] = self.heap[-1]        # the last leaf fills the hole
        self.heap.pop()
        if self.heap:
            self._heapify_down(0)
        return max_val

    def increase_key(self, index, new_value):
        if not 0 <= index < len(self.heap):
            raise IndexError("increaseKey: no element at that index")
        if new_value < self.heap[index]:
            raise ValueError("increaseKey: new value is smaller")
        self.heap[index] = new_value
        self._heapify_up(index)

    def get_min(self):              # the minimum is a leaf: heap[n//2 .. n-1]
        if not self.heap:
            raise IndexError("getMin on an empty heap")
        return min(self.heap[len(self.heap) // 2:])

    def build_max_heap(self, arr):
        self.heap = list(arr)
        for i in range(len(self.heap) // 2 - 1, -1, -1):
            self._heapify_down(i)

    @staticmethod
    def is_max_heap(arr):
        return all(arr[(i - 1) // 2] >= arr[i] for i in range(1, len(arr)))
```

Kode 16.3 (Lanjutan) *p1601_max_heap.py: Soal 16.1 dalam Python.*

```
    def clear(self):
        self.heap.clear()


def yes_no(b):
    return "Yes" if b else "No"


def main():
    h = MaxHeap()
    for x in (3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5):
        h.insert(x)
    print("Max-Heap array: ", end="")
    h.print_heap()
    print("The max element is", h.get_max())
    h.delete_root()
    print("After deleting the root, heap array: ", end="")
    h.print_heap()
    h.heapify_array([10, 20, 15, 30, 40])
    print("Heapified array: ", end="")
    h.print_heap()
    print("Heap sorted array:", " ".join(map(str, h.heap_sort())))

    print("\n=== The five new operations ===\n")
    print("Clearing the existing heap ...")
    h.clear()
    print("Heap after clearing: ", end="")
    h.print_heap()
    for x in (3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5):
        h.insert(x)
    print("Max-Heap array: ", end="")
    h.print_heap()

    print("\n1. Extracted Max:", h.extract_max())
    print("Max-Heap array: ", end="")
    h.print_heap()

    h.increase_key(2, 8)
    print("\n2. Heap after increasing key at index 2 to 8: ", end="")
    h.print_heap()

    print("\n3. Minimum value in the heap:", h.get_min(), end="\n\n")
```

Kode 16.3 (Lanjutan) *p1601_max_heap.py: Soal 16.1 dalam Python.*

```
    a1 = [10, 7, 8, 5, 2, 3, 1]
    print("4a. BuildMaxHeap()\nArray: {10, 7, 8, 5, 2, 3, 1}")
    h.clear()
    h.build_max_heap(a1)
    print("Max-Heap built from unsorted array: ", end="")
    h.print_heap()
    print("\n5a. IsMaxHeap()")
    print("Is the array {10, 7, 8, 5, 2, 3, 1} a Max-Heap?",
          yes_no(MaxHeap.is_max_heap(a1)))

    a2 = [10, 7, 6, 5, 1, 8, 2]
    print("\n4b. BuildMaxHeap()\nArray: {10, 7, 6, 5, 1, 8, 2}")
    h.clear()
    h.build_max_heap(a2)
    print("Max-Heap built from unsorted array: ", end="")
    h.print_heap()
    print("\n5b. IsMaxHeap()")
    print("Is the array {10, 7, 6, 5, 1, 8, 2} a Max-Heap?",
          yes_no(MaxHeap.is_max_heap(a2)))
    print("Is the heap built from it a Max-Heap?",
          yes_no(MaxHeap.is_max_heap(h.heap)))

    print("\n=== Edge cases ===")
    print(f"isMaxHeap({{}}) -> {yes_no(MaxHeap.is_max_heap([]))}, "
          f"isMaxHeap({{7}}) -> {yes_no(MaxHeap.is_max_heap([7]))}")
    try:
        h.increase_key(0, 1)
    except ValueError as e:
        print("caught:", e)
    try:
        h.increase_key(7, 99)
    except IndexError as e:
        print("caught:", e)
    h.clear()
    try:
        h.extract_max()
    except IndexError as e:
        print("caught:", e)


if __name__ == "__main__":
    main()
```

16.3.6 Penyelesaian keliru yang menggoda

Dua jalan pintas yang menggoda layak dijalankan, karena salah satunya *lolos pada driver yang diberikan*. Yang pertama adalah anggapan bahwa elemen terkecil sebuah *max-heap* berada di ujung larik; yang kedua adalah menulis batas gelung `isMaxHeap` dengan ungkapan yang akrab, `arr.size() - 2`:

```
int getMinWrong(const vector<int>& heap) { return heap.back(); }       // WRONG
for (size_t i = 0; i <= (arr.size() - 2) / 2; i++)                     // WRONG for n < 2
```

Program `wrong/p1601_wrong.cpp` memanggil keduanya, mula-mula pada *heap* dari *driver*, lalu pada *heap* tiga elemen, lalu pada larik satu elemen, dengan kompilasi yang menyalakan *sanitizer*:

```
getMinWrong(sample) = 1
getMinWrong({5, 3, 4}) = 4
```

```
==ERROR: AddressSanitizer: heap-buffer-overflow ...
READ of size 4 ... in isMaxHeapWrong ... p1601_wrong.cpp:12
0x... is located 0 bytes after 4-byte region
SUMMARY: AddressSanitizer: heap-buffer-overflow ... p1601_wrong.cpp:12 in isMaxHeapWrong
```

Pada *heap* dari *driver*, `getMinWrong` menjawab 1, yang benar, karena kebetulan: elemen terakhirnya kebetulan salah satu dari kedua angka 1. Sisipkan 5, 3, 4 sebagai gantinya, dan *heap*-nya menjadi `[5, 3, 4]`; elemen terakhirnya 4, padahal minimumnya 3. Elemen terakhir selalu berupa daun, tetapi, menurut Persamaan (16.3), ia hanya satu dari sekitar $n/2$ daun. Kesalahan kedua lebih parah. `arr.size()` bertipe `size_t` yang *unsigned*, sehingga untuk larik satu elemen `arr.size() - 2` bukanlah -1 melainkan `size_t` terbesar, sekitar 1.8×10^{19}; gelung berjalan, membaca `arr[1]` yang tidak ada, dan AddressSanitizer menghentikan program dengan laporan di atas (laporan itu telah dipendekkan; laporan lengkapnya menyebutkan setiap bingkai *stack*). Tanpa *sanitizer*, program membaca bita apa pun yang kebetulan berada setelah larik dan dapat mencetak jawaban mana saja. `isMaxHeap` kita menghindari kedua masalah itu dengan bergelung atas anak-anak, `for (size_t i = 1; i < arr.size(); i++)`, yang batasnya tidak mungkin *underflow*.

16.3.7 Uji diri

> **UJI DIRI.** Mulailah dari *heap* `[8, 5, 6, 4, 5, 3, 2, 1, 3, 1]` (*heap* setelah langkah 2). Panggil `extractMax()` dua kali. Apa yang dikembalikan setiap pemanggilan, dan seperti apa larik setelah masing-masing? Kemudian panggil `getMin()`: indeks mana saja yang diperiksanya?
>
> *Jawaban.* Pemanggilan pertama mengembalikan 8; elemen terakhir 1 pindah ke akar dan tenggelam melewati 6 (anak yang lebih besar), lalu melewati 3 (di indeks 5, lebih besar daripada 2): `[6, 5, 3, 4, 5, 1, 2, 1, 3]`. Pemanggilan kedua mengembalikan 6; elemen terakhir 3 pindah ke akar, tenggelam melewati 5 (indeks 1, lebih besar daripada 3), lalu melewati 5 (indeks 4, lebih besar daripada 4): `[5, 5, 3, 4, 3, 1, 2, 1]`. Kini $n = 8$, sehingga menurut Persamaan (16.3) `getMin` memeriksa indeks 4 sampai 7, berisi 3, 1, 2, 1, dan mengembalikan 1.

16.4 Soal 16.2 — Menyusun Tiket Menurut Urutan Perjalanan

16.4.1 Soalnya

> **SOAL.** Given a list of tickets, find the itinerary in travel order, using hashing. (1) Create a hash map `dataset` of the tickets: every entry has the form *from* → *to*. (2) Find the starting point of the itinerary: (a) create a reverse hash map `reverseMap` whose entries have the form *to* → *from*; (b) traverse `dataset`; for every key, check whether it exists in `reverseMap`; the key that cannot be found is the starting point. (3) From the starting point, traverse `dataset` and print the itinerary. Input: Cianjur → Bandung, Badung → Denpasar, Gianyar → Cianjur, Denpasar → Gianyar. Output: `Badung -> Denpasar, Denpasar -> Gianyar, Gianyar -> Cianjur, Cianjur -> Bandung`.
>
> *Dengan kata lain:* dari setumpuk tiket, tentukan rencana perjalanan menurut urutannya dengan memakai hashing. Buat *hash map* `dataset` berisi pasangan asal → tujuan; buat *hash map* kebalikannya, `reverseMap`, berisi tujuan → asal; kota asal yang tidak ditemukan di `reverseMap` adalah titik awal; dari titik awal itu, ikuti `dataset` dan cetak perjalanannya.

(Badung dan Bandung adalah dua tempat yang berbeda: Badung adalah kabupaten di Bali, bertetangga dengan Denpasar dan Gianyar, sedangkan Bandung adalah ibu kota Jawa Barat, bertetangga dengan Cianjur. Selisih satu huruf itu termasuk yang menjadikan contohnya bahan uji yang baik.)

16.4.2 Dengan kata-kata kita sendiri

Seorang pelancong memegang setumpuk tiket sekali jalan yang sudah teracak. Setiap tiket meninggalkan satu kota dan tiba di kota lain. Tiket-tiket itu membentuk satu perjalanan: setiap kota ditinggalkan paling banyak sekali, dan perjalanan tidak pernah kembali ke kota yang sudah dikunjungi. Kita harus mencetak tiket-tiket itu menurut urutan pemakaiannya.

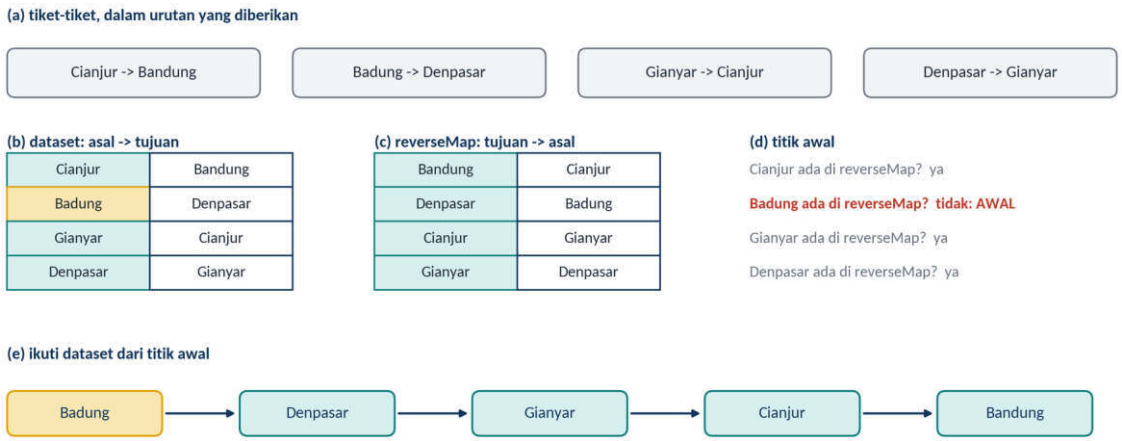
Apa yang dimiliki tiket pertama dan tidak dimiliki tiket lain? Kota asalnya *tidak pernah menjadi kota tujuan*. Setiap tiket lain meninggalkan kota yang menjadi tujuan tiket sebelumnya. Jadi titik awal adalah satu-satunya kota asal yang tidak muncul di sisi tujuan, dan sebuah *hash set* atau *hash map* berisi kota-kota tujuan menjawab "pernahkah kota ini dituju?" dalam waktu konstan. Setelah titik awal diketahui, tiket berikutnya adalah tiket yang meninggalkan kota tempat kita berada, dan *hash map* dari kota asal ke kota tujuan menjawab "ke mana tiket yang meninggalkan kota ini pergi?" dalam waktu konstan. Berjalan dari titik awal, satu pencarian per tiket, mencetak perjalanannya.

> **DEFINISI.** Setumpuk n tiket *membentuk satu rantai* bila kesemua n kota asalnya berbeda dan tepat satu di antaranya, yaitu *titik awal*, bukan kota tujuan, dan bila mengikuti tiket-tiket dari titik awal memakai kesemua n tiket. Rantai itu adalah sebuah lintasan di graf berarah yang simpul-simpulnya kota dan busur-busurnya tiket (Bab 13).

Definisi itu memberi tahu apa saja yang dapat salah, dan Langkah 8 metode kita meminta kita menentukan tindakannya: dua tiket yang meninggalkan kota yang sama (maka `dataset` bahkan tidak dapat dibangun, karena sebuah *map* hanya menyimpan satu nilai per kunci), tidak ada titik awal sama sekali (tiket-tiket membentuk putaran), beberapa titik awal (dua perjalanan terpisah), atau satu titik awal tetapi penelusurannya berhenti sebelum semua tiket terpakai (sebuah rantai ditambah putaran terpisah). Program kita melaporkan setiap keadaan itu alih-alih mencetak perjalanan yang tidak lengkap.

16.4.3 Dengan tangan, selangkah demi selangkah

Gambar 16.6 mengikuti ketiga langkah soal pada contohnya.



Gambar 16.6 Soal 16.2 dengan tangan: tiket-tiket sebagaimana diberikan (a), *map* dari asal ke tujuan (b) dan kebalikannya (c), pencarian satu-satunya kota asal yang tidak pernah dituju (d), dan penelusuran dari kota itu (e).

1. **Bangun `dataset`.** Cianjur → Bandung, Badung → Denpasar, Gianyar → Cianjur, Denpasar → Gianyar. Empat penyisipan, masing-masing rata-rata $O(1)$.
2. **Bangun `reverseMap`.** Bandung → Cianjur, Denpasar → Badung, Cianjur → Gianyar, Gianyar → Denpasar.

3. **Cari titik awal.** Apakah Cianjur pernah dituju? Ya (dari Gianyar). Badung? *Tidak*. Badung adalah titik awal. (Kita tetap memeriksa dua sisanya, Gianyar dan Denpasar, yang keduanya kota tujuan, agar dapat melaporkan lebih dari satu titik awal bila masukannya rusak.)
4. **Telusuri.** Dari Badung, `dataset` memberi Denpasar; dari Denpasar, Gianyar; dari Gianyar, Cianjur; dari Cianjur, Bandung; Bandung bukan kunci `dataset`, sehingga perjalanan berakhir. Empat dari empat tiket terpakai: keluarannya adalah baris yang diharapkan.

Setiap langkah terdiri atas sejumlah konstan operasi *hash map* per tiket, sehingga seluruh algoritma berongkos rata-rata $O(n)$ untuk n tiket. Alternatif yang paling jelas, "untuk setiap tiket, cari di tumpukan tiket yang meninggalkan kota tujuannya", berongkos $O(n)$ per pencarian dan $O(n^2)$ seluruhnya: untuk sepuluh ribu tiket, seratus juta perbandingan dibandingkan sekitar empat puluh ribu operasi hash.

16.4.4 *Map* mana yang merupakan *hash map*?

Soal meminta *hash map*, dan di C++ kita tergoda meraih wadah yang namanya cukup `std::map`. Wadah itu bukan tabel hash. Standar C++ mewajibkan `std::map` menjaga kunci-kuncinya terurut dan melakukan pencarian, penyisipan, dan penghapusan dalam waktu logaritmik, dan setiap pustaka standar memenuhi syarat itu dengan pohon pencarian biner yang seimbang, yaitu *red-black tree*: sepupu seimbang dari pohon pencarian biner di Bab 11 [9], [5]. Tabel hash-nya adalah `std::unordered_map`. Program yang menyelesaikan soal ini dengan `std::map` memang mencetak perjalanan yang benar, karena kedua wadah dapat menjawab "kunci ini menuju ke mana?", tetapi program itu tidak memakai hashing, membayar $O(\lg n)$ perbandingan per pencarian alih-alih rata-rata $O(1)$, dan mengunjungi kuncinya menurut abjad, hal yang tidak pernah dijanjikan tabel hash.

Kode 16.4 memperlihatkan perbedaannya. Program ini memasukkan keempat kota asal ke kedua wadah, mencetak urutan kunjungan masing-masing, menanyakan kepada tabel hash di *bucket* mana setiap kunci mendarat, dan menghitung perbandingan yang dilakukan `std::map` untuk menemukan satu kunci.

Kode 16.4 `p1602_map_vs_hash.cpp`: empat kunci yang sama di `std::map`, sebuah pohon pencarian, dan di `std::unordered_map`, sebuah tabel hash.

```cpp
// p1602_map_vs_hash.cpp - std::map is a search tree; unordered_map a hash table.
#include <iostream>
#include <map>
#include <string>
#include <unordered_map>
using namespace std;

static int comparisons = 0;
struct CountingLess {                       // an ordinary "<" that counts its calls
    bool operator()(const string& a, const string& b) const {
        comparisons++;
        return a < b;
    }
};

int main() {
    const char* from[] = {"Cianjur", "Badung", "Gianyar", "Denpasar"};
    const char* to[] = {"Bandung", "Denpasar", "Cianjur", "Gianyar"};

    map<string, string, CountingLess> tree;     // a red-black tree, ordered by key
    unordered_map<string, string> table;        // a hash table with chained buckets
    for (int i = 0; i < 4; i++) {
        tree[from[i]] = to[i];
        table[from[i]] = to[i];
    }

    cout << "std::map visits the keys in sorted order:\n ";
    for (const auto& kv : tree) cout << " " << kv.first;
    cout << "\nstd::unordered_map visits them bucket by bucket:\n ";
    for (const auto& kv : table) cout << " " << kv.first;
    cout << "\n\nThe hash table has " << table.bucket_count() << " buckets:\n";
    for (int i = 0; i < 4; i++)
        cout << "  " << from[i] << " is in bucket " << table.bucket(from[i])
             << "\n";

    comparisons = 0;
    tree.find("Gianyar");
    cout << "\nmap.find(\"Gianyar\") used " << comparisons << " key comparisons\n";
    cout << "unordered_map.find(\"Gianyar\") hashes the key once and looks in "
         << "bucket " << table.bucket("Gianyar") << "\n";
    return 0;
}
```

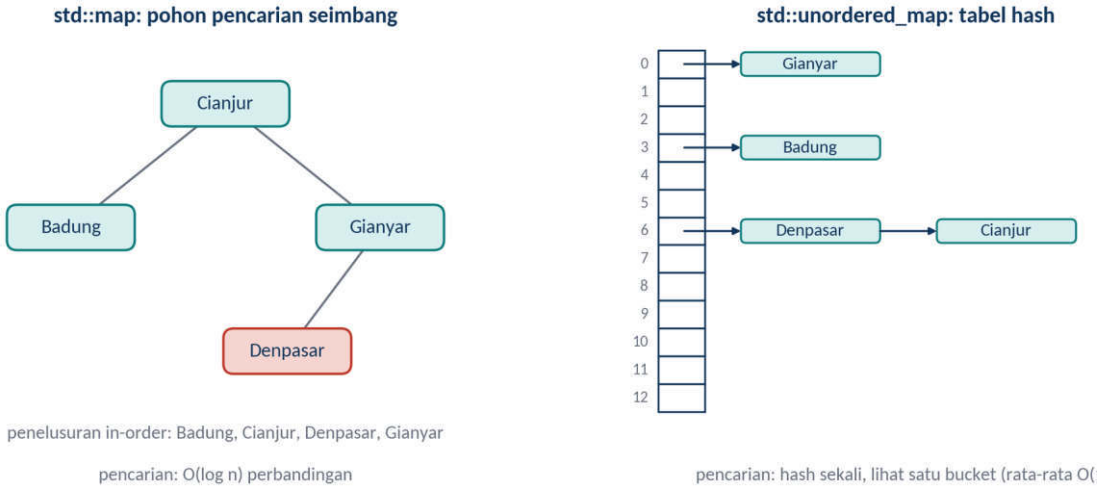
```
std::map visits the keys in sorted order:
  Badung Cianjur Denpasar Gianyar
std::unordered_map visits them bucket by bucket:
  Gianyar Badung Denpasar Cianjur

The hash table has 13 buckets:
  Cianjur is in bucket 6
  Badung is in bucket 3
  Gianyar is in bucket 0
  Denpasar is in bucket 6

map.find("Gianyar") used 4 key comparisons
unordered_map.find("Gianyar") hashes the key once and looks in bucket 0
```

Gambar 16.7 menggambar kedua wadah sebagaimana dilukiskan keluarannya. Pohon mengunjungi Badung, Cianjur, Denpasar, Gianyar: menurut abjad, karena penelusuran *in-order* sebuah pohon pencarian selalu demikian (Bab 11). Tabel hash mengunjunginya menurut urutan *bucket*-nya, yang bergantung pada fungsi hash pustaka dan banyaknya *bucket*, di sini 13; Cianjur dan Denpasar *bertabrakan* di *bucket* 6 dan berbagi satu rantai. Kompilator lain, atau kompilator yang sama setelah beberapa penyisipan lagi memicu *rehash*, dapat mencetak urutan yang berbeda. Menemukan “Gianyar” di pohon memerlukan 4 perbandingan string utuh;

menemukannya di tabel memerlukan satu perhitungan hash dan satu tengokan ke dalam satu *bucket*.



Gambar 16.7 Kunci-kunci yang sama di `std::map`, *red-black tree* yang dijaga terurut (simpul merah adalah simpul merah dalam skema *red-black*), dan di `std::unordered_map`, tabel hash dengan 13 *bucket* tempat Cianjur dan Denpasar bertabrakan.

Tidak ada wadah yang "lebih baik"; keduanya menjawab pertanyaan yang berbeda. Bila tugas memerlukan kunci-kunci secara berurutan, kunci terkecil, atau semua kunci di antara dua batas, pohonlah pilihan yang tepat dan tabel hash pilihan yang salah. Bila tugas hanya memerlukan "adakah kunci ini, dan apa pasangannya?", tabel hash-lah pilihan yang tepat. Subbab 16.8 mengukur tawar-menawar ini dengan tepat pada sepuluh ribu buku. Di Java namanya berterus terang, `HashMap` lawan `TreeMap`; di Python, `dict` adalah tabel hash dan pustaka standarnya sama sekali tidak memiliki *tree map* [6], [7].

16.4.5 Penyelesaian C++

Kode 16.5 mengikuti ketiga langkah dengan dua `unordered_map`. Program ini memeriksa masukan sebagaimana diminta definisi di Subbab 16.4.2 dan menjalankan enam tumpukan tiket: yang diberikan, satu putaran mobil buku Pustaka Ceria melintasi Jawa Timur, satu tiket saja, dan tiga tumpukan yang rusak.

Kode 16.5 `p1602_itinerary.cpp`: Soal 16.2 dengan dua *hash map*, dan laporan untuk setiap cara sebuah tumpukan tiket gagal membentuk satu rantai.

```cpp
// p1602_itinerary.cpp - Problem 16.2: put tickets in travel order, with hashing.
#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>
using namespace std;

struct Ticket {
    string from, to;
};

// Fills route with the tickets in travel order. Returns "" on success,
// otherwise a sentence that says why the tickets cannot be put in order.
string findItinerary(const vector<Ticket>& tickets, vector<Ticket>& route) {
    unordered_map<string, string> dataset;      // from -> to  (a hash table)
    unordered_map<string, string> reverseMap;   // to -> from  (a hash table)
    for (const Ticket& t : tickets) {
        if (!dataset.emplace(t.from, t.to).second)
            return "two tickets leave " + t.from;
        reverseMap[t.to] = t.from;
    }

    // The start is the only city that is left but never arrived at.
    vector<string> starts;
    for (const Ticket& t : tickets)
        if (reverseMap.find(t.from) == reverseMap.end()) starts.push_back(t.from);
    if (starts.empty()) return "no starting point (the tickets form a loop)";
    if (starts.size() > 1) {
        string s = to_string(starts.size()) + " starting points:";
        for (const string& c : starts) s += " " + c;
        return s;
    }

    // Follow the tickets: from the start, look up where each city leads.
    string city = starts[0];
    auto it = dataset.find(city);
    while (it != dataset.end() && route.size() < tickets.size()) {
        route.push_back({it->first, it->second});
        it = dataset.find(it->second);
    }
    if (route.size() != tickets.size())
        return "the tickets do not form one chain";
    return "";
}
```

Kode 16.5 (Lanjutan) *p1602_itinerary.cpp: Soal 16.2 dengan dua hash map, dan laporan untuk setiap cara sebuah tumpukan tiket gagal membentuk satu rantai.*

```
void solve(const string& name, const vector<Ticket>& tickets) {
    cout << "== " << name << "\nInput:\n";
    for (size_t i = 0; i < tickets.size(); i++)
        cout << (i ? "; " : "") << tickets[i].from << " -> " << tickets[i].to;
    vector<Ticket> route;
    string error = findItinerary(tickets, route);
    if (!error.empty()) {
        cout << "\nInvalid input: " << error << "\n";
        return;
    }
    cout << "\nStart: " << route[0].from << "\nOutput:\n";
    for (size_t i = 0; i < route.size(); i++)
        cout << (i ? ", " : "") << route[i].from << " -> " << route[i].to;
    cout << "\n";
}

int main() {
    solve("The given tickets", {{"Cianjur", "Bandung"}, {"Badung", "Denpasar"},
                                {"Gianyar", "Cianjur"}, {"Denpasar", "Gianyar"}});
    solve("The book van of Pustaka Ceria",
          {{"Malang", "Batu"}, {"Surabaya", "Sidoarjo"},
           {"Pasuruan", "Malang"}, {"Sidoarjo", "Pasuruan"}});
    solve("A single ticket", {{"Kediri", "Blitar"}});
    solve("A loop", {{"Bandung", "Jakarta"}, {"Jakarta", "Bandung"}});
    solve("Two separate trips", {{"Solo", "Yogyakarta"}, {"Semarang", "Kudus"}});
    solve("A chain plus a loop",
          {{"Bogor", "Depok"}, {"Bekasi", "Tangerang"}, {"Tangerang", "Bekasi"}});
    return 0;
}
```

```
== The given tickets
Input:
Cianjur -> Bandung; Badung -> Denpasar; Gianyar -> Cianjur; Denpasar -> Gianyar
Start: Badung
Output:
Badung -> Denpasar, Denpasar -> Gianyar, Gianyar -> Cianjur, Cianjur -> Bandung
== The book van of Pustaka Ceria
Input:
Malang -> Batu; Surabaya -> Sidoarjo; Pasuruan -> Malang; Sidoarjo -> Pasuruan
Start: Surabaya
Output:
Surabaya -> Sidoarjo, Sidoarjo -> Pasuruan, Pasuruan -> Malang, Malang -> Batu
== A single ticket
Input:
Kediri -> Blitar
Start: Kediri
Output:
Kediri -> Blitar
== A loop
Input:
Bandung -> Jakarta; Jakarta -> Bandung
Invalid input: no starting point (the tickets form a loop)
== Two separate trips
Input:
Solo -> Yogyakarta; Semarang -> Kudus
Invalid input: 2 starting points: Solo Semarang
== A chain plus a loop
Input:
Bogor -> Depok; Bekasi -> Tangerang; Tangerang -> Bekasi
Invalid input: the tickets do not form one chain
```

Baris demi baris. `dataset.emplace(t.from, t.to)` menyisipkan hanya bila kuncinya baru dan mengembalikan pasangan yang `.second`-nya menyatakan apakah penyisipan terjadi; `false` di

sana berarti dua tiket meninggalkan kota yang sama, dan fungsi langsung melaporkannya. Titik awal dicari dengan bergelung atas tiket-tiket *menurut urutan masukan* sambil menanyakan `reverseMap.find(t.from)`. Kita bergelung atas masukan, bukan atas `dataset`, karena alasan yang penting di luar soal ini: urutan tabel hash mengunjungi kuncinya tidak ditentukan, dan program yang keluarannya bergantung pada urutan itu dapat mencetak baris berbeda pada kompilator berbeda. Dengan bergelung atas masukan, setiap baris keluaran menjadi deterministik, dan itulah pula sebabnya versi Java dan Python mencetak baris-baris yang persis sama. Penelusuran mencari kota saat ini dengan `dataset.find` dan berhenti ketika kota itu tidak memiliki tiket keluar, atau ketika telah memakai tiket sebanyak yang ada, yang melindunginya dari putaran yang tersembunyi di belakang titik awal (tumpukan terakhir). Perbandingan penutup `route.size() != tickets.size()` menangkap rantai yang berakhir terlalu cepat.

Ongkos. Membangun kedua *map* berarti n penyisipan masing-masing, mencari titik awal berarti n pencarian, dan penelusuran paling banyak n pencarian: rata-rata $O(n)$. Pencetakan di `solve` juga $O(n)$. Memorinya dua *map* berisi n entri masing-masing.

16.4.6 Penyelesaian Java dan Python

Kode 16.6 dan **Kode 16.7** mencetak keenam hasil yang sama. `putIfAbsent` di Java mengembalikan nilai lama, atau `null` bila belum ada, dan dengan cara itulah ia melaporkan tiket kedua dari kota yang sama; Python menguji `if frm in dataset` terlebih dahulu.

Kode 16.6 `P1602Itinerary.java`: Soal 16.2 dalam Java dengan dua `HashMap`.

```java
// P1602Itinerary.java - Problem 16.2: put tickets in travel order, with hashing.
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

public class P1602Itinerary {

    record Ticket(String from, String to) {}

    // Fills route with the tickets in travel order. Returns "" on success,
    // otherwise a sentence that says why the tickets cannot be put in order.
    static String findItinerary(List<Ticket> tickets, List<Ticket> route) {
        HashMap<String, String> dataset = new HashMap<>();     // from -> to
        HashMap<String, String> reverseMap = new HashMap<>();  // to -> from
        for (Ticket t : tickets) {
            if (dataset.putIfAbsent(t.from(), t.to()) != null)
                return "two tickets leave " + t.from();
            reverseMap.put(t.to(), t.from());
        }

        // The start is the only city that is left but never arrived at.
        List<String> starts = new ArrayList<>();
        for (Ticket t : tickets)
            if (!reverseMap.containsKey(t.from())) starts.add(t.from());
        if (starts.isEmpty()) return "no starting point (the tickets form a loop)";
        if (starts.size() > 1)
            return starts.size() + " starting points: " + String.join(" ", starts);

        // Follow the tickets: from the start, look up where each city leads.
        String city = starts.get(0);
        while (dataset.containsKey(city) && route.size() < tickets.size()) {
            String next = dataset.get(city);
            route.add(new Ticket(city, next));
            city = next;
        }
        if (route.size() != tickets.size())
            return "the tickets do not form one chain";
        return "";
    }
```

Kode 16.6 (Lanjutan) *P1602Itinerary.java: Soal 16.2 dalam Java dengan dua HashMap.*

```
    static void solve(String name, List<Ticket> tickets) {
        StringBuilder sb = new StringBuilder("== " + name + "\nInput:\n");
        for (int i = 0; i < tickets.size(); i++)
            sb.append(i > 0 ? "; " : "").append(tickets.get(i).from())
              .append(" -> ").append(tickets.get(i).to());
        List<Ticket> route = new ArrayList<>();
        String error = findItinerary(tickets, route);
        if (!error.isEmpty()) {
            System.out.println(sb + "\nInvalid input: " + error);
            return;
        }
        sb.append("\nStart: ").append(route.get(0).from()).append("\nOutput:\n");
        for (int i = 0; i < route.size(); i++)
            sb.append(i > 0 ? ", " : "").append(route.get(i).from())
              .append(" -> ").append(route.get(i).to());
        System.out.println(sb);
    }

    static Ticket t(String from, String to) {
        return new Ticket(from, to);
    }

    public static void main(String[] args) {
        solve("The given tickets",
                List.of(t("Cianjur", "Bandung"), t("Badung", "Denpasar"),
                        t("Gianyar", "Cianjur"), t("Denpasar", "Gianyar")));
        solve("The book van of Pustaka Ceria",
                List.of(t("Malang", "Batu"), t("Surabaya", "Sidoarjo"),
                        t("Pasuruan", "Malang"), t("Sidoarjo", "Pasuruan")));
        solve("A single ticket", List.of(t("Kediri", "Blitar")));
        solve("A loop", List.of(t("Bandung", "Jakarta"), t("Jakarta", "Bandung")));
        solve("Two separate trips",
                List.of(t("Solo", "Yogyakarta"), t("Semarang", "Kudus")));
        solve("A chain plus a loop", List.of(t("Bogor", "Depok"),
                t("Bekasi", "Tangerang"), t("Tangerang", "Bekasi")));
    }
}
```

Kode 16.7 `p1602_itinerary.py`: Soal 16.2 dalam Python dengan dua `dict`.

```python
# p1602_itinerary.py - Problem 16.2: put tickets in travel order, with hashing.


def find_itinerary(tickets):
    """Returns (route, "") on success, or ([], reason) if there is no chain."""
    dataset = {}                    # from -> to   (a dict is a hash table)
    reverse_map = {}                # to -> from
    for frm, to in tickets:
        if frm in dataset:
            return [], "two tickets leave " + frm
        dataset[frm] = to
        reverse_map[to] = frm

    # The start is the only city that is left but never arrived at.
    starts = [frm for frm, _ in tickets if frm not in reverse_map]
    if not starts:
        return [], "no starting point (the tickets form a loop)"
    if len(starts) > 1:
        return [], f"{len(starts)} starting points: " + " ".join(starts)

    # Follow the tickets: from the start, look up where each city leads.
    route, city = [], starts[0]
    while city in dataset and len(route) < len(tickets):
        route.append((city, dataset[city]))
        city = dataset[city]
    if len(route) != len(tickets):
        return [], "the tickets do not form one chain"
    return route, ""


def solve(name, tickets):
    print(f"== {name}\nInput:")
    print("; ".join(f"{a} -> {b}" for a, b in tickets))
    route, error = find_itinerary(tickets)
    if error:
        print("Invalid input:", error)
        return
    print("Start:", route[0][0])
    print("Output:")
    print(", ".join(f"{a} -> {b}" for a, b in route))
```

Kode 16.7 (Lanjutan) *p1602_itinerary.py: Soal 16.2 dalam Python dengan dua dict.*

```python
def main():
    solve("The given tickets", [("Cianjur", "Bandung"), ("Badung", "Denpasar"),
                                ("Gianyar", "Cianjur"), ("Denpasar", "Gianyar")])
    solve("The book van of Pustaka Ceria",
          [("Malang", "Batu"), ("Surabaya", "Sidoarjo"),
           ("Pasuruan", "Malang"), ("Sidoarjo", "Pasuruan")])
    solve("A single ticket", [("Kediri", "Blitar")])
    solve("A loop", [("Bandung", "Jakarta"), ("Jakarta", "Bandung")])
    solve("Two separate trips", [("Solo", "Yogyakarta"), ("Semarang", "Kudus")])
    solve("A chain plus a loop",
          [("Bogor", "Depok"), ("Bekasi", "Tangerang"), ("Tangerang", "Bekasi")])


if __name__ == "__main__":
    main()
```

16.4.7 Penyelesaian keliru yang menggoda

Jalan pintas yang menggoda adalah melompati Langkah 2 sama sekali dan memulai dari tiket pertama di tumpukan, dengan alasan bahwa tumpukan itu "daftar tiket" dan daftar dimulai dari depan:

```
string city = tickets[0].first;      // WRONG: "the first ticket is the first leg"
```

Dijalankan pada contohnya (`wrong/p1602_wrong.cpp`), program ini mencetak

```
Cianjur -> Bandung
```

lalu berhenti. Tiket pertama di tumpukan adalah Cianjur → Bandung, yang justru potongan *terakhir* perjalanan; dari Bandung tidak ada tiket, dan tiga perempat perjalanan hilang tanpa sepatah kata pun. Inti soal ini justru bahwa tumpukannya teracak, sehingga posisi sebuah tiket di masukan tidak mengatakan apa pun tentang posisinya dalam perjalanan. Kekeliruan kedua yang lebih halus bersembunyi di gelung pencetakan banyak penyelesaian: pencacah yang menentukan apakah tanda koma pemisah dicetak dideklarasikan sebagai `int a;` dan tidak pernah diberi nilai 0. Nilai pertamanya adalah sisa apa pun yang ada di memori itu, sehingga tiket pertama bisa mendapat koma di depannya, bisa juga tidak, dan Valgrind melaporkan "Conditional jump or move depends on uninitialised value(s)". Inisialisasikan setiap pencacah di tempat ia dideklarasikan, atau, seperti penyelesaian kita, tentukan pemisahnya dari indeks gelung.

16.4.8 Uji diri

> **UJI DIRI.** Tiket-tiketnya adalah Semarang → Solo, Jakarta → Cirebon, Solo → Madiun, Cirebon → Semarang, Madiun → Surabaya. Bangun `reverseMap`, cari titik awalnya, lalu tuliskan perjalanannya. Kemudian tambahkan tiket Surabaya → Jakarta: apa yang dilaporkan program kita, dan mengapa?
>
> *Jawaban.* `reverseMap`: Solo → Semarang, Cirebon → Jakarta, Madiun → Solo, Semarang → Cirebon, Surabaya → Madiun. Kota-kota asalnya Semarang, Jakarta, Solo, Cirebon, Madiun; hanya Jakarta yang bukan kunci `reverseMap`, jadi Jakarta adalah titik awal. Perjalanannya: `Jakarta -> Cirebon, Cirebon -> Semarang, Semarang -> Solo, Solo -> Madiun, Madiun -> Surabaya`. Dengan tambahan Surabaya → Jakarta, Jakarta pun menjadi kota tujuan, setiap kota asal menjadi kota tujuan, dan program melaporkan `no starting point (the tickets form a loop)`. Sebuah putaran tidak memiliki tiket pertama; kota mana pun dapat menjadi titik awalnya.

16.5 Soal Baru yang Dikerjakan: Graf

Ketiga soal dalam subbab ini memakai satu peta yang sama, yang digambar pada **Gambar 16.8**: delapan cabang jaringan Pustaka Ceria di Surabaya dan jalan-jalan yang boleh dilalui mobil buku. Pusat adalah perpustakaan induk; Benowo, cabang baru di ujung barat kota, belum terhubung dengan jalan mana pun yang dilalui mobil buku. Peta ini cukup kecil untuk ditelusuri dengan tangan dan memiliki semua yang dibutuhkan algoritma-algoritma untuk memperlihatkan wataknya: dua siklus, satu simpul yang tidak dapat dicapai, dan, pada Soal 16.5, sebuah jalan langsung yang *lebih lama* daripada jalan memutar.

16.5.1 Soal 16.3 — Lompatan paling sedikit antarcabang

> **SOAL.** Mobil buku berpindah antarcabang yang bertetangga melalui jalan Pusat–Gubeng, Pusat–Darmo, Pusat–Kenjeran, Gubeng–Sukolilo, Gubeng–Rungkut, Darmo–Wonokromo, Wonokromo–Rungkut, dan Kenjeran–Sukolilo; Benowo belum memiliki jalan. Setiap jalan

> dihitung satu lompatan. Untuk setiap cabang, tentukan jumlah lompatan paling sedikit dari Pusat dan cabang asal mobil pada rute semacam itu, lalu cetak rute ke Rungkut, ke Wonokromo, dan ke Benowo.

Dengan kata-kata kita sendiri. Simpul-simpulnya adalah kedelapan cabang, sisi-sisinya kedelapan jalan, tak berarah dan tak berbobot. Kita menginginkan *jarak* dari Pusat ke setiap simpul, diukur dalam banyaknya sisi, dan satu rute terpendek ke masing-masing. Itulah persis yang dihitung penelusuran melebar (Bab 13): ia menemukan simpul-simpul dalam cincin, mula-mula yang berjarak satu sisi, lalu yang berjarak dua sisi, dan seterusnya, dan pada saat pertama kali menemukan sebuah simpul ia telah menemukan rute terpendek ke simpul itu.

> **DEFINISI.** *Invarian BFS*: simpul-simpul di dalam *queue* tersusun menurut `dist` yang tidak menurun, dan yang pertama dengan yang terakhir berselisih paling banyak 1. Akibatnya, ketika simpul v ditemukan dari u, tidak ada rute yang lebih pendek ke v yang dapat ditemukan kemudian, dan `dist[v] = dist[u] + 1` sudah final.

Invarian itu menjelaskan satu baris terpenting dalam BFS: kita mengisi `dist[v]` *ketika pertama kali melihat* v, bukan ketika v keluar dari *queue*, dan tidak pernah mengubahnya lagi. Simpul yang `dist`-nya masih -1 belum terlihat; nilai lain berarti "sudah terlihat, dan final". Jadi larik yang sama berfungsi sebagai tabel jarak sekaligus tanda "sudah dikunjungi".

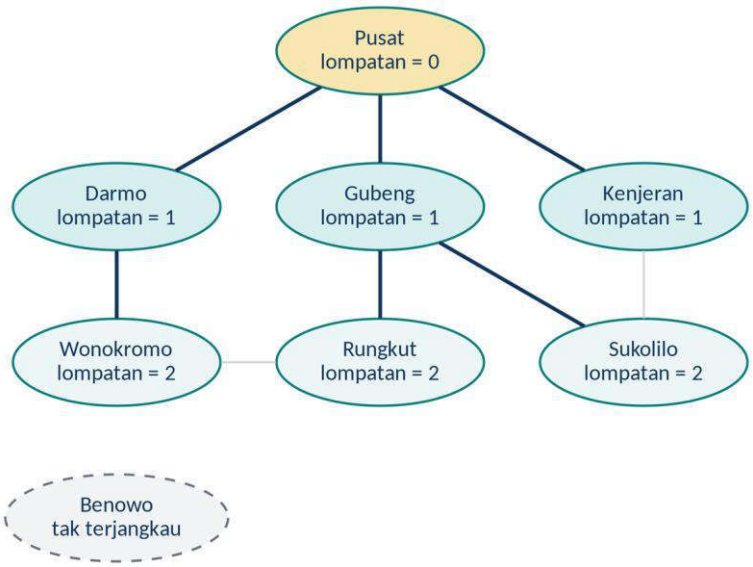
Dengan tangan, selangkah demi selangkah. *Adjacency list*-nya, menurut urutan yang dibangun program (setiap jalan ditambahkan ke kedua ujungnya, menurut urutan daftar jalan), adalah:

```
Adjacency lists:
  Pusat     : Gubeng Darmo Kenjeran
  Gubeng    : Pusat Sukolilo Rungkut
  Darmo     : Pusat Wonokromo
  Kenjeran  : Pusat Sukolilo
  Rungkut   : Gubeng Wonokromo
  Wonokromo : Darmo Rungkut
  Sukolilo  : Gubeng Kenjeran
  Benowo    :
```

Sekarang ikuti *queue*-nya.

1. `dist[Pusat] = 0`; *queue*: Pusat.
2. Ambil Pusat. Tetangganya Gubeng, Darmo, Kenjeran belum terlihat: masing-masing mendapat jarak 1 dan induk Pusat, lalu masuk *queue*. *Queue*: Gubeng, Darmo, Kenjeran.
3. Ambil Gubeng. Pusat sudah terlihat; Sukolilo dan Rungkut mendapat jarak 2 dan induk Gubeng. *Queue*: Darmo, Kenjeran, Sukolilo, Rungkut.
4. Ambil Darmo. Pusat sudah terlihat; Wonokromo mendapat jarak 2, induk Darmo. *Queue*: Kenjeran, Sukolilo, Rungkut, Wonokromo.
5. Ambil Kenjeran: Pusat dan Sukolilo sudah terlihat. Tidak ada yang berubah. Perhatikan bahwa Kenjeran–Sukolilo juga rute ke Sukolilo sepanjang 2, tetapi Sukolilo sudah berjarak 2 dari Gubeng, dan invarian memastikan jarak itu tidak dapat membaik.
6. Ambil Sukolilo, Rungkut, Wonokromo bergiliran: semua tetangganya sudah terlihat. Jalan Wonokromo–Rungkut diperiksa dua kali, sekali dari setiap ujungnya, dan tidak mengubah apa pun.
7. *Queue* kosong. Benowo tidak pernah terlihat: jaraknya tetap -1, "tak terjangkau".

Gambar 16.8 menggambar hasilnya: sisi-sisi gelap adalah *pohon BFS*, yaitu sisi-sisi tempat setiap simpul pertama kali ditemukan, sedangkan sisi-sisi pucat diperiksa tetapi tidak dipakai.



Gambar 16.8 Peta cabang ditelusuri melebar dari Pusat: simpul-simpul dalam cincin berjarak sama, pohon BFS dalam garis gelap, dua jalan yang tidak dipakai berwarna pucat, dan Benowo tak terjangkau.

Untuk mencetak rute ke Rungkut, mulailah dari Rungkut dan ikuti induk-induknya mundur: Rungkut berasal dari Gubeng, Gubeng dari Pusat, Pusat tidak berinduk. Dibalik, rutenya Pusat → Gubeng → Rungkut, dua lompatan.

Ongkos. Setiap simpul masuk *queue* paling banyak sekali, karena ditandai saat masuk, dan ketika keluar, *adjacency list*-nya dibaca sekali. *Adjacency list* sebuah graf tak berarah memuat $2|E|$ entri seluruhnya (setiap sisi muncul di kedua ujungnya), sehingga seluruh pekerjaannya

$$T_{\text{BFS}} = O(|V| + |E|). \tag{16.5}$$

Untuk peta kita, Persamaan (16.5) bernilai $8 + 2 \cdot 8 = 24$ langkah dasar. Matriks ketetanggaan akan memaksa penelusuran memeriksa semua $|V|$ sel dalam satu baris untuk setiap simpul, $O(|V|^2)$, dan itulah sebabnya BFS ditulis dengan *adjacency list* (Bab 13).

Penyelesaian C++. Kode 16.8 membangun *adjacency list* dari daftar jalan dan menjalankan BFS dari Pusat.

Kode 16.8 `p1603_bfs_hops.cpp`: penelusuran melebar dari Pusat, dengan lompatan paling sedikit, induk setiap cabang, dan tiga rute.

```cpp
// p1603_bfs_hops.cpp - Problem 16.3: fewest hops between library branches (BFS).
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <queue>
#include <string>
#include <vector>
using namespace std;

const vector<string> NAME = {"Pusat", "Gubeng", "Darmo", "Kenjeran",
                             "Rungkut", "Wonokromo", "Sukolilo", "Benowo"};

// Breadth-first search from s. dist[v] = fewest edges from s (-1: unreachable),
// parent[v] = the vertex from which v was first reached (-1: none).
void bfs(const vector<vector<int>>& adj, int s, vector<int>& dist,
         vector<int>& parent, vector<int>& order) {
    dist.assign(adj.size(), -1);
    parent.assign(adj.size(), -1);
    queue<int> q;
    dist[s] = 0;                        // mark s BEFORE it waits in the queue
    q.push(s);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (int v : adj[u]) {
            if (dist[v] == -1) {        // first sight of v: its distance is final
                dist[v] = dist[u] + 1;
                parent[v] = u;
                q.push(v);
            }
        }
    }
}

void printRoute(const vector<int>& dist, const vector<int>& parent, int t) {
    cout << "Fewest hops to " << NAME[t] << ": ";
    if (dist[t] == -1) {
        cout << "no route\n";
        return;
    }
    vector<int> path;
    for (int v = t; v != -1; v = parent[v]) path.push_back(v);   // walk back
    reverse(path.begin(), path.end());
    for (size_t i = 0; i < path.size(); i++)
        cout << (i ? " -> " : "") << NAME[path[i]];
```

Kode 16.8 (Lanjutan) *p1603_bfs_hops.cpp: penelusuran melebar dari Pusat, dengan lompatan paling sedikit, induk setiap cabang, dan tiga rute.*

```
    cout << " (" << dist[t] << " hops)\n";
}

int main() {
    int n = NAME.size();
    vector<pair<int, int>> roads = {{0, 1}, {0, 2}, {0, 3}, {1, 6},
                                    {1, 4}, {2, 5}, {5, 4}, {3, 6}};
    vector<vector<int>> adj(n);
    for (auto [u, v] : roads) {         // an undirected road is two arcs
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    cout << "Adjacency lists:\n";
    for (int u = 0; u < n; u++) {
        cout << "  " << left << setw(10) << NAME[u] << ":";
        for (int v : adj[u]) cout << " " << NAME[v];
        cout << "\n";
    }

    vector<int> dist, parent, order;
    bfs(adj, 0, dist, parent, order);
    cout << "\nOrder in which BFS takes the branches out of the queue:\n ";
    for (int u : order) cout << " " << NAME[u] << "(" << dist[u] << ")";
    cout << "\n\n" << left << setw(12) << "Branch" << setw(6) << "Hops"
         << "Reached from\n";
    for (int v = 0; v < n; v++) {
        cout << setw(12) << NAME[v];
        if (dist[v] == -1) {
            cout << "unreachable\n";
            continue;
        }
        cout << setw(6) << dist[v] << (parent[v] == -1 ? "-" : NAME[parent[v]])
             << "\n";
    }
    cout << "\n";
    printRoute(dist, parent, 4);
    printRoute(dist, parent, 5);
    printRoute(dist, parent, 7);
    return 0;
}
```

```
Adjacency lists:
  Pusat     : Gubeng Darmo Kenjeran
  Gubeng    : Pusat Sukolilo Rungkut
  Darmo     : Pusat Wonokromo
  Kenjeran  : Pusat Sukolilo
  Rungkut   : Gubeng Wonokromo
  Wonokromo : Darmo Rungkut
  Sukolilo  : Gubeng Kenjeran
  Benowo    :

Order in which BFS takes the branches out of the queue:
  Pusat(0) Gubeng(1) Darmo(1) Kenjeran(1) Sukolilo(2) Rungkut(2) Wonokromo(2)

Branch      Hops  Reached from
Pusat       0     -
Gubeng      1     Pusat
Darmo       1     Pusat
Kenjeran    1     Pusat
Rungkut     2     Gubeng
Wonokromo   2     Darmo
Sukolilo    2     Gubeng
Benowo      unreachable

Fewest hops to Rungkut: Pusat -> Gubeng -> Rungkut (2 hops)
Fewest hops to Wonokromo: Pusat -> Darmo -> Wonokromo (2 hops)
Fewest hops to Benowo: no route
```

Baris demi baris. `for (auto [u, v] : roads)` memakai *structured binding* C++17 untuk menamai kedua ujung setiap jalan; kedua `push_back` menambahkan jalan itu ke dua arah. Di dalam `bfs`, `dist.assign(n, -1)` sekaligus menentukan ukuran dan mengisi larik; `dist[s] = 0` diisi sebelum `s` masuk *queue*, aturan yang sama dengan untuk simpul lain. `if (dist[v] == -1)` di bagian dalam adalah seluruh algoritmanya: tetangga yang belum terlihat mendapat jarak final dan induknya lalu masuk *queue*. `printRoute` menelusuri induk-induk mundur ke dalam sebuah *vector* lalu membaliknya, karena induk-induk mengarah dari tujuan ke sumber, sedangkan kita ingin mencetak dari sumber.

Penyelesaian Java dan Python. **Kode 16.9** dan **Kode 16.10** mencetak baris-baris yang sama. Java memakai `ArrayDeque` sebagai *queue* (`add` di belakang, `poll` dari depan); Python memakai `collections.deque` (`append`, `popleft`), tidak pernah `list` biasa dengan `pop(0)`, yang menggeser setiap elemen sisanya dan mengubah *dequeue* $O(1)$ menjadi $O(n)$ [6], [7].

Kode 16.9 `P1603BfsHops.java`: Soal 16.3 dalam Java.

```java
// P1603BfsHops.java - Problem 16.3: fewest hops between library branches (BFS).
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class P1603BfsHops {

    static final String[] NAME = {"Pusat", "Gubeng", "Darmo", "Kenjeran",
                                  "Rungkut", "Wonokromo", "Sukolilo", "Benowo"};

    // Breadth-first search from s. dist[v] = fewest edges from s (-1: unreachable),
    // parent[v] = the vertex from which v was first reached (-1: none).
    static List<Integer> bfs(List<List<Integer>> adj, int s, int[] dist,
                             int[] parent) {
        Arrays.fill(dist, -1);
        Arrays.fill(parent, -1);
        List<Integer> order = new ArrayList<>();
        ArrayDeque<Integer> q = new ArrayDeque<>();
        dist[s] = 0;                          // mark s BEFORE it waits in the queue
        q.add(s);
        while (!q.isEmpty()) {
            int u = q.poll();
            order.add(u);
            for (int v : adj.get(u)) {
                if (dist[v] == -1) {          // first sight of v: distance is final
                    dist[v] = dist[u] + 1;
                    parent[v] = u;
                    q.add(v);
                }
            }
        }
        return order;
    }
```

Kode 16.9 (Lanjutan) *P1603BfsHops.java: Soal 16.3 dalam Java.*

```
    static void printRoute(int[] dist, int[] parent, int t) {
        System.out.print("Fewest hops to " + NAME[t] + ": ");
        if (dist[t] == -1) {
            System.out.println("no route");
            return;
        }
        List<String> path = new ArrayList<>();
        for (int v = t; v != -1; v = parent[v]) path.add(NAME[v]);   // walk back
        Collections.reverse(path);
        System.out.println(String.join(" -> ", path) + " (" + dist[t] + " hops)");
    }

    public static void main(String[] args) {
        int n = NAME.length;
        int[][] roads = {{0, 1}, {0, 2}, {0, 3}, {1, 6},
                         {1, 4}, {2, 5}, {5, 4}, {3, 6}};
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < n; i++) adj.add(new ArrayList<>());
        for (int[] r : roads) {             // an undirected road is two arcs
            adj.get(r[0]).add(r[1]);
            adj.get(r[1]).add(r[0]);
        }
        System.out.println("Adjacency lists:");
        for (int u = 0; u < n; u++) {
            StringBuilder sb = new StringBuilder(
                    String.format("  %-10s:", NAME[u]));
            for (int v : adj.get(u)) sb.append(" ").append(NAME[v]);
            System.out.println(sb);
        }

        int[] dist = new int[n], parent = new int[n];
        List<Integer> order = bfs(adj, 0, dist, parent);
        System.out.println(
                "\nOrder in which BFS takes the branches out of the queue:");
        StringBuilder sb = new StringBuilder(" ");
        for (int u : order)
            sb.append(" ").append(NAME[u]).append("(").append(dist[u]).append(")");
        System.out.println(sb);
        System.out.printf("%n%-12s%-6s%s%n", "Branch", "Hops", "Reached from");
        for (int v = 0; v < n; v++) {
            if (dist[v] == -1) {
                System.out.printf("%-12sunreachable%n", NAME[v]);
                continue;
            }
            System.out.printf("%-12s%-6d%s%n", NAME[v], dist[v],
                    parent[v] == -1 ? "-" : NAME[parent[v]]);
```

Kode 16.9 (Lanjutan) *P1603BfsHops.java: Soal 16.3 dalam Java.*

```
        }
        System.out.println();
        printRoute(dist, parent, 4);
        printRoute(dist, parent, 5);
        printRoute(dist, parent, 7);
    }
}
```

Kode 16.10 `p1603_bfs_hops.py`: Soal 16.3 dalam Python.

```python
# p1603_bfs_hops.py - Problem 16.3: fewest hops between library branches (BFS).
from collections import deque

NAME = ["Pusat", "Gubeng", "Darmo", "Kenjeran",
        "Rungkut", "Wonokromo", "Sukolilo", "Benowo"]


def bfs(adj, s):
    """dist[v] = fewest edges from s (-1: unreachable); parent[v] = where v was
    first reached from (-1: none); order = the order vertices leave the queue."""
    dist = [-1] * len(adj)
    parent = [-1] * len(adj)
    order = []
    dist[s] = 0                          # mark s BEFORE it waits in the queue
    q = deque([s])
    while q:
        u = q.popleft()
        order.append(u)
        for v in adj[u]:
            if dist[v] == -1:            # first time we see v: its distance is final
                dist[v] = dist[u] + 1
                parent[v] = u
                q.append(v)
    return dist, parent, order


def print_route(dist, parent, t):
    print(f"Fewest hops to {NAME[t]}: ", end="")
    if dist[t] == -1:
        print("no route")
        return
    path, v = [], t
    while v != -1:                       # walk back along the parents
        path.append(NAME[v])
        v = parent[v]
    print(" -> ".join(reversed(path)), f"({dist[t]} hops)")
```

Kode 16.10 (Lanjutan) *p1603_bfs_hops.py: Soal 16.3 dalam Python.*

```
def main():
    n = len(NAME)
    roads = [(0, 1), (0, 2), (0, 3), (1, 6), (1, 4), (2, 5), (5, 4), (3, 6)]
    adj = [[] for _ in range(n)]
    for u, v in roads:                  # an undirected road is two arcs
        adj[u].append(v)
        adj[v].append(u)
    print("Adjacency lists:")
    for u in range(n):
        print(f"  {NAME[u]:<10}:" + "".join(" " + NAME[v] for v in adj[u]))

    dist, parent, order = bfs(adj, 0)
    print("\nOrder in which BFS takes the branches out of the queue:")
    print(" " + "".join(f" {NAME[u]}({dist[u]})" for u in order))
    print(f"\n{'Branch':<12}{'Hops':<6}Reached from")
    for v in range(n):
        if dist[v] == -1:
            print(f"{NAME[v]:<12}unreachable")
            continue
        came_from = "-" if parent[v] == -1 else NAME[parent[v]]
        print(f"{NAME[v]:<12}{dist[v]:<6}{came_from}")
    print()
    print_route(dist, parent, 4)
    print_route(dist, parent, 5)
    print_route(dist, parent, 7)


if __name__ == "__main__":
    main()
```

Penyelesaian keliru yang menggoda. Penelusuran mendalam lebih pendek ditulis, rekursif dan anggun, dan juga mengunjungi setiap simpul yang terjangkau. Jadi mengapa tidak menjadikan kedalaman saat DFS mencapai sebuah simpul sebagai jaraknya?

```
  void dfs(int u, int d) {                  // WRONG: depth in a DFS is not a distance
      dist[u] = d;
      for (int v : adj[u]) if (dist[v] == -1) dfs(v, d + 1);
  }
```

`wrong/p1603_wrong.cpp` mencetak

```
Pusat 0
Gubeng 1
Darmo 4
Kenjeran 3
Rungkut 2
Wonokromo 3
Sukolilo 2
```

DFS dari Pusat pergi ke Gubeng (1), lalu, sebelum melihat tetangga Pusat yang lain, menyelam ke Sukolilo (2) dan Kenjeran (3), mundur, pergi ke Rungkut (2), Wonokromo (3), dan akhirnya Darmo (4). Darmo hanya berjarak satu jalan dari Pusat, tetapi DFS kebetulan mencapainya paling akhir, melalui jalan memutar yang terpanjang. DFS menjawab “dapatkah saya sampai ke sana?” dengan benar, dan merupakan alat yang tepat untuk siklus, komponen, dan urutan topologis; DFS tidak menjawab “berapa lompatan paling sedikit?”. Hanya urutan cincin demi cincin dari sebuah *queue* yang menjamin bahwa rute pertama yang ditemukan adalah rute terpendek.

> **UJI DIRI.** Pemerintah kota membuka jalan Darmo–Benowo, ditambahkan setelah kedelapan jalan itu. Berikan urutan BFS yang baru, jarak dan induk Benowo, serta rute ke sana.
>
> *Jawaban. Adjacency list* Darmo menjadi Pusat, Wonokromo, Benowo, sehingga Benowo ditemukan dari Darmo tepat setelah Wonokromo, dengan jarak 2. Urutan BFS-nya Pusat,

> Gubeng, Darmo, Kenjeran, Sukolilo, Rungkut, Wonokromo, Benowo, dan rutenya Pusat → Darmo → Benowo, 2 lompatan. Tidak ada yang lain yang berubah.

16.5.2 Soal 16.4 — Urutan membaca topik-topik

> **SOAL.** Topik-topik buku ini saling bergantung: larik mendahului pencarian, pengurutan, *stack*, dan pointer; pointer dan *stack* mendahului *linked list*; *linked list* mendahului rekursi, pohon, graf, dan hashing; rekursi dan pencarian mendahului pohon; pohon dan pengurutan mendahului *heap*; *stack* mendahului graf; pencarian mendahului hashing. Cetak urutan membaca kesebelas topik sehingga setiap topik datang setelah semua topik yang diperlukannya. Kemudian tambahkan syarat "graf sebelum pointer" dan laporkan apa yang terjadi.

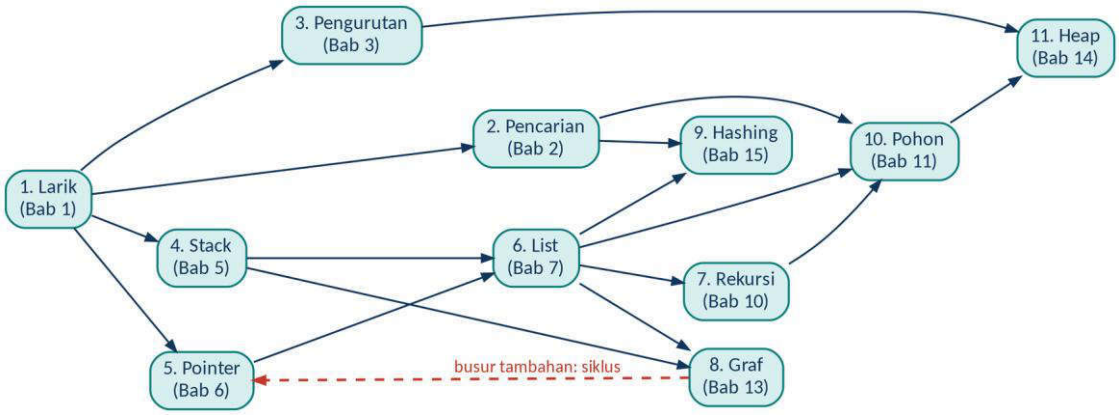
Dengan kata-kata kita sendiri. Setiap syarat "baca u sebelum v" adalah sebuah *busur* $u \to v$ pada graf berarah. Urutan simpul-simpul yang membuat setiap busur menunjuk ke depan disebut *urutan topologis* (Bab 13). Urutan itu ada tepat ketika graf tidak memiliki siklus berarah, dan graf semacam itu disebut *directed acyclic graph* (DAG). Kita memakai algoritma Kahn [10]: hitung, untuk setiap topik, berapa prasyarat yang masih ditunggunya (*in-degree*-nya); masukkan setiap topik dengan hitungan 0 ke *queue*; berulang kali ambil satu topik dari *queue*, tambahkan ke urutan, dan kurangi 1 dari hitungan setiap topik yang memerlukannya; topik yang hitungannya mencapai 0 masuk *queue*.

> **DEFINISI.** *Invarian Kahn*: setiap topik yang sudah berada di urutan memiliki semua prasyaratnya lebih awal di urutan itu, dan `indeg[v]` sama dengan banyaknya prasyarat v yang belum berada di urutan. Jika *queue* kosong padahal masih ada topik yang belum masuk, setiap topik yang tertinggal menunggu topik lain yang juga tertinggal, sehingga mengikuti prasyarat mundur di antara mereka pasti suatu saat mengulang sebuah topik: ada siklus berarah di antara mereka atau di depan mereka, dan tidak ada urutan yang mungkin.

Kalimat terakhir definisi itu adalah alasan adanya satu baris yang ditinggalkan penyelesaian yang ceroboh: *setelah* gelung, bandingkan panjang urutan dengan banyaknya topik.

Dengan tangan, selangkah demi selangkah. **Gambar 16.9** memperlihatkan kesebelas topik dan keenam belas busurnya, dengan setiap topik diberi label posisinya dalam urutan yang dihasilkan algoritma Kahn. *In-degree* awalnya: Arrays 0; Searching, Sorting, Stacks, Pointers, Recursion 1; Lists, Graphs, Heaps, Hashing 2; Trees 3.

1. *Queue*: Arrays. Ambil. Searching, Sorting, Stacks, Pointers turun ke 0. *Queue*: Searching, Sorting, Stacks, Pointers.
2. Ambil Searching: Trees 2, Hashing 1. Ambil Sorting: Heaps 1. Ambil Stacks: Lists 1, Graphs 1. Ambil Pointers: Lists 0, masuk. *Queue*: Lists.
3. Ambil Lists: Recursion 0, Trees 1, Graphs 0, Hashing 0; tiga masuk. *Queue*: Recursion, Graphs, Hashing.
4. Ambil Recursion: Trees 0, masuk. Ambil Graphs, Hashing. Ambil Trees: Heaps 0, masuk. Ambil Heaps. *Queue* kosong dan kesebelas topik telah ditempatkan.



Gambar 16.9 Kesebelas topik dan keenam belas busur "baca sebelum", setiap topik bernomor menurut posisinya dalam urutan yang ditemukan algoritma Kahn; busur merah putus-putus "graf sebelum pointer" menutup siklus Pointer, List, Graf, Pointer.

Urutan itu menempatkan Graphs dan Hashing sebelum Trees, yang bukan urutan bab dalam buku ini. Namun urutan itu tetap sah untuk keenam belas busur ini: tidak ada yang mengatakan bahwa pohon harus mendahului graf. Sebuah DAG biasanya memiliki banyak urutan topologis, dan algoritma Kahn mengembalikan urutan yang dihasilkan oleh disiplin *queue*-nya. Sekarang tambahkan busur putus-putus Graphs → Pointers. Pointers kini menunggu dua topik, Arrays dan Graphs; Graphs menunggu Lists, yang menunggu Pointers. Setelah Arrays, Searching, Sorting, dan Stacks, *queue* kosong, dan tujuh topik tertinggal: tiga yang berada di siklus dan empat (Recursion, Trees, Heaps, Hashing) yang bergantung padanya.

Ongkos. Menghitung *in-degree* membaca setiap busur sekali; setiap topik masuk dan keluar *queue* paling banyak sekali; setiap busur diikuti sekali ketika pangkalnya keluar dari *queue*. Totalnya $O(|V| + |E|)$, sama seperti BFS pada Persamaan (16.5).

Penyelesaian C++. Kode 16.11 membangun grafnya, mencetak setiap topik beserta topik-topik yang memerlukannya, lalu melaporkan sebuah urutan membaca atau topik-topik yang tidak dapat ditempatkan.

Kode 16.11 `p1604_reading_order.cpp`: algoritma Kahn pada kesebelas topik buku, dengan dan tanpa busur yang menciptakan siklus.

```cpp
// p1604_reading_order.cpp - Problem 16.4: an order to read the topics (Kahn).
#include <iomanip>
#include <iostream>
#include <queue>
#include <string>
#include <vector>
using namespace std;

const vector<string> TOPIC = {"Arrays", "Searching", "Sorting", "Stacks",
                              "Pointers", "Lists", "Recursion", "Trees",
                              "Graphs", "Heaps", "Hashing"};
const vector<int> CHAPTER = {1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15};

// Kahn's algorithm. Returns the topics in an order in which every arc u -> v
// has u before v. If some topics are missing from the result, they lie on
// or behind a cycle and no valid order exists.
vector<int> kahn(const vector<vector<int>>& adj) {
    int n = adj.size();
    vector<int> indeg(n, 0);
    for (int u = 0; u < n; u++)
        for (int v : adj[u]) indeg[v]++;
    queue<int> ready;                          // topics with no unread prerequisite
    for (int u = 0; u < n; u++)
        if (indeg[u] == 0) ready.push(u);
    vector<int> order;
    while (!ready.empty()) {
        int u = ready.front();
        ready.pop();
        order.push_back(u);
        for (int v : adj[u])
            if (--indeg[v] == 0) ready.push(v);    // v's last prerequisite is read
    }
    return order;
}

void report(const vector<vector<int>>& adj) {
    vector<int> order = kahn(adj);
    int n = adj.size();
    if ((int)order.size() == n) {
        cout << "A reading order:\n";
        for (int i = 0; i < n; i++)
            cout << setw(4) << i + 1 << ". " << left << setw(10) << TOPIC[order[i]]
                 << right << " (Chapter " << CHAPTER[order[i]] << ")\n";
        return;
    }
    vector<bool> placed(n, false);
```

Kode 16.11 (Lanjutan) *p1604_reading_order.cpp: algoritma Kahn pada kesebelas topik buku, dengan dan tanpa busur yang menciptakan siklus.*

```
    for (int u : order) placed[u] = true;
    cout << "No reading order exists: " << n - order.size()
         << " topics wait for each other in a cycle or behind one:\n ";
    for (int u = 0; u < n; u++)
        if (!placed[u]) cout << " " << TOPIC[u];
    cout << "\n";
}

int main() {
    int n = TOPIC.size();
    vector<pair<int, int>> needs = {           // {u, v}: read u before v
        {0, 1}, {0, 2}, {0, 3}, {0, 4}, {4, 5}, {3, 5}, {5, 6}, {5, 7}, {6, 7},
        {1, 7}, {3, 8}, {5, 8}, {7, 9}, {2, 9}, {5, 10}, {1, 10}};
    vector<vector<int>> adj(n);
    for (auto [u, v] : needs) adj[u].push_back(v);

    cout << "Each topic, and the topics that need it:\n";
    for (int u = 0; u < n; u++) {
        cout << "  " << left << setw(10) << TOPIC[u] << right << "->";
        for (int v : adj[u]) cout << " " << TOPIC[v];
        cout << "\n";
    }
    cout << "\n";
    report(adj);

    cout << "\nNow suppose someone insists that Graphs must come before "
            "Pointers.\n";
    adj[8].push_back(4);
    report(adj);
    return 0;
}
```

```
Each topic, and the topics that need it:
  Arrays    -> Searching Sorting Stacks Pointers
  Searching -> Trees Hashing
  Sorting   -> Heaps
  Stacks    -> Lists Graphs
  Pointers  -> Lists
  Lists     -> Recursion Trees Graphs Hashing
  Recursion -> Trees
  Trees     -> Heaps
  Graphs    ->
  Heaps     ->
  Hashing   ->

A reading order:
   1. Arrays     (Chapter 1)
   2. Searching  (Chapter 2)
   3. Sorting    (Chapter 3)
   4. Stacks     (Chapter 5)
   5. Pointers   (Chapter 6)
   6. Lists      (Chapter 7)
   7. Recursion  (Chapter 10)
   8. Graphs     (Chapter 13)
   9. Hashing    (Chapter 15)
  10. Trees      (Chapter 11)
  11. Heaps      (Chapter 14)

Now suppose someone insists that Graphs must come before Pointers.
No reading order exists: 7 topics wait for each other in a cycle or behind one:
  Pointers Lists Recursion Trees Graphs Heaps Hashing
```

Baris demi baris. `kahn` hanya mengembalikan urutan; `report`-lah yang menentukan makna urutan itu, dengan membandingkan `order.size()` dengan `n`. Ungkapan `--indeg[v] == 0` mengurangi dan menguji dalam satu langkah: bernilai benar tepat ketika topik itu baru saja kehilangan prasyarat terakhirnya yang belum terbaca. Topik-topik yang tidak pernah sampai ke *queue* ditemukan dengan menandai yang sudah ditempatkan lalu mendaftar sisanya.

Penyelesaian Java dan Python. **Kode 16.12** dan **Kode 16.13** mencetak baris-baris yang sama.

Kode 16.12 `P1604ReadingOrder.java`: Soal 16.4 dalam Java.

```java
// P1604ReadingOrder.java - Problem 16.4: an order to read the topics (Kahn).
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.List;

public class P1604ReadingOrder {

    static final String[] TOPIC = {"Arrays", "Searching", "Sorting", "Stacks",
                                   "Pointers", "Lists", "Recursion", "Trees",
                                   "Graphs", "Heaps", "Hashing"};
    static final int[] CHAPTER = {1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15};

    // Kahn's algorithm. Returns the topics in an order in which every arc u -> v
    // has u before v. If some topics are missing from the result, they lie on
    // or behind a cycle and no valid order exists.
    static List<Integer> kahn(List<List<Integer>> adj) {
        int n = adj.size();
        int[] indeg = new int[n];
        for (List<Integer> out : adj)
            for (int v : out) indeg[v]++;
        ArrayDeque<Integer> ready = new ArrayDeque<>();   // no unread prerequisite
        for (int u = 0; u < n; u++)
            if (indeg[u] == 0) ready.add(u);
        List<Integer> order = new ArrayList<>();
        while (!ready.isEmpty()) {
            int u = ready.poll();
            order.add(u);
            for (int v : adj.get(u))
                if (--indeg[v] == 0) ready.add(v);    // v's last prerequisite read
        }
        return order;
    }

    static void report(List<List<Integer>> adj) {
        List<Integer> order = kahn(adj);
        int n = adj.size();
        if (order.size() == n) {
            System.out.println("A reading order:");
            for (int i = 0; i < n; i++)
                System.out.printf("%4d. %-10s (Chapter %d)%n", i + 1,
                        TOPIC[order.get(i)], CHAPTER[order.get(i)]);
            return;
        }
        boolean[] placed = new boolean[n];
        for (int u : order) placed[u] = true;
        System.out.println("No reading order exists: " + (n - order.size())
```

Kode 16.12 (Lanjutan) *P1604ReadingOrder.java: Soal 16.4 dalam Java.*

```
                + " topics wait for each other in a cycle or behind one:");
        StringBuilder sb = new StringBuilder(" ");
        for (int u = 0; u < n; u++)
            if (!placed[u]) sb.append(" ").append(TOPIC[u]);
        System.out.println(sb);
    }

    public static void main(String[] args) {
        int n = TOPIC.length;
        int[][] needs = {                          // {u, v}: read u before v
            {0, 1}, {0, 2}, {0, 3}, {0, 4}, {4, 5}, {3, 5}, {5, 6}, {5, 7}, {6, 7},
            {1, 7}, {3, 8}, {5, 8}, {7, 9}, {2, 9}, {5, 10}, {1, 10}};
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < n; i++) adj.add(new ArrayList<>());
        for (int[] e : needs) adj.get(e[0]).add(e[1]);

        System.out.println("Each topic, and the topics that need it:");
        for (int u = 0; u < n; u++) {
            StringBuilder sb = new StringBuilder(
                    String.format("  %-10s->", TOPIC[u]));
            for (int v : adj.get(u)) sb.append(" ").append(TOPIC[v]);
            System.out.println(sb);
        }
        System.out.println();
        report(adj);

        System.out.println("\nNow suppose someone insists that Graphs must come "
                + "before Pointers.");
        adj.get(8).add(4);
        report(adj);
    }
}
```

Kode 16.13 `p1604_reading_order.py`: Soal 16.4 dalam Python.

```python
# p1604_reading_order.py - Problem 16.4: an order to read the topics (Kahn).
from collections import deque

TOPIC = ["Arrays", "Searching", "Sorting", "Stacks", "Pointers", "Lists",
         "Recursion", "Trees", "Graphs", "Heaps", "Hashing"]
CHAPTER = [1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15]


def kahn(adj):
    """Returns the topics in an order in which every arc u -> v has u before v.
    Topics missing from the result lie on or behind a cycle."""
    n = len(adj)
    indeg = [0] * n
    for out in adj:
        for v in out:
            indeg[v] += 1
    ready = deque(u for u in range(n) if indeg[u] == 0)   # no unread prerequisite
    order = []
    while ready:
        u = ready.popleft()
        order.append(u)
        for v in adj[u]:
            indeg[v] -= 1
            if indeg[v] == 0:           # v's last prerequisite has been read
                ready.append(v)
    return order


def report(adj):
    order = kahn(adj)
    n = len(adj)
    if len(order) == n:
        print("A reading order:")
        for i, u in enumerate(order, 1):
            print(f"{i:4d}. {TOPIC[u]:<10} (Chapter {CHAPTER[u]})")
        return
    placed = set(order)
    print(f"No reading order exists: {n - len(order)} topics wait for each other"
          " in a cycle or behind one:")
    print(" " + "".join(" " + TOPIC[u] for u in range(n) if u not in placed))
```

Kode 16.13 (Lanjutan) *p1604_reading_order.py: Soal 16.4 dalam Python.*

```python
def main():
    n = len(TOPIC)
    needs = [(0, 1), (0, 2), (0, 3), (0, 4), (4, 5), (3, 5), (5, 6), (5, 7), (6, 7),
             (1, 7), (3, 8), (5, 8), (7, 9), (2, 9), (5, 10), (1, 10)]  # u before v
    adj = [[] for _ in range(n)]
    for u, v in needs:
        adj[u].append(v)

    print("Each topic, and the topics that need it:")
    for u in range(n):
        print(f"  {TOPIC[u]:<10}->" + "".join(" " + TOPIC[v] for v in adj[u]))
    print()
    report(adj)

    print("\nNow suppose someone insists that Graphs must come before Pointers.")
    adj[8].append(4)
    report(adj)


if __name__ == "__main__":
    main()
```

Penyelesaian keliru yang menggoda. Tinggalkan perbandingan penutup itu dan cetak apa pun yang dihasilkan gelung. Dengan siklus, `wrong/p1604_wrong.cpp` mencetak

```
Reading order: Arrays Searching Sorting Stacks
```

tanpa peringatan apa pun. Pembaca yang memercayainya akan membaca empat topik lalu bertanya-tanya ke mana tujuh sisanya. Pada masukan tanpa siklus program itu sempurna, dan karena itulah kelalaian ini lolos dari pengujian dengan contoh; Langkah 5 metode kita, yang mendaftar "siklus" di antara kasus tepi sebuah graf, adalah yang menangkapnya.

> **UJI DIRI.** Apakah Arrays, Pointers, Stacks, Lists, Searching, Hashing, Sorting, Recursion, Trees, Heaps, Graphs merupakan urutan membaca yang sah untuk keenam belas busur itu (tanpa busur tambahan)? Dapatkah topik selain Arrays dibaca pertama?
>
> *Jawaban.* Ya, urutan itu sah: periksa setiap busur, misalnya Lists (ke-4) datang setelah Pointers (ke-2) dan Stacks (ke-3), Hashing (ke-6) setelah Lists dan Searching (ke-5), Trees (ke-9) setelah Lists, Recursion (ke-8), dan Searching, Heaps (ke-10) setelah Trees dan Sorting (ke-7), dan Graphs (ke-11) setelah Stacks dan Lists. Tidak ada topik lain yang dapat datang pertama: Arrays satu-satunya topik ber-*in-degree* 0.

16.5.3 Soal 16.5 — Rute tercepat untuk mobil buku

> **SOAL.** Jalan-jalan pada Soal 16.3 kini memiliki waktu tempuh dalam menit: Pusat–Gubeng 5, Pusat–Darmo 9, Pusat–Kenjeran 14, Gubeng–Sukolilo 12, Gubeng–Rungkut 6, Darmo–Wonokromo 4, Wonokromo–Rungkut 3, Kenjeran–Sukolilo 4, dan jalan langsung Pusat–Rungkut 20 menit. Tentukan waktu tercepat dari Pusat ke setiap cabang, cabang asal mobil, dan rute tercepat ke Rungkut, Sukolilo, dan Benowo, dengan memakai *priority queue*.

Dengan kata-kata kita sendiri. Pertanyaannya bukan lagi lompatan paling sedikit: jalan langsung ke Rungkut hanya satu lompatan tetapi 20 menit, sedangkan Pusat → Gubeng → Rungkut dua lompatan dan 11 menit. Dengan panjang yang tidak negatif, algoritma Dijkstra [11] menjawab pertanyaan ini. Algoritma ini memelihara waktu sementara `dist[v]` untuk setiap cabang, dimulai dari ∞ kecuali `dist[Pusat] = 0`, lalu mengulang: ambil cabang u dengan waktu sementara terkecil yang belum *selesai*, nyatakan ia selesai, lalu *relaksasi* setiap jalan u–v: jika `dist[u] + w < dist[v]`, jalan itu menawarkan cara lebih cepat ke v, sehingga `dist[v]` diturunkan dan u dicatat sebagai induknya. Bab 13 mengerjakan "ambil yang terkecil" dengan pemindaian biasa atas semua cabang; Bab 14 menunjukkan bahwa *min-heap* mengerjakannya dalam $O(\lg n)$. Kita memakai *heap*, dengan *penghapusan malas* (*lazy deletion*): ketika `dist[v]` turun, kita cukup memasukkan entri baru $(\text{dist}[v], v)$, dan ketika entri untuk cabang yang sudah selesai keluar dari *heap*, kita mengenalinya sebagai basi dan melewatinya.

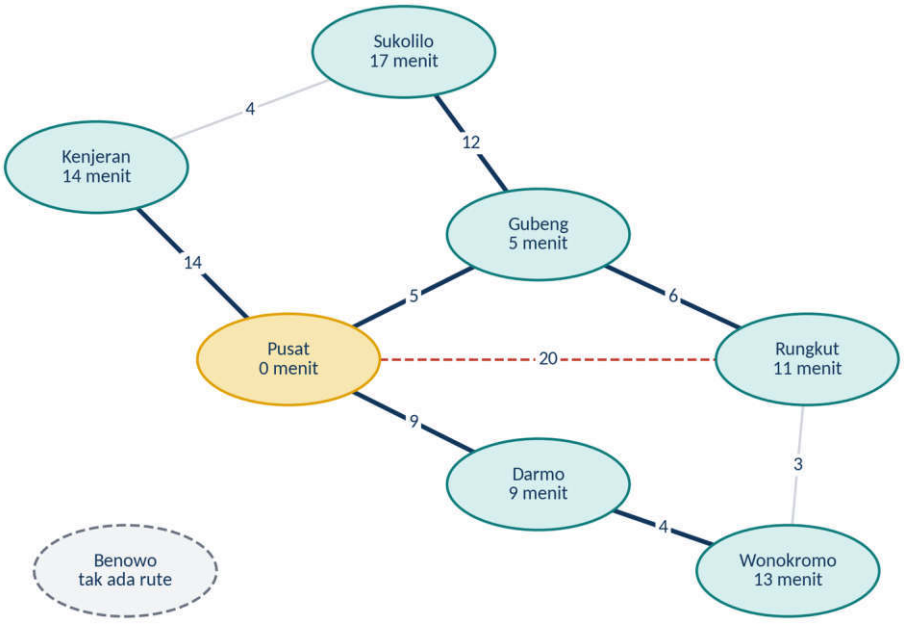
> **DEFINISI.** *Invarian Dijkstra*: untuk setiap cabang yang sudah selesai, `dist` adalah panjang rute tercepat dari sumber, dan setiap `dist` sementara milik cabang yang belum selesai adalah panjang rute tercepat yang cabang terakhirnya satu-satunya yang belum selesai. Cabang yang diambil dari *heap* dengan waktu sementara terkecil kemudian dapat dinyatakan selesai, karena setiap rute lain ke sana harus meninggalkan wilayah yang sudah selesai melalui cabang yang waktunya sudah paling sedikit sama besar, dan panjang jalan tidak pernah negatif.

Ungkapan "panjang jalan tidak pernah negatif" bekerja sungguh-sungguh: dengan jalan bernilai negatif, sebuah rute dapat meninggalkan wilayah yang sudah selesai pada waktu yang lebih besar namun tetap tiba lebih awal, dan invariannya gagal. Bab 13 memperlihatkan kegagalan itu pada contoh kecil; Bellman–Ford adalah algoritma untuk kasus tersebut [9].

Dengan tangan, selangkah demi selangkah. Program mencetak jejaknya sendiri: setiap cabang yang diambilnya dari *heap*, dan setiap perbaikan yang dibuatnya.

```
take Pusat (0)
  Gubeng: inf -> 5
  Darmo: inf -> 9
  Kenjeran: inf -> 14
  Rungkut: inf -> 20
take Gubeng (5)
  Sukolilo: inf -> 17
  Rungkut: 20 -> 11
take Darmo (9)
  Wonokromo: inf -> 13
take Rungkut (11)
take Wonokromo (13)
take Kenjeran (14)
take Sukolilo (17)
skip Rungkut (20): an old, stale entry
```

Bacalah dengan invarian dalam pikiran. Pusat diambil pada 0 dan menawarkan 5, 9, 14, dan 20. Waktu sementara terkecil adalah 5 milik Gubeng, sehingga Gubeng selesai: tidak ada rute melalui Darmo (sudah 9) atau Kenjeran (14) yang dapat mencapai Gubeng dalam waktu kurang. Dari Gubeng, Rungkut membaik dari 20 menjadi $5 + 6 = 11$; entri (20, Rungkut) tetap di *heap* tetapi kini basi. Darmo diambil pada 9 dan menawarkan Wonokromo 13. Rungkut diambil pada 11, dan jalannya ke Wonokromo menawarkan $11 + 3 = 14$, tidak lebih baik daripada 13. Wonokromo pada 13, Kenjeran pada 14 (jalannya menawarkan Sukolilo $14 + 4 = 18$, tidak lebih baik daripada 17), Sukolilo pada 17. Terakhir, (20, Rungkut) yang basi keluar dan dilewati. **Gambar 16.10** memperlihatkan waktu-waktu akhirnya dan, dalam garis gelap, *pohon lintasan terpendek*: setiap cabang beserta jalan yang memberinya waktu akhir.



Gambar 16.10 Waktu tercepat dari Pusat. Jalan-jalan gelap membentuk pohon lintasan terpendek; jalan langsung Pusat–Rungkut yang merah putus-putus (20 menit) kalah oleh jalan memutar melalui Gubeng (11 menit).

Ongkos. Setiap relaksasi yang berhasil memasukkan satu entri, sehingga *heap* menerima paling banyak $1 + 2|E|$ entri, dan setiap *push* atau *pop* berongkos $O(\lg|E|) = O(\lg|V|)$, karena $|E| < |V|^2$. Membaca setiap *adjacency list* sekali menambah $O(|V| + |E|)$, sehingga

$$T_{\text{Dijkstra}} = O((|V| + |E|)\lg|V|). \tag{16.6}$$

Versi Bab 13 yang memindai mencari waktu terkecil berongkos $O(|V|^2)$. Untuk peta jalan, yang $|E|$-nya kelipatan kecil dari $|V|$, Persamaan (16.6) jauh lebih kecil: untuk sejuta persimpangan, sekitar dua puluh juta langkah dibandingkan satu triliun.

Penyelesaian C++. Kode 16.14 adalah algoritma Dijkstra dengan `std::priority_queue`. *Priority queue* C++ secara bawaan adalah *max-heap* (**Tabel 16.3**); `greater<Entry>` mengubahnya menjadi *min-heap* atas pasangan (waktu, cabang).

Kode 16.14 `p1605_dijkstra.cpp`: algoritma Dijkstra dengan *binary min-heap* dan penghapusan malas, menjejak setiap langkah.

```
// p1605_dijkstra.cpp - Problem 16.5: quickest routes for the book van (Dijkstra).
#include <algorithm>
#include <climits>
#include <functional>
#include <iomanip>
#include <iostream>
#include <queue>
#include <string>
#include <vector>
using namespace std;

const vector<string> NAME = {"Pusat", "Gubeng", "Darmo", "Kenjeran",
                             "Rungkut", "Wonokromo", "Sukolilo", "Benowo"};
const int INF = INT_MAX;

struct Arc {
    int to, minutes;
};

string show(int d) { return d == INF ? "inf" : to_string(d); }

// Dijkstra's algorithm with a binary min-heap and "lazy deletion":
// a vertex may sit in the heap several times; only its smallest entry counts.
void dijkstra(const vector<vector<Arc>>& adj, int s, vector<int>& dist,
              vector<int>& parent) {
    dist.assign(adj.size(), INF);
    parent.assign(adj.size(), -1);
    vector<bool> done(adj.size(), false);
    using Entry = pair<int, int>;                       // (distance, vertex)
    priority_queue<Entry, vector<Entry>, greater<Entry>> heap;   // smallest on top
    dist[s] = 0;
    heap.push({0, s});
    while (!heap.empty()) {
        auto [d, u] = heap.top();
        heap.pop();
        if (done[u]) {                                  // an old, larger entry
            cout << "skip " << NAME[u] << " (" << d << "): an old, stale entry\n";
            continue;
        }
        done[u] = true;                                 // d is now final for u
        cout << "take " << NAME[u] << " (" << d << ")\n";
        for (const Arc& a : adj[u]) {
            if (!done[a.to] && d + a.minutes < dist[a.to]) {
                cout << "  " << NAME[a.to] << ": " << show(dist[a.to]) << " -> "
                     << d + a.minutes << "\n";
                dist[a.to] = d + a.minutes;
```

Kode 16.14 (Lanjutan) *p1605_dijkstra.cpp: algoritma Dijkstra dengan binary min-heap dan penghapusan malas, menjejak setiap langkah.*

```
                parent[a.to] = u;
                heap.push({dist[a.to], a.to});
            }
        }
    }
}

void printRoute(const vector<int>& dist, const vector<int>& parent, int t) {
    cout << "Quickest route to " << NAME[t] << ": ";
    if (dist[t] == INF) {
        cout << "no route\n";
        return;
    }
    vector<int> path;
    for (int v = t; v != -1; v = parent[v]) path.push_back(v);
    reverse(path.begin(), path.end());
    for (size_t i = 0; i < path.size(); i++)
        cout << (i ? " -> " : "") << NAME[path[i]];
    cout << " (" << dist[t] << " minutes)\n";
}

int main() {
    int n = NAME.size();
    int roads[][3] = {{0, 1, 5}, {0, 2, 9}, {0, 3, 14}, {1, 6, 12}, {1, 4, 6},
                      {2, 5, 4}, {5, 4, 3}, {3, 6, 4}, {0, 4, 20}};
    vector<vector<Arc>> adj(n);
    for (auto& r : roads) {                          // two-way roads
        adj[r[0]].push_back({r[1], r[2]});
        adj[r[1]].push_back({r[0], r[2]});
    }
    vector<int> dist, parent;
    dijkstra(adj, 0, dist, parent);

    cout << "\n" << left << setw(12) << "Branch" << setw(9) << "Minutes" << "Via\n";
    for (int v = 0; v < n; v++)
        cout << setw(12) << NAME[v] << setw(9) << show(dist[v])
             << (parent[v] == -1 ? "-" : NAME[parent[v]]) << "\n";
    cout << "\n";
    printRoute(dist, parent, 4);
    printRoute(dist, parent, 6);
    printRoute(dist, parent, 7);
    return 0;
}
```

```
take Pusat (0)
  Gubeng: inf -> 5
  Darmo: inf -> 9
  Kenjeran: inf -> 14
  Rungkut: inf -> 20
take Gubeng (5)
  Sukolilo: inf -> 17
  Rungkut: 20 -> 11
take Darmo (9)
  Wonokromo: inf -> 13
take Rungkut (11)
take Wonokromo (13)
take Kenjeran (14)
take Sukolilo (17)
skip Rungkut (20): an old, stale entry

Branch      Minutes  Via
Pusat       0        -
Gubeng      5        Pusat
Darmo       9        Pusat
Kenjeran    14       Pusat
Rungkut     11       Gubeng
Wonokromo   13       Darmo
Sukolilo    17       Gubeng
Benowo      inf      -

Quickest route to Rungkut: Pusat -> Gubeng -> Rungkut (11 minutes)
Quickest route to Sukolilo: Pusat -> Gubeng -> Sukolilo (17 minutes)
Quickest route to Benowo: no route
```

Baris demi baris. `INT_MAX` berperan sebagai ∞, dan `show` mencetaknya sebagai `inf`; program tidak pernah menambahkan apa pun pada `INT_MAX`, karena cabang yang waktunya tak hingga tidak pernah diambil dari *heap* (ia tidak pernah dimasukkan). `if (done[u])` adalah pengujian entri basi. Syarat `!done[a.to] && d + a.minutes < dist[a.to]` adalah relaksasinya; paruh pertamanya menghemat perbandingan yang sia-sia, karena cabang yang sudah selesai tidak mungkin membaik. Pasangan dibandingkan menurut anggota pertama lalu anggota kedua, sehingga waktu yang sama diputuskan oleh nomor cabang yang lebih kecil, di ketiga bahasa secara sama.

Penyelesaian Java dan Python. **Kode 16.15** dan **Kode 16.16** mencetak jejak yang sama. `PriorityQueue` di Java sudah merupakan *min-heap*, dan entrinya pasangan `int[]` dengan pembanding eksplisit; `heapq` di Python bekerja pada *tuple*, yang dibandingkan elemen demi elemen.

Kode 16.15 `P1605Dijkstra.java`: Soal 16.5 dalam Java.

```java
// P1605Dijkstra.java - Problem 16.5: quickest routes for the book van (Dijkstra).
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.PriorityQueue;

public class P1605Dijkstra {

    static final String[] NAME = {"Pusat", "Gubeng", "Darmo", "Kenjeran",
                                   "Rungkut", "Wonokromo", "Sukolilo", "Benowo"};
    static final int INF = Integer.MAX_VALUE;

    record Arc(int to, int minutes) {}

    static String show(int d) {
        return d == INF ? "inf" : Integer.toString(d);
    }

    // Dijkstra's algorithm with a binary min-heap and "lazy deletion":
    // a vertex may sit in the heap several times; only its smallest entry counts.
    static void dijkstra(List<List<Arc>> adj, int s, int[] dist, int[] parent) {
        Arrays.fill(dist, INF);
        Arrays.fill(parent, -1);
        boolean[] done = new boolean[adj.size()];
        PriorityQueue<int[]> heap = new PriorityQueue<>(      // {distance, vertex}
                (a, b) -> a[0] != b[0] ? Integer.compare(a[0], b[0])
                                       : Integer.compare(a[1], b[1]));
        dist[s] = 0;
        heap.add(new int[] {0, s});
        while (!heap.isEmpty()) {
            int[] top = heap.poll();
            int d = top[0], u = top[1];
            if (done[u]) {                                    // an old, larger entry
                System.out.println("skip " + NAME[u] + " (" + d
                        + "): an old, stale entry");
                continue;
            }
            done[u] = true;                                   // d is now final for u
            System.out.println("take " + NAME[u] + " (" + d + ")");
            for (Arc a : adj.get(u)) {
                if (!done[a.to()] && d + a.minutes() < dist[a.to()]) {
                    System.out.println("  " + NAME[a.to()] + ": "
                            + show(dist[a.to()]) + " -> " + (d + a.minutes()));
                    dist[a.to()] = d + a.minutes();
                    parent[a.to()] = u;
```

Kode 16.15 (Lanjutan) *P1605Dijkstra.java: Soal 16.5 dalam Java.*

```
                    heap.add(new int[] {dist[a.to()], a.to()});
                }
            }
        }
    }

    static void printRoute(int[] dist, int[] parent, int t) {
        System.out.print("Quickest route to " + NAME[t] + ": ");
        if (dist[t] == INF) {
            System.out.println("no route");
            return;
        }
        List<String> path = new ArrayList<>();
        for (int v = t; v != -1; v = parent[v]) path.add(NAME[v]);
        Collections.reverse(path);
        System.out.println(String.join(" -> ", path) + " (" + dist[t]
                + " minutes)");
    }

    public static void main(String[] args) {
        int n = NAME.length;
        int[][] roads = {{0, 1, 5}, {0, 2, 9}, {0, 3, 14}, {1, 6, 12}, {1, 4, 6},
                         {2, 5, 4}, {5, 4, 3}, {3, 6, 4}, {0, 4, 20}};
        List<List<Arc>> adj = new ArrayList<>();
        for (int i = 0; i < n; i++) adj.add(new ArrayList<>());
        for (int[] r : roads) {                          // two-way roads
            adj.get(r[0]).add(new Arc(r[1], r[2]));
            adj.get(r[1]).add(new Arc(r[0], r[2]));
        }
        int[] dist = new int[n], parent = new int[n];
        dijkstra(adj, 0, dist, parent);

        System.out.printf("%n%-12s%-9s%s%n", "Branch", "Minutes", "Via");
        for (int v = 0; v < n; v++)
            System.out.printf("%-12s%-9s%s%n", NAME[v], show(dist[v]),
                    parent[v] == -1 ? "-" : NAME[parent[v]]);
        System.out.println();
        printRoute(dist, parent, 4);
        printRoute(dist, parent, 6);
        printRoute(dist, parent, 7);
    }
}
```

Kode 16.16 `p1605_dijkstra.py`: Soal 16.5 dalam Python.

```python
# p1605_dijkstra.py - Problem 16.5: quickest routes for the book van (Dijkstra).
import heapq

NAME = ["Pusat", "Gubeng", "Darmo", "Kenjeran",
        "Rungkut", "Wonokromo", "Sukolilo", "Benowo"]
INF = float("inf")


def show(d):
    return "inf" if d == INF else str(d)


def dijkstra(adj, s):
    """Dijkstra's algorithm with a binary min-heap and lazy deletion: a vertex
    may sit in the heap several times; only its smallest entry counts."""
    dist = [INF] * len(adj)
    parent = [-1] * len(adj)
    done = [False] * len(adj)
    dist[s] = 0
    heap = [(0, s)]                                  # (distance, vertex)
    while heap:
        d, u = heapq.heappop(heap)
        if done[u]:                                  # an old, larger entry
            print(f"skip {NAME[u]} ({d}): an old, stale entry")
            continue
        done[u] = True                               # d is now final for u
        print(f"take {NAME[u]} ({d})")
        for v, minutes in adj[u]:
            if not done[v] and d + minutes < dist[v]:
                print(f"  {NAME[v]}: {show(dist[v])} -> {d + minutes}")
                dist[v] = d + minutes
                parent[v] = u
                heapq.heappush(heap, (dist[v], v))
    return dist, parent
```

Kode 16.16 (Lanjutan) *p1605_dijkstra.py: Soal 16.5 dalam Python.*

```
def print_route(dist, parent, t):
    print(f"Quickest route to {NAME[t]}: ", end="")
    if dist[t] == INF:
        print("no route")
        return
    path, v = [], t
    while v != -1:
        path.append(NAME[v])
        v = parent[v]
    print(" -> ".join(reversed(path)), f"({dist[t]} minutes)")


def main():
    n = len(NAME)
    roads = [(0, 1, 5), (0, 2, 9), (0, 3, 14), (1, 6, 12), (1, 4, 6),
             (2, 5, 4), (5, 4, 3), (3, 6, 4), (0, 4, 20)]
    adj = [[] for _ in range(n)]
    for u, v, w in roads:                          # two-way roads
        adj[u].append((v, w))
        adj[v].append((u, w))
    dist, parent = dijkstra(adj, 0)

    print(f"\n{'Branch':<12}{'Minutes':<9}Via")
    for v in range(n):
        via = "-" if parent[v] == -1 else NAME[parent[v]]
        print(f"{NAME[v]:<12}{show(dist[v]):<9}{via}")
    print()
    print_route(dist, parent, 4)
    print_route(dist, parent, 6)
    print_route(dist, parent, 7)


if __name__ == "__main__":
    main()
```

Penyelesaian keliru yang menggoda. Penyederhanaan yang menggoda adalah memperlakukan Dijkstra seperti BFS: beri sebuah cabang waktunya pada saat pertama kali terlihat, dan jangan pernah menoleh lagi.

```
   if (!seen[v]) { seen[v] = true; dist[v] = d + w; heap.push({dist[v], v}); }   // WRONG
```

`wrong/p1605_wrong.cpp` mencetak

```
Pusat 0
Gubeng 5
Darmo 9
Kenjeran 14
Rungkut 20
Wonokromo 13
Sukolilo 17
```

Rungkut pertama kali terlihat dari Pusat, melalui jalan 20 menit, dan dikunci pada 20 bahkan sebelum Gubeng, yang menawarkan 11, sempat diambil. Pada BFS, penglihatan pertama itu final karena setiap sisi bernilai 1 dan *queue* menemui simpul-simpul menurut urutan jarak. Dengan adanya panjang, simpul yang terlihat lebih awal masih dapat dicapai lebih cepat kemudian; sebuah waktu baru final ketika cabangnya keluar dari *heap* sebagai yang terkecil. Itulah persis perbedaan antara kedua invarian.

> **UJI DIRI.** Jalur baru satu menit dibuka antara Darmo dan Rungkut. Waktu dan induk mana yang berubah?
>
> *Jawaban.* Darmo diambil pada 9 dan kini menawarkan Rungkut $9 + 1 = 10 < 11$, sehingga Rungkut menjadi 10 dengan induk Darmo. Ketika Rungkut diambil pada 10, jalannya ke

Wonokromo menawarkan $10 + 3 = 13$, yang tidak *kurang dari* 13, sehingga Wonokromo tetap 13 dengan induk Darmo. Waktu lainnya tidak berubah: Pusat 0, Gubeng 5, Darmo 9, Kenjeran 14, Wonokromo 13, Sukolilo 17, Benowo tak terjangkau.

16.6 Soal Baru yang Dikerjakan: Heap

Soal 16.1 membangun *heap* operasi demi operasi. Kedua soal dalam subbab ini memakai *heap* sebagai alat, dan keduanya memakainya dengan cara yang mengejutkan pemula: soal pertama memelihara *min-heap* untuk menemukan nilai-nilai *terbesar*, dan soal kedua memelihara dua *heap* yang saling berhadapan untuk menemukan nilai yang berada di *tengah*.

16.6.1 Soal 16.6 — k nilai terbesar dengan heap berukuran k

SOAL. (a) Delapan buku dipinjam 12, 45, 7, 30, 18, 3, 26, dan 40 kali bulan ini (kode B1001 sampai B1008). Tentukan tiga yang paling sering dipinjam dengan *heap* yang tidak pernah memuat lebih dari tiga entri, dan perlihatkan *heap* itu setelah setiap buku. (b) Tentukan lima buku dengan penilaian terbanyak dari katalog sepuluh ribu buku dengan cara yang sama, lalu periksa jawabannya dengan mengurutkan seluruh katalog.

Dengan kata-kata kita sendiri. Mengurutkan semuanya lalu mengambil k yang pertama memang berhasil, dengan ongkos waktu $O(n \lg n)$ dan salinan semua n nilai. Bila k jauh lebih kecil daripada n, sebagian besar pekerjaan itu mengurutkan nilai-nilai yang akan dibuang. Sebagai gantinya, simpan k yang terbaik sejauh ini di dalam *heap*, dan ajukan kepada setiap nilai baru satu pertanyaan: *apakah ia lebih baik daripada yang terburuk di antara yang saya simpan?* Yang terburuk di antara yang disimpan harus dapat ditemukan seketika, sehingga *heap*-nya adalah **min**-*heap*: akarnya adalah yang terkecil dari k nilai terbesar sejauh ini, nilai yang berada di ambang dibuang. Nilai baru yang tidak lebih besar daripada akar tidak mungkin termasuk k teratas dan diabaikan dalam $O(1)$; nilai yang lebih besar menggantikan akar, yang lalu tenggelam ke tempatnya dalam $O(\lg k)$.

DEFINISI. *Invarian top-k*: setelah i nilai pertama diajukan, *heap* memuat $\min(i, k)$ nilai terbesar di antara mereka, dan akarnya adalah nilai terkecil yang dimuatnya.

Dengan tangan, selangkah demi selangkah. **Gambar 16.11** mengikuti bagian (a). Tiga buku pertama mengisi *heap* tanpa syarat: 12; 12, 45; 7, 12, 45. Sejak itu setiap buku dibandingkan dengan akar. B1004 (30) mengalahkan akar 7: buang 7, simpan 30. B1005 (18) mengalahkan 12. B1006 (3) tidak mengalahkan 18 dan diabaikan dengan ongkos satu perbandingan. B1007 (26) mengalahkan 18; B1008 (40) mengalahkan 26. *Heap* berakhir dengan 30, 40, 45, dan bila dibaca dari yang terbesar: B1002 (45), B1008 (40), B1004 (30).



Gambar 16.11 Tiga buku yang paling sering dipinjam, ditemukan dengan *min-heap* berukuran 3. Setiap hitungan baru dibandingkan dengan akar, yang terkecil dari tiga yang disimpan; jika lebih besar, akarnya dibuang.

Ongkos. Setiap nilai berongkos satu perbandingan dengan akar, dan paling banyak satu penggantian seharga $O(\lg k)$, sehingga

$$T_{\text{top-}k} = O(n \lg k), \qquad \text{memori tambahan } O(k). \tag{16.7}$$

Untuk $n = 10{,}000$ dan $k = 5$, $\lg k$ sekitar 2,3 dibandingkan $\lg n \approx 13.3$ untuk pengurutan, dan memorinya lima entri alih-alih sepuluh ribu. Berapa banyak penggantian yang terjadi bergantung pada urutan masukan: dalam urutan terburuk, yaitu naik, *setiap* nilai setelah k yang pertama adalah rekor baru dan menyebabkan penggantian; dalam urutan terbaik, yaitu turun, tidak ada satu pun; dan dalam urutan acak, nilai ke-i termasuk k terbesar sejauh ini dengan peluang k/i, sehingga harapan banyaknya penggantian adalah

$$\sum_{i=k+1}^{n} \frac{k}{i} \approx k \ln \frac{n}{k}, \tag{16.8}$$

yang menurut Persamaan (16.8) sekitar $5 \ln 2000 \approx 38$ untuk katalog kita: beberapa puluh perbaikan di antara sepuluh ribu nilai.

Penyelesaian C++. **Kode 16.17** menyimpan *heap* di dalam `vector` dengan algoritma standar `push_heap` dan `pop_heap`, agar dapat mencetak isi *heap* sambil berjalan; `greater<Item>` membuat keduanya membangun *min-heap*. Bagian (b) membaca katalog tiga kali dalam urutan berbeda dan menghitung penggantian setiap kali.

Kode 16.17 `p1606_top_k.cpp`: k nilai terbesar dengan *min-heap* berukuran k, dijejak pada delapan hitungan peminjaman dan dijalankan pada seluruh katalog dalam tiga urutan.

```cpp
// p1606_top_k.cpp - Problem 16.6: the k largest values with a min-heap of size k.
#include <algorithm>
#include <fstream>
#include <functional>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
using namespace std;

using Item = pair<long long, string>;              // (value, name)

// A min-heap kept in a vector: heap.front() is the SMALLEST of the k kept so far.
struct TopK {
    size_t k;
    vector<Item> heap;
    int replaced = 0;
    // Returns true if the item was kept.
    bool offer(const Item& x) {
        if (heap.size() < k) {                     // not full yet: always keep
            heap.push_back(x);
            push_heap(heap.begin(), heap.end(), greater<Item>());
            return true;
        }
        if (x <= heap.front()) return false;       // not above the smallest kept
        pop_heap(heap.begin(), heap.end(), greater<Item>());   // drop the smallest
        heap.back() = x;
        push_heap(heap.begin(), heap.end(), greater<Item>());
        replaced++;
        return true;
    }
    vector<Item> largestFirst() const {
        vector<Item> v = heap;
        sort(v.rbegin(), v.rend());
        return v;
    }
    string contents() const {                      // smallest first, for tracing
        vector<Item> v = heap;
        sort(v.begin(), v.end());
        string s;
        for (const Item& x : v) s += " " + to_string(x.first);
        return s;
    }
};
```

Kode 16.17 (Lanjutan) *p1606_top_k.cpp: k nilai terbesar dengan min-heap berukuran k, dijejak pada delapan hitungan peminjaman dan dijalankan pada seluruh katalog dalam tiga urutan.*

```
int main() {
    cout << "Part 1: the three most-borrowed books this month (k = 3)\n";
    vector<Item> loans = {{12, "B1001"}, {45, "B1002"}, {7, "B1003"},
                          {30, "B1004"}, {18, "B1005"}, {3, "B1006"},
                          {26, "B1007"}, {40, "B1008"}};
    TopK small{3, {}};
    for (const Item& x : loans) {
        long long before = small.heap.size() == 3 ? small.heap.front().first : -1;
        bool kept = small.offer(x);
        cout << "  " << x.second << " (" << x.first << "): ";
        if (!kept)
            cout << "not more than the smallest kept (" << before << "), ignored\n";
        else
            cout << "heap holds" << small.contents() << "\n";
    }
    cout << "Top 3:";
    for (const Item& x : small.largestFirst())
        cout << " " << x.second << " (" << x.first << ")";
    cout << "\n\nPart 2: the five most-rated books of the catalog (k = 5)\n";

    ifstream in("../data/catalog.tsv");
    if (!in) {
        cout << "cannot open ../data/catalog.tsv\n";
        return 1;
    }
    string line;
    getline(in, line);                              // skip the header
    vector<pair<long long, Item>> books;            // (id, (ratings, title))
    while (getline(in, line)) {
        stringstream ss(line);
        string id, year, ratings, title;
        getline(ss, id, '\t');
        getline(ss, year, '\t');
        getline(ss, ratings, '\t');
        getline(ss, title, '\t');
        books.push_back({stoll(id), {stoll(ratings), title}});
    }
    cout << "Read " << books.size() << " books.\n";
```

Kode 16.17 (Lanjutan) *p1606_top_k.cpp: k nilai terbesar dengan min-heap berukuran k, dijejak pada delapan hitungan peminjaman dan dijalankan pada seluruh katalog dalam tiga urutan.*

```
    TopK inFileOrder{5, {}};                        // the file: most-rated first
    for (const auto& b : books) inFileOrder.offer(b.second);
    sort(books.begin(), books.end());               // now in order of identifier
    TopK top{5, {}};
    for (const auto& b : books) top.offer(b.second);
    vector<Item> rising;                            // the worst order: fewest first
    for (const auto& b : books) rising.push_back(b.second);
    sort(rising.begin(), rising.end());
    TopK worst{5, {}};
    for (const Item& x : rising) worst.offer(x);
    cout << "Changes to the heap after it filled up:\n  " << inFileOrder.replaced
         << " in file order (most-rated first), " << top.replaced
         << " in identifier order,\n  " << worst.replaced
         << " in increasing order of ratings.\n";

    vector<Item> best = top.largestFirst();
    for (size_t i = 0; i < best.size(); i++)
        cout << "  " << i + 1 << ". " << best[i].first << "  " << best[i].second
             << "\n";

    vector<Item>& all = rising;                     // the slow way: sort everything
    reverse(all.begin(), all.end());
    bool same = equal(best.begin(), best.end(), all.begin());
    cout << "Same five as sorting all " << all.size() << " books: "
         << (same ? "yes" : "no") << "\n";
    return 0;
}
```

```
Part 1: the three most-borrowed books this month (k = 3)
  B1001 (12): heap holds 12
  B1002 (45): heap holds 12 45
  B1003 (7): heap holds 7 12 45
  B1004 (30): heap holds 12 30 45
  B1005 (18): heap holds 18 30 45
  B1006 (3): not more than the smallest kept (18), ignored
  B1007 (26): heap holds 26 30 45
  B1008 (40): heap holds 30 40 45
Top 3: B1002 (45) B1008 (40) B1004 (30)

Part 2: the five most-rated books of the catalog (k = 5)
Read 10000 books.
Changes to the heap after it filled up:
  0 in file order (most-rated first), 9 in identifier order,
  9995 in increasing order of ratings.
  1. 4942365  The Hunger Games (The Hunger Games, #1)
  2. 4800065  Harry Potter and the Sorcerer's Stone (Harry Potter, #1)
  3. 3916824  Twilight (Twilight, #1)
  4. 3340896  To Kill a Mockingbird
  5. 2773745  The Great Gatsby
Same five as sorting all 10000 books: yes
```

Baris demi baris. `offer` adalah seluruh algoritmanya. Selama *heap* memiliki kurang dari *k* entri, ia memasukkan tanpa syarat. Sesudahnya `x <= heap.front()` menolak apa pun yang tidak mengalahkan akar; `pop_heap` memindahkan akar ke belakang *vector* dan memulihkan *heap* pada sisanya, sel belakang ditimpa dengan butir baru, dan `push_heap` membiarkannya naik ke tempatnya. Hasil pada katalog memperlihatkan Persamaan (16.8) beserta kedua ekstremnya dalam angka nyata. Berkasnya mendaftar buku dari yang penilaiannya terbanyak ke bawah, urutan terbaik yang mungkin: setelah lima yang pertama, tidak ada yang mengalahkan akar, dan *heap* tidak pernah berubah. Dalam urutan naik menurut banyaknya penilaian, urutan terburuk, setiap satu dari 9.995 buku lainnya menyebabkan penggantian. Dalam urutan identitas, *heap*

hanya berubah 9 kali, bahkan lebih sedikit daripada 38 yang akan diberikan urutan acak, karena buku-buku tertua dan paling terkenal di katalog memiliki identitas terkecil. Baris terakhir memastikan bahwa kelima buku yang disimpan tepat sama dengan lima buku pertama dari katalog yang diurutkan penuh.

Penyelesaian Java dan Python. **Kode 16.18** dan **Kode 16.19** mencetak baris-baris yang sama. `PriorityQueue` di Java dan `heapq` di Python sudah merupakan *min-heap*; `heapq.heapreplace` di Python mengeluarkan akar dan memasukkan butir baru dalam satu *sift-down*, persis "ganti akar" kita [7].

Kode 16.18 `P1606TopK.java`: Soal 16.6 dalam Java.

```java
// P1606TopK.java - Problem 16.6: the k largest values with a min-heap of size k.
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.PriorityQueue;

public class P1606TopK {

    record Item(long value, String name) {}

    static final Comparator<Item> ORDER =
            Comparator.comparingLong(Item::value).thenComparing(Item::name);

    // A min-heap of at most k items: heap.peek() is the SMALLEST of those kept.
    static class TopK {
        final int k;
        final PriorityQueue<Item> heap = new PriorityQueue<>(ORDER);
        int replaced = 0;

        TopK(int k) {
            this.k = k;
        }

        boolean offer(Item x) {                    // true if the item was kept
            if (heap.size() < k) {                 // not full yet: always keep
                heap.add(x);
                return true;
            }
            if (ORDER.compare(x, heap.peek()) <= 0)
                return false;                      // not above the smallest kept
            heap.poll();                           // drop the smallest
            heap.add(x);
            replaced++;
            return true;
        }

        List<Item> largestFirst() {
            List<Item> v = new ArrayList<>(heap);
            v.sort(ORDER.reversed());
            return v;
        }
```

Kode 16.18 (Lanjutan) *P1606TopK.java: Soal 16.6 dalam Java.*

```
        String contents() {                         // smallest first, for tracing
            List<Item> v = new ArrayList<>(heap);
            v.sort(ORDER);
            StringBuilder sb = new StringBuilder();
            for (Item x : v) sb.append(" ").append(x.value());
            return sb.toString();
        }
    }

    public static void main(String[] args) throws IOException {
        System.out.println(
                "Part 1: the three most-borrowed books this month (k = 3)");
        long[] counts = {12, 45, 7, 30, 18, 3, 26, 40};
        TopK small = new TopK(3);
        for (int i = 0; i < counts.length; i++) {
            Item x = new Item(counts[i], "B100" + (i + 1));
            long before = small.heap.size() == 3 ? small.heap.peek().value() : -1;
            boolean kept = small.offer(x);
            System.out.print("  " + x.name() + " (" + x.value() + "): ");
            if (!kept)
                System.out.println("not more than the smallest kept (" + before
                        + "), ignored");
            else
                System.out.println("heap holds" + small.contents());
        }
        StringBuilder sb = new StringBuilder("Top 3:");
        for (Item x : small.largestFirst())
            sb.append(" ").append(x.name()).append(" (").append(x.value())
              .append(")");
        System.out.println(sb);
        System.out.println(
                "\nPart 2: the five most-rated books of the catalog (k = 5)");

        List<String> lines = Files.readAllLines(Path.of("../data/catalog.tsv"),
                                                StandardCharsets.UTF_8);
        List<long[]> ids = new ArrayList<>();       // {id, index into items}
        List<Item> items = new ArrayList<>();
        for (String line : lines.subList(1, lines.size())) {    // skip the header
            String[] f = line.split("\t");
            ids.add(new long[] {Long.parseLong(f[0]), items.size()});
            items.add(new Item(Long.parseLong(f[2]), f[3]));
        }
        System.out.println("Read " + items.size() + " books.");
```

Kode 16.18 (Lanjutan) *P1606TopK.java: Soal 16.6 dalam Java.*

```
        TopK inFileOrder = new TopK(5);            // the file: most-rated first
        for (Item x : items) inFileOrder.offer(x);
        ids.sort(Comparator.comparingLong(a -> a[0]));   // now in identifier order
        TopK top = new TopK(5);
        for (long[] a : ids) top.offer(items.get((int) a[1]));
        List<Item> rising = new ArrayList<>(items);      // the worst order
        rising.sort(ORDER);
        TopK worst = new TopK(5);
        for (Item x : rising) worst.offer(x);
        System.out.println("Changes to the heap after it filled up:\n  "
                + inFileOrder.replaced + " in file order (most-rated first), "
                + top.replaced + " in identifier order,\n  " + worst.replaced
                + " in increasing order of ratings.");

        List<Item> best = top.largestFirst();
        for (int i = 0; i < best.size(); i++)
            System.out.println("  " + (i + 1) + ". " + best.get(i).value() + "  "
                    + best.get(i).name());
        List<Item> all = new ArrayList<>(rising);  // the slow way: sort everything
        all.sort(ORDER.reversed());
        boolean same = best.equals(all.subList(0, 5));
        System.out.println("Same five as sorting all " + all.size() + " books: "
                + (same ? "yes" : "no"));
    }
}
```

Kode 16.19 `p1606_top_k.py`: Soal 16.6 dalam Python.

```python
# p1606_top_k.py - Problem 16.6: the k largest values with a min-heap of size k.
import heapq


class TopK:
    """A min-heap of at most k items: heap[0] is the SMALLEST of those kept."""

    def __init__(self, k):
        self.k, self.heap, self.replaced = k, [], 0

    def offer(self, x):                     # True if the item was kept
        if len(self.heap) < self.k:         # not full yet: always keep
            heapq.heappush(self.heap, x)
            return True
        if x <= self.heap[0]:               # not above the smallest kept
            return False
        heapq.heapreplace(self.heap, x)     # drop the smallest, add x
        self.replaced += 1
        return True

    def largest_first(self):
        return sorted(self.heap, reverse=True)

    def contents(self):                     # smallest first, for tracing
        return "".join(f" {v}" for v, _ in sorted(self.heap))


def main():
    print("Part 1: the three most-borrowed books this month (k = 3)")
    counts = [12, 45, 7, 30, 18, 3, 26, 40]
    small = TopK(3)
    for i, c in enumerate(counts, 1):
        x = (c, f"B100{i}")
        before = small.heap[0][0] if len(small.heap) == 3 else -1
        kept = small.offer(x)
        print(f"  {x[1]} ({c}): ", end="")
        if not kept:
            print(f"not more than the smallest kept ({before}), ignored")
        else:
            print("heap holds" + small.contents())
    print("Top 3:" + "".join(f" {name} ({v})" for v, name in small.largest_first()))
    print("\nPart 2: the five most-rated books of the catalog (k = 5)")
```

Kode 16.19 (Lanjutan) *p1606_top_k.py: Soal 16.6 dalam Python.*

```
    books = []                              # (id, (ratings, title))
    with open("../data/catalog.tsv", encoding="utf-8") as f:
        next(f)                             # skip the header
        for line in f:
            fields = line.rstrip("\n").split("\t")
            books.append((int(fields[0]), (int(fields[2]), fields[3])))
    print(f"Read {len(books)} books.")

    in_file_order = TopK(5)                 # the file: most-rated first
    for _, x in books:
        in_file_order.offer(x)
    books.sort()                            # now in order of identifier
    top = TopK(5)
    for _, x in books:
        top.offer(x)
    rising = sorted(x for _, x in books)    # the worst order: fewest first
    worst = TopK(5)
    for x in rising:
        worst.offer(x)
    print(f"Changes to the heap after it filled up:\n  {in_file_order.replaced} in "
          f"file order (most-rated first), {top.replaced} in identifier order,\n  "
          f"{worst.replaced} in increasing order of ratings.")

    best = top.largest_first()
    for i, (v, title) in enumerate(best, 1):
        print(f"  {i}. {v}  {title}")
    everything = sorted(rising, reverse=True)   # the slow way: sort everything
    same = best == everything[:5]
    print(f"Same five as sorting all {len(everything)} books:",
          "yes" if same else "no")


if __name__ == "__main__":
    main()
```

Penyelesaian keliru yang menggoda. "Nilai-nilai terbesar, jadi *max-heap*": di C++ itulah yang diberikan `std::priority_queue<int>` secara bawaan, dan kita tergoda untuk memasukkan semuanya lalu mengeluarkan satu setiap kali ukurannya melebihi k:

```
  priority_queue<int> heap;            // WRONG: a MAX-heap; top() is the largest
  heap.push(c); if (heap.size() > 3) heap.pop();
```

`wrong/p1606_wrong.cpp` mencetak

```
Top 3: 12 7 3
```

tiga hitungan *terkecil*. Setiap `pop` membuang nilai terbesar sejauh ini, yang justru ingin kita simpan. Kode yang sama dengan `PriorityQueue` Java atau `heapq` Python kebetulan benar, karena keduanya *min-heap*; inilah baris berbahaya pertama dari **Tabel 16.3** yang sedang bekerja. Aturan yang perlu diingat adalah yang tertulis di invarian: untuk menyimpan k yang *terbesar*, Anda harus dapat membuang yang *terkecil* di antara mereka dengan cepat, sehingga akar *heap* harus yang terkecil.

> **UJI DIRI.** Dengan $k = 3$ dan hitungan 5, 9, 2, 9, 7, 1, 8, apa isi *heap* setelah setiap nilai, dan berapa penggantian yang terjadi? Apa yang berubah bila nilai yang sama dengan akar diizinkan menggantikannya?
>
> *Jawaban.* 5; 5, 9; 2, 5, 9; lalu 9 mengalahkan 2: 5, 9, 9; 7 mengalahkan 5: 7, 9, 9; 1 diabaikan; 8 mengalahkan 7: 8, 9, 9. Tiga penggantian; tiga teratasnya 9, 9, 8. Bila nilai yang sama diizinkan menggantikan akar, tidak ada yang berubah di sini, karena tidak ada nilai yang sama dengan akar pada saat nilai itu tiba; dengan $k = 1$ dan hitungan 9, 9, angka 9 yang kedua akan

menyebabkan penggantian yang sia-sia. Menolak nilai yang sama dengan akar menghemat pekerjaan dan, untuk rekaman yang bernama, mempertahankan yang tiba lebih dahulu.

16.6.2 Soal 16.7 — Median sejauh ini, dengan dua heap

SOAL. Pengunjung meja baca Pustaka Ceria menunggu 5, 15, 1, 3, 8, 7, 9, 10, 20, dan 2 menit, dalam urutan itu. Setelah setiap pengunjung, cetak median waktu tunggu sejauh ini (untuk banyaknya yang genap, rata-rata dari dua nilai tengah). Setiap pembaruan harus berongkos $O(\lg n)$, bukan pengurutan.

Dengan kata-kata kita sendiri. Median adalah nilai di tengah daftar yang terurut. Mengurutkan setelah setiap pengunjung berongkos $O(n \lg n)$ setiap kali. Namun kita tidak pernah memerlukan seluruh daftar terurut, hanya bagian tengahnya, sehingga nilai-nilai itu kita bagi menjadi separuh bawah dan separuh atas. Nilai-nilai tengahnya adalah *yang terbesar dari separuh bawah* dan *yang terkecil dari separuh atas*: simpan separuh bawah di *max-heap* `low` dan separuh atas di *min-heap* `high`, maka kedua nilai tengah berada di akar.

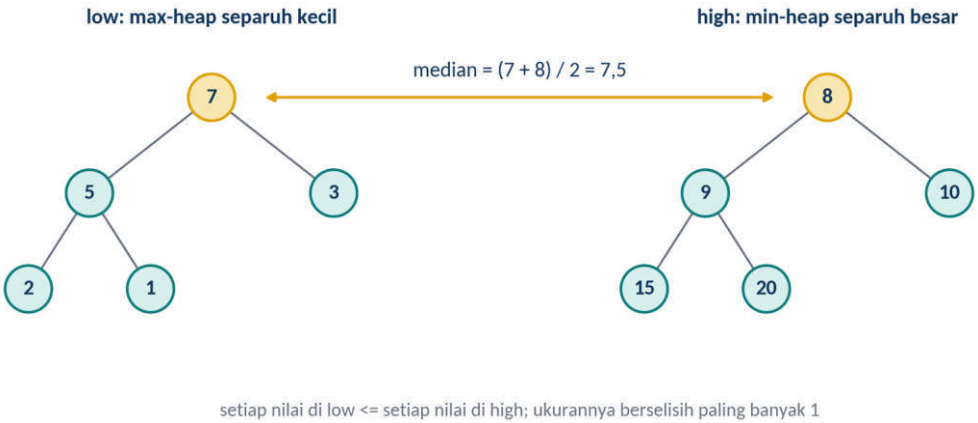
DEFINISI. *Invarian dua heap*: (i) setiap nilai di `low` paling besar sama dengan setiap nilai di `high`, yang berlaku bila `low.top() <= high.top()`; (ii) `low.size()` sama dengan `high.size()` atau `high.size() + 1`. Maka median adalah `low.top()` bila banyaknya ganjil, dan `(low.top() + high.top()) / 2` bila genap.

Nilai baru masuk ke `low` bila paling besar sama dengan `low.top()`, dan ke `high` bila tidak; itu menjaga bagian (i). Hal itu dapat merusak bagian (ii), dan satu pemindahan memperbaikinya: bila `low` kelebihan dua, pindahkan puncaknya ke `high`; bila `high` lebih besar, pindahkan puncaknya ke `low`. Memindahkan *puncak* menjaga bagian (i), karena puncak `low` adalah nilai terbesar separuh bawah dan menjadi yang terkecil di separuh atas, dan sebaliknya.

Dengan tangan, selangkah demi selangkah. Program mencetak kedua *heap* (masing-masing didaftar menurut urutan nilainya akan keluar) setelah setiap pengunjung:

```
add  5  median  5.0   low {5}  high {}
add 15  median 10.0   low {5}  high {15}
add  1  median  5.0   low {5 1}  high {15}
add  3  median  4.0   low {3 1}  high {5 15}
add  8  median  5.0   low {5 3 1}  high {8 15}
add  7  median  6.0   low {5 3 1}  high {7 8 15}
add  9  median  7.0   low {7 5 3 1}  high {8 9 15}
add 10  median  7.5   low {7 5 3 1}  high {8 9 10 15}
add 20  median  8.0   low {8 7 5 3 1}  high {9 10 15 20}
add  2  median  7.5   low {7 5 3 2 1}  high {8 9 10 15 20}
Two heaps agree with sorting every time: yes
```

Perhatikan baris keempat. Setelah 5, 15, 1 kita memiliki `low` = {5, 1} dan `high` = {15}. Nilai 3 paling besar 5, sehingga masuk ke `low`, yang kini berisi tiga nilai lawan satu: terlalu banyak. Puncaknya, 5, pindah ke `high`: `low` = {3, 1}, `high` = {5, 15}, dan mediannya $(3 + 5)/2 = 4$. Periksa: daftar terurut 1, 3, 5, 15 memiliki nilai tengah 3 dan 5. Setelah kesepuluh pengunjung, **Gambar 16.12** memperlihatkan kedua *heap*; mediannya $(7 + 8)/2 = 7.5$, yang merupakan tengah dari daftar terurut 1, 2, 3, 5, 7, 8, 9, 10, 15, 20. Baris terakhir keluaran memastikan bahwa kedua *heap* sepakat dengan pengurutan penuh setelah setiap pengunjung.



Gambar 16.12 Kedua *heap* setelah kesepuluh waktu tunggu. Akar-akarnya adalah dua nilai tengah daftar terurut, sehingga median dibaca dalam waktu konstan.

Ongkos. Setiap nilai baru berongkos satu *push* dan paling banyak satu pemindahan antar-*heap*, yaitu satu *pop* dan satu *push*: $O(\lg n)$. Membaca median berongkos $O(1)$. Untuk n nilai, seluruh aliran berongkos $O(n \lg n)$, dibandingkan $O(n^2 \lg n)$ bila mengurutkan setelah setiap nilai.

Penyelesaian C++. Kode 16.20 menyimpan `low` sebagai `priority_queue<int>` bawaan, yang merupakan *max-heap*, dan `high` dengan `greater<int>`. Di sini bawaan C++ tepat seperti yang kita inginkan untuk separuh bawah.

Kode 16.20 `p1607_running_median.cpp`: median berjalan dengan *max-heap* untuk separuh bawah dan *min-heap* untuk separuh atas, diperiksa terhadap pengurutan setelah setiap nilai.

```cpp
// p1607_running_median.cpp - Problem 16.7: the median so far, with two heaps.
#include <algorithm>
#include <functional>
#include <iomanip>
#include <iostream>
#include <queue>
#include <sstream>
#include <string>
#include <vector>
using namespace std;

class RunningMedian {
    priority_queue<int> low;                                  // max-heap: smaller half
    priority_queue<int, vector<int>, greater<int>> high;  // min-heap: larger half
    // Invariant: every value in low <= every value in high, and
    //            low.size() == high.size() or low.size() == high.size() + 1.
public:
    void add(int x) {
        if (low.empty() || x <= low.top()) low.push(x);
        else high.push(x);
        if (low.size() > high.size() + 1) {                // low too big: move top
            high.push(low.top());
            low.pop();
        } else if (high.size() > low.size()) {             // high too big: move top
            low.push(high.top());
            high.pop();
        }
    }
    double median() const {
        if (low.size() > high.size()) return low.top();
        return (low.top() + high.top()) / 2.0;
    }
    template <class PQ>
    static string list(PQ pq) {                            // a copy, emptied
        string s;
        while (!pq.empty()) {
            s += (s.empty() ? "" : " ") + to_string(pq.top());
            pq.pop();
        }
        return "{" + s + "}";
    }
    string show() const { return "low " + list(low) + "  high " + list(high); }
};
```

Kode 16.20 (Lanjutan) *p1607_running_median.cpp: median berjalan dengan max-heap untuk separuh bawah dan min-heap untuk separuh atas, diperiksa terhadap pengurutan setelah setiap nilai.*

```
double bruteForceMedian(vector<int> v) {                // sort everything: slow
    sort(v.begin(), v.end());
    size_t n = v.size();
    return n % 2 ? v[n / 2] : (v[n / 2 - 1] + v[n / 2]) / 2.0;
}

int main() {
    vector<int> waits = {5, 15, 1, 3, 8, 7, 9, 10, 20, 2};
    RunningMedian rm;
    vector<int> seen;
    bool agree = true;
    cout << fixed << setprecision(1);
    for (int x : waits) {
        rm.add(x);
        seen.push_back(x);
        agree = agree && rm.median() == bruteForceMedian(seen);
        cout << "add " << setw(2) << x << "  median " << setw(4) << rm.median()
             << "    " << rm.show() << "\n";
    }
    cout << "Two heaps agree with sorting every time: " << (agree ? "yes" : "no")
         << "\n";
    return 0;
}
```

Baris demi baris. `add` mengerjakan ketiga langkahnya: menempatkan, lalu paling banyak satu dari dua perbaikan. `median` memakai `2.0` agar rata-rata 7 dan 8 menjadi 7.5, bukan bilangan bulat 7. `list` adalah *template* yang menerima salinan salah satu *heap* (parameter `PQ pq` diberikan menurut nilai) dan mengosongkan salinan itu untuk mencetaknya berurutan; *heap* yang asli tidak tersentuh. Perbandingan dengan `bruteForceMedian`, yang mengurutkan salinan semua nilai yang telah dilihat, adalah Langkah 9 metode kita: kembaran lambat yang jelas benar.

Penyelesaian Java dan Python. **Kode 16.21** dan **Kode 16.22** mencetak baris-baris yang sama. Java membangun *max-heap* dengan `Collections.reverseOrder()`. `heapq` di Python sepanjang sebagian besar umurnya hanya berupa *min-heap*, sehingga `low` menyimpan nilai-nilai yang *dinegasikan*: bilangan negatif terkecil adalah nilai terbesar. (Python 3.14 menambahkan `heapq.heappush_max` dan kerabatnya, dan Bab 14 memperlihatkannya; trik negasi berjalan di semua versi.)

Kode 16.21 `P1607RunningMedian.java`: Soal 16.7 dalam Java.

```java
// P1607RunningMedian.java - Problem 16.7: the median so far, with two heaps.
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.PriorityQueue;

public class P1607RunningMedian {

    static class RunningMedian {
        // max-heap for the smaller half, min-heap for the larger half.
        // Invariant: every value in low <= every value in high, and
        //            low.size() == high.size() or low.size() == high.size() + 1.
        final PriorityQueue<Integer> low =
                new PriorityQueue<>(Collections.reverseOrder());
        final PriorityQueue<Integer> high = new PriorityQueue<>();

        void add(int x) {
            if (low.isEmpty() || x <= low.peek()) low.add(x);
            else high.add(x);
            if (low.size() > high.size() + 1) high.add(low.poll());   // low too big
            else if (high.size() > low.size()) low.add(high.poll()); // high too big
        }

        double median() {
            if (low.size() > high.size()) return low.peek();
            return (low.peek() + high.peek()) / 2.0;
        }

        static String list(PriorityQueue<Integer> pq) {   // a copy, emptied
            PriorityQueue<Integer> copy = new PriorityQueue<>(pq.comparator());
            copy.addAll(pq);
            List<String> s = new ArrayList<>();
            while (!copy.isEmpty()) s.add(String.valueOf(copy.poll()));
            return "{" + String.join(" ", s) + "}";
        }

        String show() {
            return "low " + list(low) + "  high " + list(high);
        }
    }
```

Kode 16.21 (Lanjutan) *P1607RunningMedian.java: Soal 16.7 dalam Java.*

```
    static double bruteForceMedian(List<Integer> seen) {  // sort everything: slow
        List<Integer> v = new ArrayList<>(seen);
        Collections.sort(v);
        int n = v.size();
        return n % 2 == 1 ? v.get(n / 2) : (v.get(n / 2 - 1) + v.get(n / 2)) / 2.0;
    }

    public static void main(String[] args) {
        int[] waits = {5, 15, 1, 3, 8, 7, 9, 10, 20, 2};
        RunningMedian rm = new RunningMedian();
        List<Integer> seen = new ArrayList<>();
        boolean agree = true;
        for (int x : waits) {
            rm.add(x);
            seen.add(x);
            agree = agree && rm.median() == bruteForceMedian(seen);
            System.out.printf("add %2d  median %4.1f   %s%n", x, rm.median(),
                    rm.show());
        }
        System.out.println("Two heaps agree with sorting every time: "
                + (agree ? "yes" : "no"));
    }
}
```

Kode 16.22 `p1607_running_median.py`: Soal 16.7 dalam Python, dengan separuh bawah disimpan dalam bentuk negasi.

```python
# p1607_running_median.py - Problem 16.7: the median so far, with two heaps.
import heapq


class RunningMedian:
    """low holds the smaller half as a max-heap (values stored negated, because
    heapq is a min-heap); high holds the larger half as a min-heap.
    Invariant: every value in low <= every value in high, and
    len(low) == len(high) or len(low) == len(high) + 1."""

    def __init__(self):
        self.low, self.high = [], []

    def add(self, x):
        if not self.low or x <= -self.low[0]:
            heapq.heappush(self.low, -x)
        else:
            heapq.heappush(self.high, x)
        if len(self.low) > len(self.high) + 1:        # low too big: move its top
            heapq.heappush(self.high, -heapq.heappop(self.low))
        elif len(self.high) > len(self.low):          # high too big: move its top
            heapq.heappush(self.low, -heapq.heappop(self.high))

    def median(self):
        if len(self.low) > len(self.high):
            return float(-self.low[0])
        return (-self.low[0] + self.high[0]) / 2

    def show(self):
        low = " ".join(str(v) for v in sorted((-x for x in self.low), reverse=True))
        high = " ".join(str(v) for v in sorted(self.high))
        return f"low {{{low}}}  high {{{high}}}"


def brute_force_median(seen):                         # sort everything: slow
    v = sorted(seen)
    n = len(v)
    return float(v[n // 2]) if n % 2 else (v[n // 2 - 1] + v[n // 2]) / 2
```

Kode 16.22 (Lanjutan) *p1607_running_median.py: Soal 16.7 dalam Python, dengan separuh bawah disimpan dalam bentuk negasi.*

```python
def main():
    waits = [5, 15, 1, 3, 8, 7, 9, 10, 20, 2]
    rm, seen, agree = RunningMedian(), [], True
    for x in waits:
        rm.add(x)
        seen.append(x)
        agree = agree and rm.median() == brute_force_median(seen)
        print(f"add {x:2d}  median {rm.median():4.1f}   {rm.show()}")
    print("Two heaps agree with sorting every time:", "yes" if agree else "no")


if __name__ == "__main__":
    main()
```

Penyelesaian keliru yang menggoda. Menempatkan setiap nilai di sisi yang benar terasa sudah cukup; penyeimbangan tampak seperti hiasan. Tanpa penyeimbangan (`wrong/p1607_wrong.cpp`):

```
add 5  median 5
add 15  median 10
add 1  median 5
add 3  median 10
add 8  median 5
add 7  median 6
add 9  median 5
add 10  median 6
add 20  median 5
add 2  median 6
```

Tiga median pertama benar, lalu yang keempat 10 alih-alih 4, dan sejak itu jawabannya lebih sering salah daripada benar. Setelah 5, 15, 1, 3, kedua separuh adalah {5, 3, 1} dan {15}: bagian (i) invarian berlaku, tetapi bagian (ii) tidak, sehingga akar-akarnya tidak lagi merupakan nilai-nilai tengah. Median adalah pernyataan tentang *posisi*, dan hanya syarat ukuranlah yang mengikat akar-akar itu pada posisi tengah.

> **UJI DIRI.** Lanjutkan aliran itu dengan 6 lalu 30. Berikan kedua *heap* dan mediannya setelah masing-masing.
>
> *Jawaban.* 6 paling besar sama dengan `low.top()` = 7, sehingga masuk ke `low`, yang kini berisi enam nilai lawan lima; ukuran itu diperbolehkan (low boleh lebih satu), jadi tidak ada pemindahan: `low` = {7, 6, 5, 3, 2, 1}, `high` = {8, 9, 10, 15, 20}, median 7 (sebelas nilai; daftar terurut 1, 2, 3, 5, 6, 7, 8, 9, 10, 15, 20 memiliki 7 di tengahnya). Kemudian 30 > 7 masuk ke `high`, ukuran 6 dan 6: median $(7 + 8)/2 = 7.5$.

16.7 Soal Baru yang Dikerjakan: Hashing

Soal 16.2 memakai tabel hash dari pustaka. Kedua soal dalam subbab ini membangun tabel hash dengan tangan, satu untuk masing-masing strategi tabrakan di Bab 15, dan masing-masing membidik kekeliruan yang disembunyikan pustaka dari kita: indeks yang dihitung dengan bilangan bulat yang meluap (*overflow*), dan penghapusan yang merusak pencarian kunci yang masih ada.

16.7.1 Soal 16.8 — Tabel hash dengan separate chaining, dengan tangan

> **SOAL.** Simpan delapan buku Pustaka Ceria, dengan judul sebagai kunci dan kode buku sebagai nilai, di dalam tabel hash dengan *separate chaining* yang dimulai dengan 5 *bucket* dan tumbuh menjadi $2m + 1$ *bucket* setiap kali banyaknya kunci melebihi banyaknya *bucket*. Pakai hash polinomial $h = (h \cdot 31 + c) \bmod m$ atas karakter-karakter c dari judul. Cetak tabel sebelum dan sesudah tumbuh; cari dua judul yang ada dan satu yang tidak ada; hapus satu judul dua kali.

Dengan kata-kata kita sendiri. Sebuah larik berisi m *bucket*; *bucket* i adalah kepala sebuah *singly linked list* (Bab 7) berisi entri-entri yang kuncinya ter-hash ke i. Untuk mencari sebuah kunci, hitung *bucket*-nya dan telusuri satu rantai itu. Untuk menyisipkan, telusuri rantai untuk memastikan kuncinya baru, lalu taruh sebuah simpul di depan. Untuk menghapus, temukan simpulnya lalu lepaskan tautannya. *Load factor* (faktor muat) adalah

$$\alpha = \frac{n}{m}, \tag{16.9}$$

yaitu panjang rantai rata-rata. Dengan fungsi hash yang menyebarkan kunci secara merata, pencarian yang gagal menelusuri satu rantai penuh dan pencarian yang berhasil menelusuri sekitar separuhnya, sehingga keduanya berongkos [9]

$$T_{\text{rantai}} = \Theta(1 + \alpha). \tag{16.10}$$

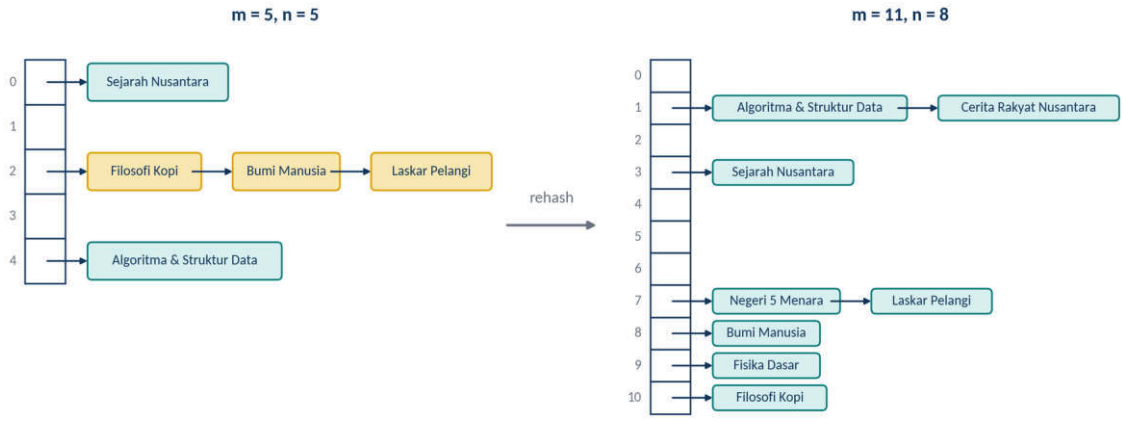
Menurut Persamaan (16.9) dan Persamaan (16.10), menjaga $\alpha \leq 1$, seperti yang dilakukan aturan kita, membuat setiap operasi rata-rata $O(1)$; harganya adalah *rehash* sesekali, yang memindahkan setiap simpul ke larik yang lebih besar. Karena lariknya kira-kira berlipat dua setiap kali, total ongkos semua *rehash* selama n penyisipan adalah $O(n)$, sehingga setiap penyisipan membayar $O(1)$ teramortisasi (Bab 15).

> **DEFINISI.** *Invarian chaining*: setiap kunci k yang tersimpan berada di rantai *bucket* $h(k)$, dan muncul di sana tepat sekali.

Dengan tangan, selangkah demi selangkah. Hash sebuah judul dihitung satu karakter demi satu karakter, dan direduksi modulo m setelah setiap langkah agar nilai berjalannya tetap kecil. Reduksi itu tidak mengubah hasil, karena $(a \bmod m) \cdot 31 + c \equiv a \cdot 31 + c \pmod m$. Untuk "Fisika Dasar" dan $m = 11$, mulai dari $h = 0$: `F` (kode 70) memberi $70 \bmod 11 = 4$; `i` (105) memberi $(4 \cdot 31 + 105) \bmod 11 = 229 \bmod 11 = 9$; `s` (115) memberi $394 \bmod 11 = 9$; dan seterusnya melalui spasi (32) serta `D`, `a`, `s`, `a`, `r`, berakhir pada 9. Jadi "Fisika Dasar" tinggal di *bucket* 9, sesuai yang dicetak program. Program memperlihatkan setiap *bucket* dan setiap rantai:

```
put "Laskar Pelangi" -> bucket 2
put "Bumi Manusia" -> bucket 2
put "Filosofi Kopi" -> bucket 2
put "Sejarah Nusantara" -> bucket 0
put "Algoritma & Struktur Data" -> bucket 4
m = 5, n = 5, load factor 1.00
  [ 0] Sejarah Nusantara
  [ 1] -
  [ 2] Filosofi Kopi -> Bumi Manusia -> Laskar Pelangi
  [ 3] -
  [ 4] Algoritma & Struktur Data
put "Cerita Rakyat Nusantara" -> bucket 0
  (load factor 6/5 > 1: rehash to 11 buckets)
put "Fisika Dasar" -> bucket 9
put "Negeri 5 Menara" -> bucket 7
m = 11, n = 8, load factor 0.73
  [ 0] -
  [ 1] Algoritma & Struktur Data -> Cerita Rakyat Nusantara
  [ 2] -
  [ 3] Sejarah Nusantara
  [ 4] -
  [ 5] -
  [ 6] -
  [ 7] Negeri 5 Menara -> Laskar Pelangi
  [ 8] Bumi Manusia
  [ 9] Fisika Dasar
  [10] Filosofi Kopi
get "Fisika Dasar" -> B1007
get "Laskar Pelangi" -> B1001
get "Harry Potter" -> not found
remove "Bumi Manusia" -> true
remove "Bumi Manusia" -> false
m = 11, n = 7, load factor 0.64
  [ 0] -
  [ 1] Algoritma & Struktur Data -> Cerita Rakyat Nusantara
  [ 2] -
  [ 3] Sejarah Nusantara
  [ 4] -
  [ 5] -
  [ 6] -
  [ 7] Negeri 5 Menara -> Laskar Pelangi
  [ 8] -
  [ 9] Fisika Dasar
  [10] Filosofi Kopi
```

Tabel pertama adalah yang menarik. Dengan hanya 5 *bucket*, tiga dari lima judul pertama mendarat di *bucket* 2 dan membentuk rantai tiga simpul; yang terbaru, “Filosofi Kopi”, berada di depan karena penyisipan dilakukan di depan. Judul keenam menaikkan n menjadi $6 > 5$, dan tabel tumbuh menjadi 11 *bucket*. Setiap simpul *ditautkan ulang*, bukan disalin, ke *bucket* tempat kuncinya ter-hash modulo 11. Setelah judul kedelapan, $\alpha = 8/11 \approx 0.73$, rantai terpanjang berisi dua simpul, dan enam *bucket* kosong. Itu wajar: melempar 8 kunci secara acak ke 11 *bucket* menyisakan sekitar $11(1 - 1/11)^8 \approx 5$ yang kosong. **Gambar 16.13** menggambar tabel sebelum dan sesudah *rehash*.



Gambar 16.13 Tabel berantai dengan 5 *bucket* (tiga judul bertabrakan di *bucket* 2) dan setelah *rehash* menjadi 11 *bucket*, tempat tidak ada rantai yang lebih panjang daripada dua.

Penyelesaian C++. Kode 16.23 membangun simpul-simpulnya dengan tangan memakai `new` dan membebaskannya di destruktor, seperti di Bab 7; menyalin tabel dilarang (`= delete`) karena salinannya akan berbagi simpul dan membebaskannya dua kali.

Kode 16.23 `p1608_chaining.cpp`: tabel hash dengan *separate chaining*, penyisipan di depan, penghapusan melalui pointer ke tautan, dan *rehash* ketika *load factor* melebihi 1.

```
// p1608_chaining.cpp - Problem 16.8: a hash table with separate chaining, by hand.
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class ChainedTable {
    struct Node {
        string key, value;
        Node* next;
    };
    vector<Node*> bucket;          // bucket[i]: first node of chain i, or nullptr
    size_t n = 0;                  // number of keys stored

    // Polynomial hash, reduced modulo m at every step so it never overflows.
    static size_t hashOf(const string& key, size_t m) {
        size_t h = 0;
        for (unsigned char c : key) h = (h * 31 + c) % m;
        return h;
    }

    void rehash(size_t newM) {     // relink every node into a bigger table
        vector<Node*> old = bucket;
        bucket.assign(newM, nullptr);
        for (Node* head : old) {
            while (head != nullptr) {
                Node* next = head->next;                  // save the way on first
                size_t i = hashOf(head->key, newM);
                head->next = bucket[i];                   // front insertion
                bucket[i] = head;
                head = next;
            }
        }
    }
```

Kode 16.23 (Lanjutan) *p1608_chaining.cpp: tabel hash dengan separate chaining, penyisipan di depan, penghapusan melalui pointer ke tautan, dan rehash ketika load factor melebihi 1.*

```
public:
    explicit ChainedTable(size_t m) : bucket(m, nullptr) {}
    ~ChainedTable() {
        for (Node* head : bucket)
            while (head != nullptr) {
                Node* next = head->next;
                delete head;
                head = next;
            }
    }
    ChainedTable(const ChainedTable&) = delete;
    ChainedTable& operator=(const ChainedTable&) = delete;

    size_t indexOf(const string& key) const { return hashOf(key, bucket.size()); }

    void put(const string& key, const string& value) {
        size_t i = indexOf(key);
        for (Node* p = bucket[i]; p != nullptr; p = p->next)
            if (p->key == key) {                          // already there: update
                p->value = value;
                return;
            }
        bucket[i] = new Node{key, value, bucket[i]};      // front of chain i
        n++;
        if (n > bucket.size()) {                          // load factor above 1
            cout << "  (load factor " << n << "/" << bucket.size()
                 << " > 1: rehash to " << 2 * bucket.size() + 1 << " buckets)\n";
            rehash(2 * bucket.size() + 1);
        }
    }

    const string* get(const string& key) const {          // nullptr if absent
        for (Node* p = bucket[indexOf(key)]; p != nullptr; p = p->next)
            if (p->key == key) return &p->value;
        return nullptr;
    }
```

Kode 16.23 (Lanjutan) *p1608_chaining.cpp: tabel hash dengan separate chaining, penyisipan di depan, penghapusan melalui pointer ke tautan, dan rehash ketika load factor melebihi 1.*

```
    bool remove(const string& key) {
        Node** link = &bucket[indexOf(key)];            // the pointer to change
        while (*link != nullptr && (*link)->key != key) link = &(*link)->next;
        if (*link == nullptr) return false;
        Node* dead = *link;
        *link = dead->next;                             // unlink, THEN free
        delete dead;
        n--;
        return true;
    }

    void print() const {
        cout << "m = " << bucket.size() << ", n = " << n << ", load factor "
             << fixed << setprecision(2) << (double)n / bucket.size() << "\n";
        for (size_t i = 0; i < bucket.size(); i++) {
            cout << "  [" << setw(2) << i << "]";
            if (bucket[i] == nullptr) cout << " -";
            for (Node* p = bucket[i]; p != nullptr; p = p->next)
                cout << (p != bucket[i] ? " -> " : " ") << p->key;
            cout << "\n";
        }
    }
};

int main() {
    const char* books[][2] = {
        {"Laskar Pelangi", "B1001"}, {"Bumi Manusia", "B1002"},
        {"Filosofi Kopi", "B1003"}, {"Sejarah Nusantara", "B1004"},
        {"Algoritma & Struktur Data", "B1005"},
        {"Cerita Rakyat Nusantara", "B1006"},
        {"Fisika Dasar", "B1007"}, {"Negeri 5 Menara", "B1008"}};
    ChainedTable t(5);
    for (int i = 0; i < 8; i++) {
        cout << "put \"" << books[i][0] << "\" -> bucket " << t.indexOf(books[i][0])
             << "\n";
        t.put(books[i][0], books[i][1]);
        if (i == 4) t.print();
    }
    t.print();
    for (const char* q : {"Fisika Dasar", "Laskar Pelangi", "Harry Potter"}) {
        const string* v = t.get(q);
        cout << "get \"" << q << "\" -> " << (v ? *v : "not found") << "\n";
    }
    cout << "remove \"Bumi Manusia\" -> " << boolalpha << t.remove("Bumi Manusia")
         << "\nremove \"Bumi Manusia\" -> " << t.remove("Bumi Manusia") << "\n";
    t.print();
```

Kode 16.23 (Lanjutan) *p1608_chaining.cpp: tabel hash dengan separate chaining, penyisipan di depan, penghapusan melalui pointer ke tautan, dan rehash ketika load factor melebihi 1.*

```
    return 0;
}
```

Baris demi baris. `hashOf` memakai `unsigned char` agar bita di atas 127 (huruf beraksen dalam UTF-8) terhitung sebagai bilangan positif, dan aritmetika `size_t` yang direduksi di setiap langkah, sehingga tidak ada nilai antara yang melebihi $31 \cdot m + 255$. `rehash` menelusuri setiap rantai lama, menyimpan `head->next` sebelum menautkan ulang simpul ke larik baru, aturan dari Bab 8. `remove` memakai *pointer ke tautan*, `Node** link`, yang mula-mula menunjuk pointer kepala *bucket* lalu medan `next` setiap simpul bergiliran; ketika kuncinya ditemukan, `*link = dead->next` melepaskan simpul itu, baik ia pertama di rantainya maupun tidak, tanpa kasus khusus,

dan baru setelah itu simpulnya dihapus. Valgrind dan AddressSanitizer tidak melaporkan galat maupun kebocoran.

Penyelesaian Java dan Python. **Kode 16.24** dan **Kode 16.25** mencetak baris-baris yang sama. Kedua bahasa tidak menyediakan pointer ke tautan, sehingga `remove` berjalan dengan pasangan biasa `prev`, `p` dan menangani simpul pertama secara terpisah.

Kode 16.24 `P1608Chaining.java`: Soal 16.8 dalam Java.

```java
// P1608Chaining.java - Problem 16.8: a hash table with separate chaining, by hand.
public class P1608Chaining {

    static class ChainedTable {
        static class Node {
            final String key;
            String value;
            Node next;

            Node(String key, String value, Node next) {
                this.key = key;
                this.value = value;
                this.next = next;
            }
        }

        Node[] bucket;             // bucket[i]: first node of chain i, or null
        int n = 0;                 // number of keys stored

        ChainedTable(int m) {
            bucket = new Node[m];
        }

        // Polynomial hash, reduced modulo m at every step so it never overflows.
        static int hashOf(String key, int m) {
            int h = 0;
            for (int i = 0; i < key.length(); i++) h = (h * 31 + key.charAt(i)) % m;
            return h;
        }

        int indexOf(String key) {
            return hashOf(key, bucket.length);
        }

        void rehash(int newM) {    // relink every node into a bigger table
            Node[] old = bucket;
            bucket = new Node[newM];
            for (Node head : old) {
                while (head != null) {
                    Node next = head.next;              // save the way on first
                    int i = hashOf(head.key, newM);
                    head.next = bucket[i];              // front insertion
                    bucket[i] = head;
                    head = next;
                }
            }
```

Kode 16.24 (Lanjutan) *P1608Chaining.java: Soal 16.8 dalam Java.*

```
    }

    void put(String key, String value) {
        int i = indexOf(key);
        for (Node p = bucket[i]; p != null; p = p.next)
            if (p.key.equals(key)) {                // already there: update
                p.value = value;
                return;
            }
        bucket[i] = new Node(key, value, bucket[i]); // front of chain i
        n++;
        if (n > bucket.length) {                    // load factor above 1
            System.out.println("  (load factor " + n + "/" + bucket.length
                    + " > 1: rehash to " + (2 * bucket.length + 1)
                    + " buckets)");
            rehash(2 * bucket.length + 1);
        }
    }

    String get(String key) {                        // null if absent
        for (Node p = bucket[indexOf(key)]; p != null; p = p.next)
            if (p.key.equals(key)) return p.value;
        return null;
    }

    boolean remove(String key) {
        int i = indexOf(key);
        Node prev = null, p = bucket[i];
        while (p != null && !p.key.equals(key)) {
            prev = p;
            p = p.next;
        }
        if (p == null) return false;
        if (prev == null) bucket[i] = p.next;       // unlink the first node
        else prev.next = p.next;                    // unlink a later node
        n--;
        return true;
    }
```

Kode 16.24 (Lanjutan) *P1608Chaining.java: Soal 16.8 dalam Java.*

```
        void print() {
            System.out.printf("m = %d, n = %d, load factor %.2f%n", bucket.length,
                    n, (double) n / bucket.length);
            for (int i = 0; i < bucket.length; i++) {
                StringBuilder sb = new StringBuilder(String.format("  [%2d]", i));
                if (bucket[i] == null) sb.append(" -");
                for (Node p = bucket[i]; p != null; p = p.next)
                    sb.append(p != bucket[i] ? " -> " : " ").append(p.key);
                System.out.println(sb);
            }
        }
    }

    public static void main(String[] args) {
        String[][] books = {
            {"Laskar Pelangi", "B1001"}, {"Bumi Manusia", "B1002"},
            {"Filosofi Kopi", "B1003"}, {"Sejarah Nusantara", "B1004"},
            {"Algoritma & Struktur Data", "B1005"},
            {"Cerita Rakyat Nusantara", "B1006"},
            {"Fisika Dasar", "B1007"}, {"Negeri 5 Menara", "B1008"}};
        ChainedTable t = new ChainedTable(5);
        for (int i = 0; i < 8; i++) {
            System.out.println("put \"" + books[i][0] + "\" -> bucket "
                    + t.indexOf(books[i][0]));
            t.put(books[i][0], books[i][1]);
            if (i == 4) t.print();
        }
        t.print();
        for (String q : new String[] {"Fisika Dasar", "Laskar Pelangi",
                                      "Harry Potter"}) {
            String v = t.get(q);
            System.out.println("get \"" + q + "\" -> "
                    + (v != null ? v : "not found"));
        }
        for (int i = 0; i < 2; i++)
            System.out.println("remove \"Bumi Manusia\" -> "
                    + t.remove("Bumi Manusia"));
        t.print();
    }
}
```

Kode 16.25 `p1608_chaining.py`: Soal 16.8 dalam Python.

```
# p1608_chaining.py - Problem 16.8: a hash table with separate chaining, by hand.


class Node:
    def __init__(self, key, value, next_node):
        self.key, self.value, self.next = key, value, next_node


class ChainedTable:
    def __init__(self, m):
        self.bucket = [None] * m    # bucket[i]: first node of chain i, or None
        self.n = 0                  # number of keys stored

    @staticmethod
    def hash_of(key, m):
        """Polynomial hash, reduced modulo m at every step (as in C++ and Java)."""
        h = 0
        for c in key:
            h = (h * 31 + ord(c)) % m
        return h

    def index_of(self, key):
        return self.hash_of(key, len(self.bucket))

    def _rehash(self, new_m):       # relink every node into a bigger table
        old, self.bucket = self.bucket, [None] * new_m
        for head in old:
            while head is not None:
                nxt = head.next                         # save the way on first
                i = self.hash_of(head.key, new_m)
                head.next = self.bucket[i]              # front insertion
                self.bucket[i] = head
                head = nxt

    def put(self, key, value):
        i = self.index_of(key)
        p = self.bucket[i]
        while p is not None:
            if p.key == key:                            # already there: update
                p.value = value
                return
            p = p.next
        self.bucket[i] = Node(key, value, self.bucket[i])   # front of chain i
        self.n += 1
        m = len(self.bucket)
        if self.n > m:                                  # load factor above 1
```

Kode 16.25 (Lanjutan) *p1608_chaining.py: Soal 16.8 dalam Python.*

```
            print(f"  (load factor {self.n}/{m} > 1: "
                  f"rehash to {2 * m + 1} buckets)")
            self._rehash(2 * m + 1)

    def get(self, key):                                  # None if absent
        p = self.bucket[self.index_of(key)]
        while p is not None:
            if p.key == key:
                return p.value
            p = p.next
        return None

    def remove(self, key):
        i = self.index_of(key)
        prev, p = None, self.bucket[i]
        while p is not None and p.key != key:
            prev, p = p, p.next
        if p is None:
            return False
        if prev is None:
            self.bucket[i] = p.next                      # unlink the first node
        else:
            prev.next = p.next                           # unlink a later node
        self.n -= 1
        return True

    def print(self):
        m = len(self.bucket)
        print(f"m = {m}, n = {self.n}, load factor {self.n / m:.2f}")
        for i, head in enumerate(self.bucket):
            keys = []
            while head is not None:
                keys.append(head.key)
                head = head.next
            print(f"  [{i:2d}] " + (" -> ".join(keys) if keys else "-"))
```

Kode 16.25 (Lanjutan) *p1608_chaining.py: Soal 16.8 dalam Python.*

```
def main():
    books = [("Laskar Pelangi", "B1001"), ("Bumi Manusia", "B1002"),
             ("Filosofi Kopi", "B1003"), ("Sejarah Nusantara", "B1004"),
             ("Algoritma & Struktur Data", "B1005"),
             ("Cerita Rakyat Nusantara", "B1006"),
             ("Fisika Dasar", "B1007"), ("Negeri 5 Menara", "B1008")]
    t = ChainedTable(5)
    for i, (title, code) in enumerate(books):
        print(f'put "{title}" -> bucket {t.index_of(title)}')
        t.put(title, code)
        if i == 4:
            t.print()
    t.print()
    for q in ("Fisika Dasar", "Laskar Pelangi", "Harry Potter"):
        v = t.get(q)
        print(f'get "{q}" -> {v if v is not None else "not found"}')
    print('remove "Bumi Manusia" ->', str(t.remove("Bumi Manusia")).lower())
    print('remove "Bumi Manusia" ->', str(t.remove("Bumi Manusia")).lower())
    t.print()


if __name__ == "__main__":
    main()
```

Penyelesaian keliru yang menggoda. Rumus buku teks $h = \sum c_i \cdot 31^{L-1-i}$ mengundang penyalinan langsung dengan `int` biasa, dengan reduksi modulo m hanya di akhir:

```
int h = 0;
for (char c : key) h = h * 31 + c;   // WRONG: overflows for keys of 7+ letters
return h % m;                        // ...and a negative h gives a negative index
```

`wrong/p1608_wrong.cpp` menghitung *bucket* untuk "Laskar Pelangi" dengan cara ini lalu menyimpan judulnya di sana. Dikompilasi biasa, program ini mencetak

```
Laskar Pelangi -> bucket -1
Segmentation fault
```

dan, dikompilasi dengan *sanitizer*, program ini melaporkan kedua galatnya:

```
p1608_wrong.cpp:9:30: runtime error: signed integer overflow:
    73191252 * 31 cannot be represented in type 'int'
Laskar Pelangi -> bucket -1
==ERROR: AddressSanitizer: heap-buffer-overflow ... READ of size 8 ...
    #1 ... in main ... p1608_wrong.cpp:18
```

31^6 sudah sekitar 8.9×10^8, sehingga setelah enam atau tujuh karakter $h \cdot 31$ melampaui $2^{31} - 1 \approx 2.1 \times 10^9$. Di C++, *overflow* bilangan bulat *signed* adalah perilaku tak terdefinisi; dalam praktik nilainya berputar menjadi bilangan negatif, dan sisa pembagian bilangan negatif bernilai negatif di C++ dan Java (Bab 8), sehingga indeksnya -1 dan `bucket[-1]` terletak sebelum larik. Perbaikannya adalah dua hal yang dilakukan `hashOf` kita: aritmetika *unsigned*, yang perputarannya terdefinisi, dan reduksi di setiap langkah. Di Java, kekeliruan yang sama memberikan `ArrayIndexOutOfBoundsException`; pakailah `Math.floorMod(h, m)`. Bilangan bulat Python tidak pernah meluap dan `%` tidak pernah memberi hasil negatif untuk m positif, sehingga galat ini tidak dapat terjadi di sana, tetapi nilai yang tidak direduksi tumbuh sampai ratusan digit untuk kunci yang panjang dan memperlambat setiap perhitungan hash.

> **UJI DIRI.** Pada tabel 11 *bucket*, *bucket* mana yang ditelusuri, dan berapa judul yang dibandingkan, untuk `get("Negeri 5 Menara")`, `get("Laskar Pelangi")`, dan `get("Harry Potter")`? (Hash "Harry Potter" modulo 11 adalah 4.)
>
> *Jawaban.* "Negeri 5 Menara": *bucket* 7, satu perbandingan (ia berada di depan). "Laskar Pelangi": *bucket* 7, dua perbandingan. "Harry Potter": *bucket* 4, yang kosong, sehingga nol perbandingan dan jawabannya "not found". Menurut Persamaan (16.10), ongkosnya bergantung pada rantainya, tidak pernah pada kedelapan judul secara keseluruhan.

16.7.2 Soal 16.9 — Open addressing, dan menghapus dengan aman

> **SOAL.** Sisipkan kunci 10, 22, 31, 4, 15, 28, 17, 88, 59 ke tabel berisi 11 slot dengan $h(k) = k \bmod 11$ dan *linear probing*. Kemudian hapus 15, cari 59 dan 26, lalu sisipkan 37. Perlihatkan setiap barisan *probe* dan tabel setelah setiap perubahan.

Dengan kata-kata kita sendiri. *Open addressing* menyimpan kunci-kunci di dalam larik itu sendiri, satu per slot. Bila slot $h(k)$ sudah terisi, *linear probing* mencoba $h(k)+1$, $h(k)+2$, dan seterusnya, berputar kembali di ujung larik. Pencarian mengikuti barisan yang sama dan boleh berhenti begitu mencapai slot yang *kosong*, karena penyisipan k pun akan berhenti di sana. Aturan berhenti itulah yang membuat penghapusan pelik: mengosongkan slot kunci yang dihapus dapat memotong lintasan *probe* kunci lain yang pernah terdorong melewatinya.

> **DEFINISI.** *Invarian probing*: bila kunci k tersimpan di slot j, setiap slot di lintasan *probe* dari $h(k)$ sampai j (tidak termasuk slot itu sendiri) terisi atau berupa *tombstone* (nisan), yaitu slot yang ditandai "terhapus". Karena itu pencarian melewati *tombstone* dan hanya berhenti pada

> slot yang benar-benar kosong; penyisipan boleh memakai ulang *tombstone* pertama yang dilewatinya, setelah memastikan kuncinya tidak ada lebih jauh di depan.

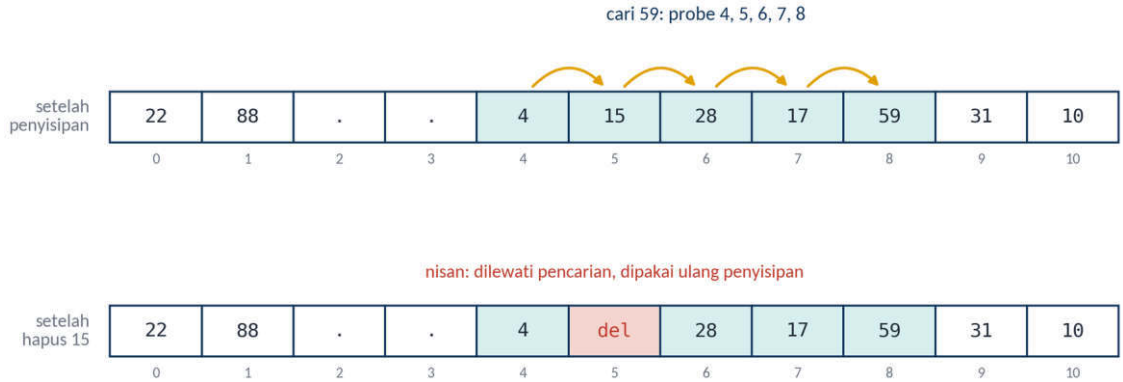
Dengan tangan, selangkah demi selangkah. Ini latihan klasik dari Cormen dkk. [9]. Program mencetak setiap barisan *probe*:

```
h(k) = k mod 11, linear probing
insert 10: probes 10 -> slot 10
insert 22: probes 0 -> slot 0
insert 31: probes 9 -> slot 9
insert  4: probes 4 -> slot 4
insert 15: probes 4 5 -> slot 5
insert 28: probes 6 -> slot 6
insert 17: probes 6 7 -> slot 7
insert 88: probes 0 1 -> slot 1
insert 59: probes 4 5 6 7 8 -> slot 8
  slot:   0   1   2   3   4   5   6   7   8   9  10
  key:   22  88   .   .   4  15  28  17  59  31  10

remove 15: true
  slot:   0   1   2   3   4   5   6   7   8   9  10
  key:   22  88   .   .   4 del  28  17  59  31  10
search 59: probes 4 5 6 7 8 -> slot 8
search 26: probes 4 5 6 7 8 9 10 0 1 2 -> not found
insert 37: probes 4 5 6 7 8 9 10 0 1 2 -> slot 5 (the tombstone)
  slot:   0   1   2   3   4   5   6   7   8   9  10
  key:   22  88   .   .   4  37  28  17  59  31  10
```

Empat kunci pertama mendarat di rumahnya: $10 \to 10$, $22 \to 0$, $31 \to 9$, $4 \to 4$. Lalu $15 \bmod 11 = 4$ bertabrakan dengan 4 dan pindah ke 5. $28 \to 6$. $17 \bmod 11 = 6$ bertabrakan dan pindah ke 7. $88 \bmod 11 = 0$ bertabrakan dengan 22 dan pindah ke 1. Dan $59 \bmod 11 = 4$ mendapati 4, 5, 6, 7 terisi semua dan mendarat di 8 setelah lima *probe*. Kunci-kunci yang ter-hash berdekatan telah melebur menjadi satu *klaster*, slot 4 sampai 10 dan berputar ke 0 dan 1, dan setiap kunci baru yang ter-hash ke dalamnya harus berjalan sampai ujungnya: *primary clustering* inilah kelemahan *linear probing*.

Sekarang penghapusannya. Slot 5 berisi 15, dan lintasan *probe* 59 melewatinya. `remove` kita menandai slot 5 dengan `del`, sebuah *tombstone*. Pencarian 59 memeriksa 4 (berisi 4, bukan 59), 5 (*tombstone*: terus), 6, 7, 8: ketemu. Pencarian 26 ($26 \bmod 11 = 4$) menelusuri seluruh klaster, 4 sampai 10 lalu 0 dan 1, dan berhenti pada slot kosong 2: tidak ada, setelah sepuluh *probe*. Terakhir 37 ($37 \bmod 11 = 4$): penyisipan mengingat *tombstone* di 5 sebagai slot bebas pertama, tetapi terus memeriksa sampai slot kosong 2 untuk memastikan 37 belum tersimpan lebih jauh; ternyata belum, sehingga 37 masuk ke slot 5, dan *tombstone* itu terpakai ulang. **Gambar 16.14** memperlihatkan tabel sebelum dan sesudah penghapusan.



Gambar 16.14 *Linear probing* dengan $m = 11$: lintasan *probe* 59 melalui slot 4 sampai 8. Menghapus 15 meninggalkan *tombstone* di slot 5, yang dilewati pencarian dan dipakai ulang penyisipan.

Ongkos. Untuk *load factor* $\alpha < 1$ dan fungsi hash yang menyebarkan kunci secara merata, analisis klasik Knuth tentang *linear probing* [12] memberikan rata-rata banyaknya *probe* untuk pencarian yang berhasil, S, dan yang gagal, U:

$$S \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right), \qquad U \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right). \tag{16.11}$$

Pada $\alpha = 0.5$, Persamaan (16.11) memberikan 1,5 dan 2,5 *probe*; pada $\alpha = 9/11 \approx 0.82$ milik kita hasilnya sekitar 3,3 dan 15,6, dan pencarian 26 yang gagal memang memerlukan sepuluh. Suku kuadratnya adalah alasan tabel *open addressing* dijaga jauh dari penuh: `dict` Python, yang memakai *open addressing*, menjaga muatannya paling banyak dua pertiga, dan bahkan `HashMap` Java yang berantai sudah tumbuh pada 0,75 [6], [7]. *Tombstone* ikut dihitung dalam muatan sampai *rehash* menyapunya.

Penyelesaian C++. **Kode 16.26** menyimpan larik kunci dan larik keadaan dengan tiga nilai, `EMPTY`, `FULL`, dan `DELETED`.

Kode 16.26 `p1609_linear_probing.cpp`: *open addressing* dengan *linear probing* dan *tombstone*, menjejak setiap barisan *probe*.

```cpp
// p1609_linear_probing.cpp - Problem 16.9: open addressing, and deleting safely.
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class ProbingTable {
    enum State { EMPTY, FULL, DELETED };                // DELETED = a "tombstone"
    vector<int> key;
    vector<State> state;
    int m;

    int h(int k) const { return k % m; }                // keys are non-negative

public:
    explicit ProbingTable(int size) : key(size, 0), state(size, EMPTY), m(size) {}

    // Returns the slot of k, or -1. A tombstone does NOT stop the search.
    int search(int k, string& trace) const {
        for (int j = 0; j < m; j++) {
            int i = (h(k) + j) % m;
            trace += " " + to_string(i);
            if (state[i] == EMPTY) return -1;           // k would have been here
            if (state[i] == FULL && key[i] == k) return i;
        }
        return -1;                                      // looked at every slot
    }

    // Returns the slot used, or -1 if k is already present or the table is full.
    int insert(int k, string& trace) {
        int firstFree = -1;
        for (int j = 0; j < m; j++) {
            int i = (h(k) + j) % m;
            trace += " " + to_string(i);
            if (state[i] == FULL) {
                if (key[i] == k) return -1;             // already present
                continue;
            }
            if (firstFree == -1) firstFree = i;         // remember the first hole
            if (state[i] == EMPTY) break;               // k is certainly absent
        }
        if (firstFree == -1) return -1;                 // full
        key[firstFree] = k;
        state[firstFree] = FULL;
        return firstFree;
```

Kode 16.26 (Lanjutan) *p1609_linear_probing.cpp: open addressing dengan linear probing dan tombstone, menjejak setiap barisan probe.*

```
    }

    bool remove(int k) {
        string ignored;
        int i = search(k, ignored);
        if (i == -1) return false;
        state[i] = DELETED;                    // NOT EMPTY: keep the runs whole
        return true;
    }

    void print() const {
        cout << "  slot:";
        for (int i = 0; i < m; i++) cout << setw(4) << i;
        cout << "\n  key: ";
        for (int i = 0; i < m; i++) {
            if (state[i] == FULL) cout << setw(4) << key[i];
            else cout << setw(4) << (state[i] == EMPTY ? "." : "del");
        }
        cout << "\n";
    }
};

int main() {
    ProbingTable t(11);
    cout << "h(k) = k mod 11, linear probing\n";
    for (int k : {10, 22, 31, 4, 15, 28, 17, 88, 59}) {
        string trace;
        int slot = t.insert(k, trace);
        cout << "insert " << setw(2) << k << ": probes" << trace << " -> slot "
             << slot << "\n";
    }
    t.print();

    cout << "\nremove 15: " << boolalpha << t.remove(15) << "\n";
    t.print();
    for (int k : {59, 26}) {
        string trace;
        int slot = t.search(k, trace);
        cout << "search " << k << ": probes" << trace << " -> "
             << (slot == -1 ? string("not found") : "slot " + to_string(slot))
             << "\n";
    }
    string trace;
    int slot = t.insert(37, trace);
    cout << "insert 37: probes" << trace << " -> slot " << slot
         << " (the tombstone)\n";
```

Kode 16.26 (Lanjutan) *p1609_linear_probing.cpp: open addressing dengan linear probing dan tombstone, menjejak setiap barisan probe.*

```
    t.print();
    return 0;
}
```

Baris demi baris. `search` berhenti pada `EMPTY` dan mengabaikan `DELETED`. `insert` mengingat slot pertama yang tidak `FULL` di `firstFree` tetapi terus berjalan sampai slot `EMPTY` membuktikan kuncinya tidak ada, dan menyerah setelah *m probe* bila tabel penuh. `remove` mengubah keadaan menjadi `DELETED`, tidak pernah `EMPTY`; komentarnya menyebutkan alasannya.

Penyelesaian Java dan Python. **Kode 16.27** dan **Kode 16.28** mencetak baris-baris yang sama; Java memakai `enum` untuk ketiga keadaan.

Kode 16.27 `P1609LinearProbing.java`: Soal 16.9 dalam Java.

```java
// P1609LinearProbing.java - Problem 16.9: open addressing, and deleting safely.
public class P1609LinearProbing {

    enum State { EMPTY, FULL, DELETED }                  // DELETED = a "tombstone"

    static class ProbingTable {
        final int[] key;
        final State[] state;
        final int m;

        ProbingTable(int size) {
            m = size;
            key = new int[m];
            state = new State[m];
            java.util.Arrays.fill(state, State.EMPTY);
        }

        int h(int k) {
            return k % m;                                // keys are non-negative
        }

        // Returns the slot of k, or -1. A tombstone does NOT stop the search.
        int search(int k, StringBuilder trace) {
            for (int j = 0; j < m; j++) {
                int i = (h(k) + j) % m;
                trace.append(" ").append(i);
                if (state[i] == State.EMPTY) return -1;  // k would have been here
                if (state[i] == State.FULL && key[i] == k) return i;
            }
            return -1;                                   // looked at every slot
        }

        // Returns the slot used, or -1 if k is present or the table is full.
        int insert(int k, StringBuilder trace) {
            int firstFree = -1;
            for (int j = 0; j < m; j++) {
                int i = (h(k) + j) % m;
                trace.append(" ").append(i);
                if (state[i] == State.FULL) {
                    if (key[i] == k) return -1;          // already present
                    continue;
                }
                if (firstFree == -1) firstFree = i;      // remember the first hole
                if (state[i] == State.EMPTY) break;      // k is certainly absent
            }
            if (firstFree == -1) return -1;              // full
```

Kode 16.27 (Lanjutan) *P1609LinearProbing.java: Soal 16.9 dalam Java.*

```
            key[firstFree] = k;
            state[firstFree] = State.FULL;
            return firstFree;
        }

        boolean remove(int k) {
            int i = search(k, new StringBuilder());
            if (i == -1) return false;
            state[i] = State.DELETED;             // NOT EMPTY: keep the runs whole
            return true;
        }

        void print() {
            StringBuilder a = new StringBuilder("  slot:");
            StringBuilder b = new StringBuilder("  key: ");
            for (int i = 0; i < m; i++) {
                a.append(String.format("%4d", i));
                if (state[i] == State.FULL) b.append(String.format("%4d", key[i]));
                else b.append(String.format("%4s",
                                            state[i] == State.EMPTY ? "." : "del"));
            }
            System.out.println(a + "\n" + b);
        }
    }

    public static void main(String[] args) {
        ProbingTable t = new ProbingTable(11);
        System.out.println("h(k) = k mod 11, linear probing");
        for (int k : new int[] {10, 22, 31, 4, 15, 28, 17, 88, 59}) {
            StringBuilder trace = new StringBuilder();
            int slot = t.insert(k, trace);
            System.out.printf("insert %2d: probes%s -> slot %d%n", k, trace, slot);
        }
        t.print();

        System.out.println("\nremove 15: " + t.remove(15));
        t.print();
        for (int k : new int[] {59, 26}) {
            StringBuilder trace = new StringBuilder();
            int slot = t.search(k, trace);
            System.out.println("search " + k + ": probes" + trace + " -> "
                    + (slot == -1 ? "not found" : "slot " + slot));
        }
        StringBuilder trace = new StringBuilder();
        int slot = t.insert(37, trace);
        System.out.println("insert 37: probes" + trace + " -> slot " + slot
```

Kode 16.27 (Lanjutan) *P1609LinearProbing.java: Soal 16.9 dalam Java.*

```
                + " (the tombstone)");
        t.print();
    }
}
```

Kode 16.28 `p1609_linear_probing.py`: Soal 16.9 dalam Python.

```
# p1609_linear_probing.py - Problem 16.9: open addressing, and deleting safely.
EMPTY, FULL, DELETED = 0, 1, 2              # DELETED = a "tombstone"


class ProbingTable:
    def __init__(self, m):
        self.m = m
        self.key = [0] * m
        self.state = [EMPTY] * m

    def h(self, k):
        return k % self.m                   # keys are non-negative

    def search(self, k, trace):
        """Returns the slot of k, or -1. A tombstone does NOT stop the search."""
        for j in range(self.m):
            i = (self.h(k) + j) % self.m
            trace.append(i)
            if self.state[i] == EMPTY:      # k would have been here
                return -1
            if self.state[i] == FULL and self.key[i] == k:
                return i
        return -1                           # looked at every slot

    def insert(self, k, trace):
        """Returns the slot used, or -1 if k is present or the table is full."""
        first_free = -1
        for j in range(self.m):
            i = (self.h(k) + j) % self.m
            trace.append(i)
            if self.state[i] == FULL:
                if self.key[i] == k:        # already present
                    return -1
                continue
            if first_free == -1:            # remember the first hole
                first_free = i
            if self.state[i] == EMPTY:      # k is certainly absent
                break
        if first_free == -1:                # full
            return -1
        self.key[first_free] = k
        self.state[first_free] = FULL
        return first_free
```

Kode 16.28 (Lanjutan) *p1609_linear_probing.py: Soal 16.9 dalam Python.*

```
    def remove(self, k):
        i = self.search(k, [])
        if i == -1:
            return False
        self.state[i] = DELETED             # NOT EMPTY: keep chains whole
        return True

    def print(self):
        print("  slot:" + "".join(f"{i:4d}" for i in range(self.m)))
        cells = [f"{self.key[i]:4d}" if self.state[i] == FULL else
                 f"{'.' if self.state[i] == EMPTY else 'del':>4}"
                 for i in range(self.m)]
        print("  key: " + "".join(cells))


def probes(trace):
    return "".join(f" {i}" for i in trace)


def main():
    t = ProbingTable(11)
    print("h(k) = k mod 11, linear probing")
    for k in (10, 22, 31, 4, 15, 28, 17, 88, 59):
        trace = []
        slot = t.insert(k, trace)
        print(f"insert {k:2d}: probes{probes(trace)} -> slot {slot}")
    t.print()

    print("\nremove 15:", str(t.remove(15)).lower())
    t.print()
    for k in (59, 26):
        trace = []
        slot = t.search(k, trace)
        print(f"search {k}: probes{probes(trace)} -> "
              + ("not found" if slot == -1 else f"slot {slot}"))
    trace = []
    slot = t.insert(37, trace)
    print(f"insert 37: probes{probes(trace)} -> slot {slot} (the tombstone)")
    t.print()


if __name__ == "__main__":
    main()
```

Penyelesaian keliru yang menggoda. Menghapus dengan mengosongkan slot adalah gagasan pertama yang wajar, dan tampaknya berhasil, karena kunci yang dihapus memang tidak lagi ditemukan. `wrong/p1609_wrong.cpp` menghapus 15 dengan cara itu lalu mencari 59:

```
before: search 59 -> slot 8
after removing 15: search 59 -> slot -1
```

59 tidak dihapus; ia masih berada di slot 8. Namun pencariannya dimulai di 4, menemukan 4, pindah ke 5, melihat slot kosong, dan menyimpulkan menurut aturan berhenti bahwa 59 tidak pernah disisipkan. Tabel kini *kehilangan* kunci yang dipegangnya, yang di katalog perpustakaan berarti buku yang ada di rak tetapi menurut komputer tidak ada. Kekeliruan yang sama tidak dapat terjadi dengan *chaining*, karena melepas simpul dari rantai tidak memengaruhi kunci lain; itulah salah satu alasan Bab 15 membandingkan kedua strategi.

> **UJI DIRI.** Mulai dari tabel setelah kesembilan penyisipan, hapus 17 (dengan *tombstone*) lalu sisipkan 50 ($50 \bmod 11 = 6$). Slot mana yang diperiksa penyisipan itu, dan di mana 50 ditempatkan? Apakah jawabannya berubah bila 17 dihapus dengan mengosongkan slotnya?

Jawaban. 50 memeriksa 6 (28), 7 (*tombstone*, diingat), 8 (59), 9 (31), 10 (10), 0 (22), 1 (88), 2 (kosong: 50 pasti tidak ada) lalu masuk ke slot 7, *tombstone* yang diingat. Bila slotnya dikosongkan, 50 akan langsung berhenti di 7 dan masuk ke sana, kebetulan tempat yang sama, tetapi tabelnya rusak untuk setiap kunci yang lintasannya melalui 7: 59 masih ditemukan (lintasannya 4, 5, 6, 7 kini mencapai 50 di 7 lalu berlanjut ke 8), namun penghapusan 50 berikutnya dengan pengosongan akan kembali menghilangkan 59. *Tombstone* membuat setiap barisan operasi aman, bukan hanya barisan yang ini.

16.8 Memilih Struktur Data yang Tepat

Setiap bab dalam buku ini menjawab pertanyaan berbentuk “bagaimana struktur ini bekerja?”. Pertanyaan yang dihadapi pemrogram di dunia kerja adalah kebalikannya: *inilah tugasnya; struktur yang mana?* Subbab ini menjawabnya dua kali, pertama dengan tabel yang memadatkan seluruh buku, lalu dengan pengukuran setiap struktur pada tugas nyata yang sama, agar tabel itu tidak menjadi soal kepercayaan belaka.

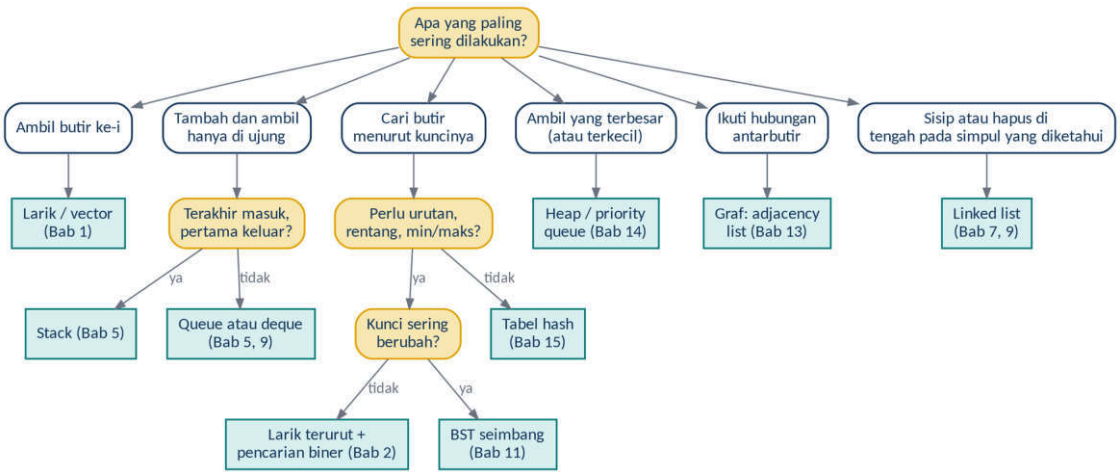
16.8.1 Mulailah dari operasinya, bukan dari strukturnya

Struktur data dipilih berdasarkan operasi yang paling sering dikerjakan tugas, dan seberapa sering setiap operasi dikerjakan. Katalog sepuluh ribu buku yang sama memerlukan struktur berbeda bila pustakawan paling sering mencari buku menurut kodenya, paling sering mendaftar buku di antara dua kode, paling sering menanyakan sepuluh buku terpopuler, atau paling sering menelusuri “orang-orang yang pernah menulis bersama orang ini”. Jadi langkah pertama selalu Langkah 2 metode kita: tuliskan satu kalimat yang menamai operasi dominannya. **Tabel 16.4** kemudian mengubah kalimat itu menjadi sebuah struktur.

Tabel 16.4 Memilih struktur data menurut operasi yang paling sering dikerjakan tugas: seluruh buku dalam satu halaman. n adalah banyaknya butir, r banyaknya hasil kueri rentang.

Tugas paling sering perlu…	Pilih	Ongkos operasi itu	Waspadai	Bab
membaca atau menulis butir ke-i	larik / `vector`	$O(1)$	penyisipan di tengah $O(n)$	1
mencari butir, jarang, tanpa persiapan	larik tak terurut + pencarian sekuensial	$O(n)$	cocok untuk n kecil atau pencarian tunggal	2
mencari menurut kunci, sekaligus mendaftar kunci secara berurutan atau dalam rentang, dengan perubahan yang jarang	larik terurut + pencarian biner	pencarian $O(\lg n)$, $O(\lg n + r)$ untuk r hasil	setiap penyisipan menggeser $O(n)$ butir	2, 3
menambah dan mengambil hanya di ujung terbaru (*undo*, kurung, DFS)	*stack*	$O(1)$	*pop* pada *stack* kosong	5
melayani menurut urutan kedatangan (antrean, BFS)	*queue*, atau *deque* untuk kedua ujung	$O(1)$	larik linear bergeser; pakai yang sirkular	5, 9
menyisipkan atau menghapus di tengah pada simpul yang sudah dipegang	*linked list*	$O(1)$	menemukan simpulnya tetap $O(n)$; lokalitas *cache* buruk	7, 9
mencari menurut kunci, sekaligus terkecil, terbesar, pendahulu, penerus, atau rentang, dengan perubahan yang sering	BST seimbang (`std::map`, `TreeMap`)	$O(\lg n)$	BST tak seimbang merosot menjadi $O(n)$ pada masukan terurut	11
berulang kali mengambil butir terbesar atau terkecil sementara butir terus berdatangan	*heap* / *priority queue*	sisip dan ambil $O(\lg n)$, intip $O(1)$	*heap* tidak dapat mencari; bawaan C++ adalah *max-heap*	14
k terbesar dari banyak butir, atau median berjalan	*min-heap* berukuran k; dua *heap*	$O(n \lg k)$; $O(\lg n)$ per butir	arah *heap* (**Tabel 16.3**)	14
mencari, menyisipkan, atau menghapus menurut kunci, tidak lebih	tabel hash (`unordered_map`, `HashMap`, `dict`)	rata-rata $O(1)$	tanpa urutan, tanpa rentang; hash buruk atau tabel penuh merosot ke $O(n)$	15
memodelkan hubungan: rute, ketergantungan, "siapa terhubung dengan siapa"	graf dengan *adjacency list*	BFS/DFS $O(\|V\| + \|E\|)$	matriks ketetanggaan memakan memori $\|V\|^2$	13
langkah paling sedikit / urutan yang menghormati ketergantungan / rute tercepat	BFS / Kahn / Dijkstra dengan *heap*	$O(\|V\| + \|E\|)$ / sama / $O((\|V\| + \|E\|)\lg\|V\|)$	kedalaman DFS bukan jarak; bobot negatif merusak Dijkstra	13, 14

Gambar 16.15 menggambar penalaran yang sama sebagai pohon keputusan. Ia bertanya lebih dahulu *apa yang paling sering Anda kerjakan*, baru kemudian satu atau dua pertanyaan lanjutan. Cabangnya yang terpenting adalah yang di tengah: "cari menurut kunci" berujung pada tabel hash hanya bila Anda *tidak pernah* memerlukan urutan; begitu tugas juga memerlukan "yang terkecil", "yang berikutnya", atau "semua di antara", jawabannya menjadi pohon pencarian atau, bila datanya jarang berubah, larik terurut.



Gambar 16.15 Memilih struktur: mulailah dari operasi yang paling sering dikerjakan tugas, lalu jawab paling banyak dua pertanyaan lanjutan. Bab yang mengajarkan setiap struktur tertulis di dalam kurung.

Dua peringatan menjaga tabel itu tetap jujur. Pertama, ongkos-ongkosnya adalah *laju pertumbuhan*, dan untuk n kecil faktor konstanlah yang menentukan; pencarian sekuensial atas dua puluh butir sering lebih cepat daripada meng-hash-nya. Kedua, tugas nyata jarang hanya memiliki satu operasi, dan program nyata menggabungkan struktur, seperti Soal 16.2 menggabungkan dua *hash map* dengan penelusuran sepanjang rantai, dan Soal 16.5 menggabungkan *adjacency list* dengan *heap*. Tabel itu memberi tahu struktur mana yang semestinya memikul operasi yang *dominan*.

16.8.2 Percobaannya

Katalog sepuluh ribu buku Pustaka Ceria (Bab 2, dari himpunan data goodbooks-10k [13]) adalah bahan uji yang realistis: identitas nyata, yang berkisar dari 1 sampai lebih dari 33 juta dan jauh dari tersebar merata, banyaknya penilaian yang nyata, dan nama-nama nyata orang yang dicantumkan pada setiap buku. **Kode 16.29** memuatnya dan mengukur enam struktur pada empat operasi.

- **Membangun**: menyimpan 10.000 identitas, menurut urutan berkas, yaitu urutan popularitas.
- **Mencari**: 2.000 identitas, 1.000 di antaranya ada (setiap buku kesepuluh) dan 1.000 tidak ada (nomor bebas berikutnya setelah setiap buku kesepuluh, dengan geseran lima).
- **Rentang**: 200 pertanyaan "berapa buku yang identitasnya berada di $[a, a + 50{,}000)$?", dengan a tersebar di seluruh rentang.
- **Menyisipkan**: 1.000 buku baru (1.000 identitas yang tidak ada tadi) ke salinan baru struktur itu.

Keenam strukturnya adalah larik tak terurut (pencarian sekuensial), larik terurut (pencarian biner, `lower_bound` untuk rentang), *singly linked list*, pohon pencarian biner tak seimbang dari Bab 11 dengan kunci disisipkan menurut urutan berkas, pohon pencarian seimbang dari pustaka, dan tabel hash dari pustaka. Setiap struktur harus menemukan 1.000 buku yang sama dan menghitung banyaknya buku yang sama di dalam rentang-rentang itu; program memeriksanya. Setiap pengukuran adalah yang terbaik dari lima kali jalan, untuk mengurangi gangguan program lain di komputer. Dua percobaan lagi menyusul: sepuluh buku dengan penilaian terbanyak dengan *heap* berukuran 10 dibandingkan pengurutan penuh, dan graf orang-orang yang dicantumkan pada buku yang sama.

Kode 16.29 `p1610_choose.cpp`: enam struktur diukur pada katalog yang sama (membangun, mencari, kueri rentang, menyisipkan), sepuluh buku dengan penilaian terbanyak dengan *heap* dan dengan pengurutan, serta graf orang-orang yang dicantumkan pada buku yang sama.

```
// p1610_choose.cpp - one catalog, every structure of the book, measured.
#include <algorithm>
#include <chrono>
#include <cstdio>
#include <forward_list>
#include <fstream>
#include <functional>
#include <map>
#include <queue>
#include <sstream>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <vector>
using namespace std;

struct Book {
    long long id, ratings;
    int year;
    string title, authors;
};

// ------------- the binary search tree of Chapter 11 (not balanced) -------------
struct BST {
    struct Node {
        long long key;
        const Book* book;
        Node *left = nullptr, *right = nullptr;
    };
    Node* root = nullptr;
    size_t count = 0;
    BST() = default;
    BST(const BST&) = delete;
    BST& operator=(const BST&) = delete;
    ~BST() { destroy(root); }
    static void destroy(Node* p) {
        if (!p) return;
        destroy(p->left);
        destroy(p->right);
        delete p;
    }
    void insert(long long key, const Book* b) {
        Node** link = &root;
        while (*link) link = key < (*link)->key ? &(*link)->left : &(*link)->right;
        *link = new Node{key, b};
        count++;
```

Kode 16.29 (Lanjutan) *p1610_choose.cpp: enam struktur diukur pada katalog yang sama (membangun, mencari, kueri rentang, menyisipkan), sepuluh buku dengan penilaian terbanyak dengan heap dan dengan pengurutan, serta graf orang-orang yang dicantumkan pada buku yang sama.*

```
    }
    const Book* find(long long key) const {
        Node* p = root;
        while (p && p->key != key) p = key < p->key ? p->left : p->right;
        return p ? p->book : nullptr;
    }
    static long long countRange(const Node* p, long long lo, long long hi) {
        if (!p) return 0;
        if (p->key < lo) return countRange(p->right, lo, hi);     // skip left side
        if (p->key >= hi) return countRange(p->left, lo, hi);     // skip right side
        return 1 + countRange(p->left, lo, hi) + countRange(p->right, lo, hi);
    }
    static int height(const Node* p) {
        return p ? 1 + max(height(p->left), height(p->right)) : 0;
    }
};

// Runs f several times and returns the best time in milliseconds.
double bestMs(int reps, const function<void()>& f) {
    double best = 1e300;
    for (int r = 0; r < reps; r++) {
        auto t0 = chrono::steady_clock::now();
        f();
        auto t1 = chrono::steady_clock::now();
        best = min(best, chrono::duration<double, milli>(t1 - t0).count());
    }
    return best;
}

// Like bestMs, but make() builds a fresh structure first, and only op() is timed.
template <class S>
double bestMsAfter(int reps, const function<S()>& make,
                   const function<void(S&)>& op) {
    double best = 1e300;
    for (int r = 0; r < reps; r++) {
        S c = make();
        auto t0 = chrono::steady_clock::now();
        op(c);
        auto t1 = chrono::steady_clock::now();
        best = min(best, chrono::duration<double, milli>(t1 - t0).count());
    }
    return best;
}

volatile long long sink;          // a result the optimizer cannot throw away
```

Kode 16.29 (Lanjutan) *p1610_choose.cpp: enam struktur diukur pada katalog yang sama (membangun, mencari, kueri rentang, menyisipkan), sepuluh buku dengan penilaian terbanyak dengan heap dan dengan pengurutan, serta graf orang-orang yang dicantumkan pada buku yang sama.*

```
int main() {
    ifstream in("../data/catalog.tsv");
    if (!in) {
        printf("cannot open ../data/catalog.tsv\n");
        return 1;
    }
    vector<Book> books;
    string line;
    getline(in, line);
    while (getline(in, line)) {
        stringstream ss(line);
        string f[5];
        for (auto& x : f) getline(ss, x, '\t');
        books.push_back({stoll(f[0]), stoll(f[2]), stoi(f[1]), f[3], f[4]});
    }
    size_t n = books.size();

    // 1,000 identifiers that are present and 1,000 that are absent.
    unordered_set<long long> present;
    for (const Book& b : books) present.insert(b.id);
    vector<long long> queries, fresh;
    for (size_t i = 0; i < n; i += 10) queries.push_back(books[i].id);
    for (size_t i = 5; i < n; i += 10) {
        long long q = books[i].id + 1;
        while (present.count(q)) q++;
        queries.push_back(q);
        fresh.push_back(q);                 // also used as the 1,000 new books
    }
    vector<pair<long long, long long>> ranges;       // 200 ranges [lo, lo + 50,000)
    for (long long i = 0; i < 200; i++)
        ranges.push_back({i * 150000, i * 150000 + 50000});

    printf("Catalog: %zu books; %zu lookups (half absent); %zu range queries; "
           "%zu inserts\n\n", n, queries.size(), ranges.size(), fresh.size());
    printf("%-28s%10s%12s%12s%12s\n", "structure", "build ms", "lookup us",
           "range us", "insert us");
    const int R = 5;
    vector<long long> answers;              // all structures must give the same
    auto row = [&](const char* name, double b, double l, double r, double ins,
                   long long found, long long inRange) {
        printf("%-28s%10.3f%12.3f%12.3f%12.3f\n", name, b,
               1000 * l / queries.size(), 1000 * r / ranges.size(),
               1000 * ins / fresh.size());
        answers.push_back(found);
        answers.push_back(inRange);
    };
```

Kode 16.29 (Lanjutan) *p1610_choose.cpp: enam struktur diukur pada katalog yang sama (membangun, mencari, kueri rentang, menyisipkan), sepuluh buku dengan penilaian terbanyak dengan heap dan dengan pengurutan, serta graf orang-orang yang dicantumkan pada buku yang sama.*

```
    {   // unsorted array: append in arrival order, search by walking
        vector<long long> a;
        double tb = bestMs(R, [&] {
            a.clear();
            for (const Book& b : books) a.push_back(b.id);
        });
        long long found = 0, cnt = 0;
        double tl = bestMs(R, [&] {
            found = 0;
            for (long long q : queries)
                for (long long x : a)
                    if (x == q) { found++; break; }
        });
        double tr = bestMs(R, [&] {
            cnt = 0;
            for (auto [lo, hi] : ranges)
                for (long long x : a) cnt += (x >= lo && x < hi);
        });
        double ti = bestMsAfter<vector<long long>>(R, [&] { return a; },
            [&](vector<long long>& c) {
                for (long long q : fresh) c.push_back(q);
            });
        row("unsorted array", tb, tl, tr, ti, found, cnt);
    }
    {   // sorted array: sort once, binary search (Chapter 2)
        vector<long long> a;
        double tb = bestMs(R, [&] {
            a.clear();
            for (const Book& b : books) a.push_back(b.id);
            sort(a.begin(), a.end());
        });
        long long found = 0, cnt = 0;
        double tl = bestMs(R, [&] {
            found = 0;
            for (long long q : queries)
                found += binary_search(a.begin(), a.end(), q);
        });
        double tr = bestMs(R, [&] {
            cnt = 0;
            for (auto [lo, hi] : ranges)
                cnt += lower_bound(a.begin(), a.end(), hi) -
                       lower_bound(a.begin(), a.end(), lo);
        });
        double ti = bestMsAfter<vector<long long>>(R, [&] { return a; },
            [&](vector<long long>& c) {
```

Kode 16.29 (Lanjutan) *p1610_choose.cpp: enam struktur diukur pada katalog yang sama (membangun, mencari, kueri rentang, menyisipkan), sepuluh buku dengan penilaian terbanyak dengan heap dan dengan pengurutan, serta graf orang-orang yang dicantumkan pada buku yang sama.*

```
            for (long long q : fresh)
                c.insert(upper_bound(c.begin(), c.end(), q), q);
        });
    row("sorted array + binary search", tb, tl, tr, ti, found, cnt);
}
{   // singly linked list: push at the front (Chapter 7)
    forward_list<long long> L;
    double tb = bestMs(R, [&] {
        L.clear();
        for (const Book& b : books) L.push_front(b.id);
    });
    long long found = 0, cnt = 0;
    double tl = bestMs(R, [&] {
        found = 0;
        for (long long q : queries)
            for (long long x : L)
                if (x == q) { found++; break; }
    });
    double tr = bestMs(R, [&] {
        cnt = 0;
        for (auto [lo, hi] : ranges)
            for (long long x : L) cnt += (x >= lo && x < hi);
    });
    double ti = bestMsAfter<forward_list<long long>>(R, [&] { return L; },
        [&](forward_list<long long>& c) {
            for (long long q : fresh) c.push_front(q);
        });
    row("singly linked list", tb, tl, tr, ti, found, cnt);
}
int bstHeight = 0;
{   // binary search tree of Chapter 11, keys inserted in arrival order
    double tb = bestMs(R, [&] {
        BST t;
        for (const Book& b : books) t.insert(b.id, &b);
        sink = t.count;
    });
    BST t;
    for (const Book& b : books) t.insert(b.id, &b);
    bstHeight = BST::height(t.root);
    long long found = 0, cnt = 0;
    double tl = bestMs(R, [&] {
        found = 0;
        for (long long q : queries) found += t.find(q) != nullptr;
    });
    double tr = bestMs(R, [&] {
        cnt = 0;
```

Kode 16.29 (Lanjutan) *p1610_choose.cpp: enam struktur diukur pada katalog yang sama (membangun, mencari, kueri rentang, menyisipkan), sepuluh buku dengan penilaian terbanyak dengan heap dan dengan pengurutan, serta graf orang-orang yang dicantumkan pada buku yang sama.*

```
            for (auto [lo, hi] : ranges) cnt += BST::countRange(t.root, lo, hi);
        });
        double ti = 1e300;                  // a BST is not copied: rebuild, untimed
        for (int r = 0; r < R; r++) {
            BST c;
            for (const Book& b : books) c.insert(b.id, &b);
            ti = min(ti, bestMs(1, [&] {
                for (long long q : fresh) c.insert(q, nullptr);
            }));
        }
        row("BST (Chapter 11, unbalanced)", tb, tl, tr, ti, found, cnt);
    }
    {   // balanced search tree: std::map is a red-black tree
        map<long long, const Book*> t;
        double tb = bestMs(R, [&] {
            t.clear();
            for (const Book& b : books) t.emplace(b.id, &b);
        });
        long long found = 0, cnt = 0;
        double tl = bestMs(R, [&] {
            found = 0;
            for (long long q : queries) found += t.count(q);
        });
        double tr = bestMs(R, [&] {
            cnt = 0;
            for (auto [lo, hi] : ranges)
                cnt += distance(t.lower_bound(lo), t.lower_bound(hi));
        });
        using M = map<long long, const Book*>;
        double ti = bestMsAfter<M>(R, [&] { return t; },
            [&](M& c) { for (long long q : fresh) c.emplace(q, nullptr); });
        row("balanced BST (std::map)", tb, tl, tr, ti, found, cnt);
    }
    {   // hash table (Chapter 15): std::unordered_map
        unordered_map<long long, const Book*> h;
        double tb = bestMs(R, [&] {
            h.clear();
            for (const Book& b : books) h.emplace(b.id, &b);
        });
        long long found = 0, cnt = 0;
        double tl = bestMs(R, [&] {
            found = 0;
            for (long long q : queries) found += h.count(q);
        });
        double tr = bestMs(R, [&] {         // no order: every range scans it all
            cnt = 0;
```

Kode 16.29 (Lanjutan) *p1610_choose.cpp: enam struktur diukur pada katalog yang sama (membangun, mencari, kueri rentang, menyisipkan), sepuluh buku dengan penilaian terbanyak dengan heap dan dengan pengurutan, serta graf orang-orang yang dicantumkan pada buku yang sama.*

```
            for (auto [lo, hi] : ranges)
                for (const auto& kv : h) cnt += (kv.first >= lo && kv.first < hi);
        });
        using H = unordered_map<long long, const Book*>;
        double ti = bestMsAfter<H>(R, [&] { return h; },
            [&](H& c) { for (long long q : fresh) c.emplace(q, nullptr); });
        row("hash table (unordered_map)", tb, tl, tr, ti, found, cnt);
    }
    bool agree = true;
    for (size_t i = 2; i < answers.size(); i++)
        agree = agree && answers[i] == answers[i % 2];
    printf("All %zu structures found %lld of the %zu lookups and counted "
           "%lld books\nin the ranges: %s\n", answers.size() / 2, answers[0],
           queries.size(), answers[1], agree ? "yes" : "no");
    printf("The BST built in arrival order has %d levels; a balanced one has 14.\n",
           bstHeight);

    // ------------- the 10 most-rated books: heap against sorting -------------
    vector<long long> top1, top2;
    double th = bestMs(R, [&] {             // min-heap of size 10 (Chapter 14)
        priority_queue<long long, vector<long long>, greater<long long>> heap;
        for (const Book& b : books) {
            if (heap.size() < 10) heap.push(b.ratings);
            else if (b.ratings > heap.top()) { heap.pop(); heap.push(b.ratings); }
        }
        top1.clear();
        while (!heap.empty()) { top1.push_back(heap.top()); heap.pop(); }
        reverse(top1.begin(), top1.end());
    });
    double ts = bestMs(R, [&] {             // sort all 10,000 (Chapter 3)
        vector<long long> r;
        for (const Book& b : books) r.push_back(b.ratings);
        sort(r.rbegin(), r.rend());
        top2.assign(r.begin(), r.begin() + 10);
    });
    printf("\nTop 10 by ratings: heap of size 10 %.3f ms, sorting everything "
           "%.3f ms\n  (same answer: %s)\n", th, ts, top1 == top2 ? "yes" : "no");
```

Kode 16.29 (Lanjutan) *p1610_choose.cpp: enam struktur diukur pada katalog yang sama (membangun, mencari, kueri rentang, menyisipkan), sepuluh buku dengan penilaian terbanyak dengan heap dan dengan pengurutan, serta graf orang-orang yang dicantumkan pada buku yang sama.*

```
    // ------------- the graph of people credited on the same book -------------
    unordered_map<string, int> idOf;        // name -> vertex number (hashing again)
    vector<string> name;
    vector<vector<int>> adj;
    auto t0 = chrono::steady_clock::now();
    for (const Book& b : books) {
        vector<int> who;
        stringstream ss(b.authors);
        string a;
        while (getline(ss, a, ',')) {
            if (!a.empty() && a[0] == ' ') a.erase(0, 1);
            auto [it, isNew] = idOf.emplace(a, (int)name.size());
            if (isNew) {
                name.push_back(a);
                adj.emplace_back();
            }
            if (find(who.begin(), who.end(), it->second) == who.end())
                who.push_back(it->second);
        }
        for (int u : who)
            for (int v : who)
                if (u != v && find(adj[u].begin(), adj[u].end(), v) == adj[u].end())
                    adj[u].push_back(v);
    }
    size_t V = name.size(), E = 0;
    for (auto& a : adj) E += a.size();
    E /= 2;
    vector<int> comp(V, -1);                // label every connected component (BFS)
    int comps = 0;
    size_t largest = 0;
    for (size_t s = 0; s < V; s++) {
        if (comp[s] != -1) continue;
        size_t size = 0;
        queue<int> q;
        q.push(s);
        comp[s] = comps;
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            size++;
            for (int v : adj[u])
                if (comp[v] == -1) { comp[v] = comps; q.push(v); }
        }
        largest = max(largest, size);
        comps++;
    }
```

Kode 16.29 (Lanjutan) *p1610_choose.cpp: enam struktur diukur pada katalog yang sama (membangun, mencari, kueri rentang, menyisipkan), sepuluh buku dengan penilaian terbanyak dengan heap dan dengan pengurutan, serta graf orang-orang yang dicantumkan pada buku yang sama.*

```
    auto t1 = chrono::steady_clock::now();
    double tg = chrono::duration<double, milli>(t1 - t0).count();
    printf("\nPeople graph: %zu people, %zu co-credit edges, %d components,\n"
           "  the largest has %zu people\n", V, E, comps, largest);
    printf("  adjacency lists hold %zu entries; an adjacency matrix would need\n"
           "  %zu cells; built and explored in %.3f ms\n", 2 * E, V * V, tg);

    int s = idOf.at("Stephen King"), t = idOf.at("Charles Dickens");
    vector<int> parent(V, -2);              // BFS from s, remembering parents
    queue<int> q;
    q.push(s);
    parent[s] = -1;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v : adj[u])
            if (parent[v] == -2) { parent[v] = u; q.push(v); }
    }
    vector<string> path;
    for (int v = t; v != -1; v = parent[v]) path.push_back(name[v]);
    reverse(path.begin(), path.end());
    printf("  fewest hops from Stephen King to Charles Dickens: %zu\n   ",
           path.size() - 1);
    for (size_t i = 0; i < path.size(); i++)   // three names to a line
        printf(" %s%s", path[i].c_str(),
               i + 1 == path.size() ? "\n" : (i % 3 == 2 ? " ->\n   " : " ->"));
    return 0;
}
```

```
Catalog: 10000 books; 2000 lookups (half absent); 200 range queries; 1000 inserts

structure                     build ms   lookup us    range us   insert us
unsorted array                   0.023       2.280       6.021       0.004
sorted array + binary search     0.481       0.073       0.048       0.634
singly linked list               0.136      13.652      18.038       0.007
BST (Chapter 11, unbalanced)     1.262       0.053       0.168       0.131
balanced BST (std::map)          1.459       0.088       0.252       0.124
hash table (unordered_map)       0.314       0.006      44.913       0.089
All 6 structures found 1000 of the 2000 lookups and counted 4702 books
in the ranges: yes
The BST built in arrival order has 35 levels; a balanced one has 14.

Top 10 by ratings: heap of size 10 0.006 ms, sorting everything 0.420 ms
  (same answer: yes)

People graph: 5841 people, 6997 co-credit edges, 3408 components,
  the largest has 605 people
  adjacency lists hold 13994 entries; an adjacency matrix would need
  34117281 cells; built and explored in 3.578 ms
  fewest hops from Stephen King to Charles Dickens: 5
    Stephen King -> Dave McKean -> Richard Dawkins ->
    Thomas Hardy -> Patricia Ingham -> Charles Dickens
```

Kode 16.30 dan **Kode 16.31** menjalankan percobaan yang sama dalam Java dan Python. Versi Java menyimpan identitas kedua lariknya dalam `long[]` biasa (sebuah `LongArray` buatan kita), karena `ArrayList<Long>` akan memuat sepuluh ribu objek terpisah dan mengukur beban objek Java alih-alih lariknya. Pustaka standar Python tidak memiliki pohon pencarian seimbang, sehingga baris itu ditandai tidak ada.

Kode 16.30 `P1610Choose.java`: percobaan Subbab 16.8 dalam Java, dengan `TreeMap` sebagai pohon pencarian seimbang dan `HashMap` sebagai tabel hash.

```java
// P1610Choose.java - one catalog, every structure of the book, measured.
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.PriorityQueue;
import java.util.TreeMap;
import java.util.function.Supplier;

public class P1610Choose {

    record Book(long id, int year, long ratings, String title, String authors) {}

    // A growable array of long values: the array holds numbers, not objects.
    static class LongArray {
        long[] a = new long[16];
        int n = 0;

        void add(long x) {
            if (n == a.length) a = Arrays.copyOf(a, 2 * n);
            a[n++] = x;
        }

        void insertAt(int i, long x) {          // shift a[i..n-1] one place right
            if (n == a.length) a = Arrays.copyOf(a, 2 * n);
            System.arraycopy(a, i, a, i + 1, n - i);
            a[i] = x;
            n++;
        }

        LongArray copy() {
            LongArray c = new LongArray();
            c.a = Arrays.copyOf(a, a.length);
            c.n = n;
            return c;
        }
    }
```

Kode 16.30 (Lanjutan) *P1610Choose.java: percobaan Subbab 16.8 dalam Java, dengan TreeMap sebagai pohon pencarian seimbang dan HashMap sebagai tabel hash.*

```
    // First index i with a[i] >= key (Chapter 2's lower bound).
    static int lowerBound(LongArray s, long key) {
        int lo = 0, hi = s.n;
        while (lo < hi) {
            int mid = (lo + hi) >>> 1;
            if (s.a[mid] < key) lo = mid + 1;
            else hi = mid;
        }
        return lo;
    }

    // ----------- the binary search tree of Chapter 11 (not balanced) -----------
    static class BST {
        static class Node {
            final long key;
            Node left, right;

            Node(long key) {
                this.key = key;
            }
        }

        Node root;

        void insert(long key) {
            if (root == null) {
                root = new Node(key);
                return;
            }
            Node p = root;
            while (true) {
                if (key < p.key) {
                    if (p.left == null) { p.left = new Node(key); return; }
                    p = p.left;
                } else {
                    if (p.right == null) { p.right = new Node(key); return; }
                    p = p.right;
                }
            }
        }
```

Kode 16.30 (Lanjutan) *P1610Choose.java: percobaan Subbab 16.8 dalam Java, dengan TreeMap sebagai pohon pencarian seimbang dan HashMap sebagai tabel hash.*

```
        boolean contains(long key) {
            Node p = root;
            while (p != null && p.key != key) p = key < p.key ? p.left : p.right;
            return p != null;
        }

        static long countRange(Node p, long lo, long hi) {
            if (p == null) return 0;
            if (p.key < lo) return countRange(p.right, lo, hi);    // skip left side
            if (p.key >= hi) return countRange(p.left, lo, hi);   // skip right side
            return 1 + countRange(p.left, lo, hi) + countRange(p.right, lo, hi);
        }

        static int height(Node p) {
            return p == null ? 0 : 1 + Math.max(height(p.left), height(p.right));
        }
    }

    // Runs f several times and returns the best time in milliseconds.
    static double bestMs(int reps, Runnable f) {
        double best = Double.MAX_VALUE;
        for (int r = 0; r < reps; r++) {
            long t0 = System.nanoTime();
            f.run();
            best = Math.min(best, (System.nanoTime() - t0) / 1e6);
        }
        return best;
    }

    interface Op<S> {
        void run(S s);
    }

    // Like bestMs, but make() builds a fresh structure first, and only op is timed.
    static <S> double bestMsAfter(int reps, Supplier<S> make, Op<S> op) {
        double best = Double.MAX_VALUE;
        for (int r = 0; r < reps; r++) {
            S c = make.get();
            long t0 = System.nanoTime();
            op.run(c);
            best = Math.min(best, (System.nanoTime() - t0) / 1e6);
        }
        return best;
    }
```

Kode 16.30 (Lanjutan) *P1610Choose.java: percobaan Subbab 16.8 dalam Java, dengan TreeMap sebagai pohon pencarian seimbang dan HashMap sebagai tabel hash.*

```
    static final int R = 5;
    static List<Book> books = new ArrayList<>();
    static List<Long> queries = new ArrayList<>(), fresh = new ArrayList<>();
    static long[][] ranges = new long[200][];
    static List<Long> answers = new ArrayList<>();
    static long found, cnt;

    static void row(String name, double b, double l, double r, double ins) {
        System.out.printf("%-28s%10.3f%12.3f%12.3f%12.3f%n", name, b,
                1000 * l / queries.size(), 1000 * r / ranges.length,
                1000 * ins / fresh.size());
        answers.add(found);
        answers.add(cnt);
    }

    public static void main(String[] args) throws IOException {
        List<String> lines = Files.readAllLines(Path.of("../data/catalog.tsv"),
                                                StandardCharsets.UTF_8);
        for (String line : lines.subList(1, lines.size())) {
            String[] f = line.split("\t", -1);
            books.add(new Book(Long.parseLong(f[0]), Integer.parseInt(f[1]),
                               Long.parseLong(f[2]), f[3], f[4]));
        }
        int n = books.size();

        // 1,000 identifiers that are present and 1,000 that are absent.
        HashSet<Long> present = new HashSet<>();
        for (Book b : books) present.add(b.id());
        for (int i = 0; i < n; i += 10) queries.add(books.get(i).id());
        for (int i = 5; i < n; i += 10) {
            long q = books.get(i).id() + 1;
            while (present.contains(q)) q++;
            queries.add(q);
            fresh.add(q);                       // also used as the 1,000 new books
        }
        for (int i = 0; i < 200; i++)
            ranges[i] = new long[] {i * 150000L, i * 150000L + 50000};

        System.out.printf("Catalog: %d books; %d lookups (half absent); %d range "
                + "queries; %d inserts%n%n", n, queries.size(), ranges.length,
                fresh.size());
        System.out.printf("%-28s%10s%12s%12s%12s%n", "structure", "build ms",
                "lookup us", "range us", "insert us");
```

Kode 16.30 (Lanjutan) *P1610Choose.java: percobaan Subbab 16.8 dalam Java, dengan TreeMap sebagai pohon pencarian seimbang dan HashMap sebagai tabel hash.*

```
        {   // unsorted array: append in arrival order, search by walking
            LongArray[] a = {null};
            double tb = bestMs(R, () -> {
                a[0] = new LongArray();
                for (Book b : books) a[0].add(b.id());
            });
            double tl = bestMs(R, () -> {
                found = 0;
                for (long q : queries)
                    for (int i = 0; i < a[0].n; i++)
                        if (a[0].a[i] == q) { found++; break; }
            });
            double tr = bestMs(R, () -> {
                cnt = 0;
                for (long[] g : ranges)
                    for (int i = 0; i < a[0].n; i++)
                        if (a[0].a[i] >= g[0] && a[0].a[i] < g[1]) cnt++;
            });
            double ti = bestMsAfter(R, a[0]::copy, c -> {
                for (long q : fresh) c.add(q);
            });
            row("unsorted array", tb, tl, tr, ti);
        }
        {   // sorted array: sort once, binary search (Chapter 2)
            LongArray[] a = {null};
            double tb = bestMs(R, () -> {
                a[0] = new LongArray();
                for (Book b : books) a[0].add(b.id());
                Arrays.sort(a[0].a, 0, a[0].n);
            });
            double tl = bestMs(R, () -> {
                found = 0;
                for (long q : queries) {
                    int i = lowerBound(a[0], q);
                    if (i < a[0].n && a[0].a[i] == q) found++;
                }
            });
            double tr = bestMs(R, () -> {
                cnt = 0;
                for (long[] g : ranges)
                    cnt += lowerBound(a[0], g[1]) - lowerBound(a[0], g[0]);
            });
            double ti = bestMsAfter(R, a[0]::copy, c -> {
                for (long q : fresh) c.insertAt(lowerBound(c, q), q);
            });
            row("sorted array + binary search", tb, tl, tr, ti);
```

Kode 16.30 (Lanjutan) *P1610Choose.java: percobaan Subbab 16.8 dalam Java, dengan TreeMap sebagai pohon pencarian seimbang dan HashMap sebagai tabel hash.*

```
        }
        {   // linked list (Chapter 7; java.util.LinkedList is doubly linked)
            List<LinkedList<Long>> L = new ArrayList<>(List.of(new LinkedList<>()));
            double tb = bestMs(R, () -> {
                LinkedList<Long> x = new LinkedList<>();
                for (Book b : books) x.addFirst(b.id());
                L.set(0, x);
            });
            double tl = bestMs(R, () -> {
                found = 0;
                for (long q : queries)
                    for (long x : L.get(0))
                        if (x == q) { found++; break; }
            });
            double tr = bestMs(R, () -> {
                cnt = 0;
                for (long[] g : ranges)
                    for (long x : L.get(0))
                        if (x >= g[0] && x < g[1]) cnt++;
            });
            double ti = bestMsAfter(R, () -> new LinkedList<>(L.get(0)), c -> {
                for (long q : fresh) c.addFirst(q);
            });
            row("linked list", tb, tl, tr, ti);
        }
        int bstHeight;
        {   // binary search tree of Chapter 11, keys inserted in arrival order
            BST[] t = {null};
            double tb = bestMs(R, () -> {
                t[0] = new BST();
                for (Book b : books) t[0].insert(b.id());
            });
            bstHeight = BST.height(t[0].root);
            double tl = bestMs(R, () -> {
                found = 0;
                for (long q : queries) if (t[0].contains(q)) found++;
            });
            double tr = bestMs(R, () -> {
                cnt = 0;
                for (long[] g : ranges)
                    cnt += BST.countRange(t[0].root, g[0], g[1]);
            });
            double ti = bestMsAfter(R, () -> {
                BST c = new BST();
                for (Book b : books) c.insert(b.id());
                return c;
```

Kode 16.30 (Lanjutan) *P1610Choose.java: percobaan Subbab 16.8 dalam Java, dengan TreeMap sebagai pohon pencarian seimbang dan HashMap sebagai tabel hash.*

```
            }, c -> {
                for (long q : fresh) c.insert(q);
            });
            row("BST (Chapter 11, unbalanced)", tb, tl, tr, ti);
        }
        {   // balanced search tree: java.util.TreeMap is a red-black tree
            List<TreeMap<Long, Book>> t = new ArrayList<>(List.of(new TreeMap<>()));
            double tb = bestMs(R, () -> {
                TreeMap<Long, Book> x = new TreeMap<>();
                for (Book b : books) x.put(b.id(), b);
                t.set(0, x);
            });
            double tl = bestMs(R, () -> {
                found = 0;
                for (long q : queries) if (t.get(0).containsKey(q)) found++;
            });
            double tr = bestMs(R, () -> {
                cnt = 0;
                for (long[] g : ranges) cnt += t.get(0).subMap(g[0], g[1]).size();
            });
            double ti = bestMsAfter(R, () -> new TreeMap<>(t.get(0)), c -> {
                for (long q : fresh) c.put(q, null);
            });
            row("balanced BST (TreeMap)", tb, tl, tr, ti);
        }
        {   // hash table (Chapter 15): java.util.HashMap
            List<HashMap<Long, Book>> h = new ArrayList<>(List.of(new HashMap<>()));
            double tb = bestMs(R, () -> {
                HashMap<Long, Book> x = new HashMap<>();
                for (Book b : books) x.put(b.id(), b);
                h.set(0, x);
            });
            double tl = bestMs(R, () -> {
                found = 0;
                for (long q : queries) if (h.get(0).containsKey(q)) found++;
            });
            double tr = bestMs(R, () -> {       // no order: each range scans it all
                cnt = 0;
                for (long[] g : ranges)
                    for (long k : h.get(0).keySet())
                        if (k >= g[0] && k < g[1]) cnt++;
            });
            double ti = bestMsAfter(R, () -> new HashMap<>(h.get(0)), c -> {
                for (long q : fresh) c.put(q, null);
            });
            row("hash table (HashMap)", tb, tl, tr, ti);
```

Kode 16.30 (Lanjutan) *P1610Choose.java: percobaan Subbab 16.8 dalam Java, dengan TreeMap sebagai pohon pencarian seimbang dan HashMap sebagai tabel hash.*

```
        }
        boolean agree = true;
        for (int i = 2; i < answers.size(); i++)
            agree = agree && answers.get(i).equals(answers.get(i % 2));
        System.out.printf("All %d structures found %d of the %d lookups and "
                + "counted %d books%nin the ranges: %s%n", answers.size() / 2,
                answers.get(0), queries.size(), answers.get(1),
                agree ? "yes" : "no");
        System.out.printf("The BST built in arrival order has %d levels; a "
                + "balanced one has 14.%n", bstHeight);

        // ------------- the 10 most-rated books: heap against sorting -------------
        List<Long> top1 = new ArrayList<>(), top2 = new ArrayList<>();
        double th = bestMs(R, () -> {           // min-heap of size 10 (Chapter 14)
            PriorityQueue<Long> heap = new PriorityQueue<>();
            for (Book b : books) {
                if (heap.size() < 10) heap.add(b.ratings());
                else if (b.ratings() > heap.peek()) {
                    heap.poll();
                    heap.add(b.ratings());
                }
            }
            top1.clear();
            while (!heap.isEmpty()) top1.add(heap.poll());
            Collections.reverse(top1);
        });
        double ts = bestMs(R, () -> {           // sort all 10,000 (Chapter 3)
            long[] r = new long[books.size()];
            for (int i = 0; i < r.length; i++) r[i] = books.get(i).ratings();
            Arrays.sort(r);
            top2.clear();
            for (int i = r.length - 1; i >= r.length - 10; i--) top2.add(r[i]);
        });
        System.out.printf("%nTop 10 by ratings: heap of size 10 %.3f ms, sorting "
                + "everything %.3f ms%n  (same answer: %s)%n", th, ts,
                top1.equals(top2) ? "yes" : "no");
```

Kode 16.30 (Lanjutan) *P1610Choose.java: percobaan Subbab 16.8 dalam Java, dengan TreeMap sebagai pohon pencarian seimbang dan HashMap sebagai tabel hash.*

```
        // ------------ the graph of people credited on the same book -------------
        long t0 = System.nanoTime();
        HashMap<String, Integer> idOf = new HashMap<>();    // name -> vertex number
        List<String> name = new ArrayList<>();
        List<List<Integer>> adj = new ArrayList<>();
        for (Book b : books) {
            List<Integer> who = new ArrayList<>();
            for (String a : b.authors().split(",")) {
                if (a.startsWith(" ")) a = a.substring(1);
                Integer id = idOf.get(a);
                if (id == null) {
                    id = name.size();
                    idOf.put(a, id);
                    name.add(a);
                    adj.add(new ArrayList<>());
                }
                if (!who.contains(id)) who.add(id);
            }
            for (int u : who)
                for (int v : who)
                    if (u != v && !adj.get(u).contains(v)) adj.get(u).add(v);
        }
        int V = name.size();
        long E = 0;
        for (List<Integer> a : adj) E += a.size();
        E /= 2;
        int[] comp = new int[V];               // label each connected component
        Arrays.fill(comp, -1);
        int comps = 0, largest = 0;
        for (int s = 0; s < V; s++) {
            if (comp[s] != -1) continue;
            int size = 0;
            ArrayDeque<Integer> q = new ArrayDeque<>();
            q.add(s);
            comp[s] = comps;
            while (!q.isEmpty()) {
                int u = q.poll();
                size++;
                for (int v : adj.get(u))
                    if (comp[v] == -1) { comp[v] = comps; q.add(v); }
            }
            largest = Math.max(largest, size);
            comps++;
        }
        double tg = (System.nanoTime() - t0) / 1e6;
        System.out.printf("%nPeople graph: %d people, %d co-credit edges, "
```

Kode 16.30 (Lanjutan) *P1610Choose.java: percobaan Subbab 16.8 dalam Java, dengan TreeMap sebagai pohon pencarian seimbang dan HashMap sebagai tabel hash.*

```
                + "%d components,%n"
                + "  the largest has %d people%n", V, E, comps, largest);
        System.out.printf("  adjacency lists hold %d entries; an adjacency matrix "
                + "would need%n  %d cells; built and explored in %.3f ms%n", 2 * E,
                (long) V * V, tg);

        int s = idOf.get("Stephen King"), t = idOf.get("Charles Dickens");
        int[] parent = new int[V];              // BFS from s, remembering parents
        Arrays.fill(parent, -2);
        ArrayDeque<Integer> q = new ArrayDeque<>();
        q.add(s);
        parent[s] = -1;
        while (!q.isEmpty()) {
            int u = q.poll();
            for (int v : adj.get(u))
                if (parent[v] == -2) { parent[v] = u; q.add(v); }
        }
        List<String> path = new ArrayList<>();
        for (int v = t; v != -1; v = parent[v]) path.add(name.get(v));
        Collections.reverse(path);
        StringBuilder sb = new StringBuilder("  fewest hops from Stephen King to "
                + "Charles Dickens: " + (path.size() - 1) + "\n    ");
        for (int i = 0; i < path.size(); i++)   // three names to a line
            sb.append(" ").append(path.get(i)).append(i + 1 == path.size() ? ""
                    : (i % 3 == 2 ? " ->\n    " : " ->"));
        System.out.println(sb);
    }
}
```

```
Catalog: 10000 books; 2000 lookups (half absent); 200 range queries; 1000 inserts

structure                     build ms   lookup us    range us   insert us
unsorted array                   1.046       1.546       9.619       0.085
sorted array + binary search     1.654       0.206       0.182       0.814
linked list                      1.184      19.141      29.328       0.111
BST (Chapter 11, unbalanced)     1.803       0.158       0.333       0.187
balanced BST (TreeMap)           3.057       0.269       1.763       0.318
hash table (HashMap)             1.966       0.169     113.180       0.272
All 6 structures found 1000 of the 2000 lookups and counted 4702 books
in the ranges: yes
The BST built in arrival order has 35 levels; a balanced one has 14.

Top 10 by ratings: heap of size 10 1.750 ms, sorting everything 1.890 ms
  (same answer: yes)

People graph: 5841 people, 6997 co-credit edges, 3408 components,
  the largest has 605 people
  adjacency lists hold 13994 entries; an adjacency matrix would need
  34117281 cells; built and explored in 75.398 ms
  fewest hops from Stephen King to Charles Dickens: 5
    Stephen King -> Dave McKean -> Richard Dawkins ->
    Thomas Hardy -> Patricia Ingham -> Charles Dickens
```

Kode 16.31 `p1610_choose.py`: percobaan Subbab 16.8 dalam Python, dengan `bisect` untuk larik terurut, simpul buatan sendiri untuk *list* dan pohon, dan `dict` sebagai tabel hash.

```
# p1610_choose.py - one catalog, every structure of the book, measured.
import heapq
import sys
import time
from bisect import bisect_left, insort
from collections import deque

sys.setrecursionlimit(10000)
R = 5


class ListNode:                          # a singly linked list node (Chapter 7)
    __slots__ = ("key", "next")

    def __init__(self, key, nxt):
        self.key, self.next = key, nxt


class TreeNode:                          # a binary search tree node (Chapter 11)
    __slots__ = ("key", "left", "right")

    def __init__(self, key):
        self.key, self.left, self.right = key, None, None


class BST:
    """The unbalanced binary search tree of Chapter 11."""

    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
            return
        p = self.root
        while True:
            if key < p.key:
                if p.left is None:
                    p.left = TreeNode(key)
                    return
                p = p.left
            else:
                if p.right is None:
                    p.right = TreeNode(key)
                    return
```

Kode 16.31 (Lanjutan) *p1610_choose.py: percobaan Subbab 16.8 dalam Python, dengan bisect untuk larik terurut, simpul buatan sendiri untuk list dan pohon, dan dict sebagai tabel hash.*

```
                p = p.right

    def contains(self, key):
        p = self.root
        while p is not None and p.key != key:
            p = p.left if key < p.key else p.right
        return p is not None

    @staticmethod
    def count_range(p, lo, hi):
        if p is None:
            return 0
        if p.key < lo:                          # skip the left side
            return BST.count_range(p.right, lo, hi)
        if p.key >= hi:                         # skip the right side
            return BST.count_range(p.left, lo, hi)
        return (1 + BST.count_range(p.left, lo, hi)
                + BST.count_range(p.right, lo, hi))

    @staticmethod
    def height(p):
        return 0 if p is None else 1 + max(BST.height(p.left), BST.height(p.right))


def best_ms(f, reps=R):
    """Runs f several times and returns the best time in milliseconds."""
    best = float("inf")
    for _ in range(reps):
        t0 = time.perf_counter()
        f()
        best = min(best, (time.perf_counter() - t0) * 1000)
    return best


def best_ms_after(make, op, reps=R):
    """Like best_ms, but make() builds a fresh structure first; only op is timed."""
    best = float("inf")
    for _ in range(reps):
        c = make()
        t0 = time.perf_counter()
        op(c)
        best = min(best, (time.perf_counter() - t0) * 1000)
    return best
```

Kode 16.31 (Lanjutan) *p1610_choose.py: percobaan Subbab 16.8 dalam Python, dengan bisect untuk larik terurut, simpul buatan sendiri untuk list dan pohon, dan dict sebagai tabel hash.*

```
def main():
    books = []                          # (id, year, ratings, title, authors)
    with open("../data/catalog.tsv", encoding="utf-8") as f:
        next(f)
        for line in f:
            i, y, r, t, a = line.rstrip("\n").split("\t")
            books.append((int(i), int(y), int(r), t, a))
    n = len(books)

    # 1,000 identifiers that are present and 1,000 that are absent.
    present = {b[0] for b in books}
    queries = [books[i][0] for i in range(0, n, 10)]
    fresh = []
    for i in range(5, n, 10):
        q = books[i][0] + 1
        while q in present:
            q += 1
        fresh.append(q)                 # also used as the 1,000 new books
    queries += fresh
    ranges = [(i * 150000, i * 150000 + 50000) for i in range(200)]

    print(f"Catalog: {n} books; {len(queries)} lookups (half absent); "
          f"{len(ranges)} range queries; {len(fresh)} inserts\n")
    print(f"{'structure':<28}{'build ms':>10}{'lookup us':>12}{'range us':>12}"
          f"{'insert us':>12}")
    answers = []

    def row(name, b, look, rng, ins, found, cnt):
        print(f"{name:<28}{b:10.3f}{1000 * look / len(queries):12.3f}"
              f"{1000 * rng / len(ranges):12.3f}{1000 * ins / len(fresh):12.3f}")
        answers.extend([found, cnt])

    res = {}

    # unsorted array: append in arrival order, search by walking
    def build_unsorted():
        res["a"] = [b[0] for b in books]
```

Kode 16.31 (Lanjutan) *p1610_choose.py: percobaan Subbab 16.8 dalam Python, dengan bisect untuk larik terurut, simpul buatan sendiri untuk list dan pohon, dan dict sebagai tabel hash.*

```
    def look_unsorted():
        found = 0
        for q in queries:
            for x in res["a"]:
                if x == q:
                    found += 1
                    break
        res["found"] = found

    def range_unsorted():
        res["cnt"] = sum(1 for lo, hi in ranges for x in res["a"] if lo <= x < hi)

    tb, tl = best_ms(build_unsorted), best_ms(look_unsorted)
    tr = best_ms(range_unsorted)
    ti = best_ms_after(lambda: list(res["a"]),
                       lambda c: [c.append(q) for q in fresh])
    row("unsorted array", tb, tl, tr, ti, res["found"], res["cnt"])

    # sorted array: sort once, binary search (Chapter 2)
    def build_sorted():
        res["s"] = sorted(b[0] for b in books)

    def look_sorted():
        s, found = res["s"], 0
        for q in queries:
            i = bisect_left(s, q)
            found += i < len(s) and s[i] == q
        res["found"] = found

    def range_sorted():
        s = res["s"]
        res["cnt"] = sum(bisect_left(s, hi) - bisect_left(s, lo)
                         for lo, hi in ranges)

    tb, tl, tr = best_ms(build_sorted), best_ms(look_sorted), best_ms(range_sorted)
    ti = best_ms_after(lambda: list(res["s"]),
                       lambda c: [insort(c, q) for q in fresh])
    row("sorted array + binary search", tb, tl, tr, ti, res["found"], res["cnt"])

    # singly linked list: push at the front (Chapter 7)
    def build_list():
        head = None
        for b in books:
            head = ListNode(b[0], head)
        res["head"] = head
```

Kode 16.31 (Lanjutan) *p1610_choose.py: percobaan Subbab 16.8 dalam Python, dengan bisect untuk larik terurut, simpul buatan sendiri untuk list dan pohon, dan dict sebagai tabel hash.*

```
    def look_list():
        found = 0
        for q in queries:
            p = res["head"]
            while p is not None:
                if p.key == q:
                    found += 1
                    break
                p = p.next
        res["found"] = found

    def range_list():
        cnt = 0
        for lo, hi in ranges:
            p = res["head"]
            while p is not None:
                cnt += lo <= p.key < hi
                p = p.next
        res["cnt"] = cnt

    def push_fresh(box):
        for q in fresh:
            box[0] = ListNode(q, box[0])

    tb, tl, tr = best_ms(build_list), best_ms(look_list), best_ms(range_list)
    ti = best_ms_after(lambda: [res["head"]], push_fresh)
    row("singly linked list", tb, tl, tr, ti, res["found"], res["cnt"])

    # binary search tree of Chapter 11, keys inserted in arrival order
    def build_bst():
        t = BST()
        for b in books:
            t.insert(b[0])
        return t

    tb = best_ms(lambda: res.__setitem__("t", build_bst()))
    height = BST.height(res["t"].root)

    def look_bst():
        res["found"] = sum(1 for q in queries if res["t"].contains(q))

    def range_bst():
        res["cnt"] = sum(BST.count_range(res["t"].root, lo, hi)
                         for lo, hi in ranges)
```

Kode 16.31 (Lanjutan) *p1610_choose.py: percobaan Subbab 16.8 dalam Python, dengan bisect untuk larik terurut, simpul buatan sendiri untuk list dan pohon, dan dict sebagai tabel hash.*

```
    tl, tr = best_ms(look_bst), best_ms(range_bst)
    ti = best_ms_after(build_bst, lambda c: [c.insert(q) for q in fresh])
    row("BST (Chapter 11, unbalanced)", tb, tl, tr, ti, res["found"], res["cnt"])
    print(f"{'balanced BST':<28}  (none in the Python standard library)")

    # hash table (Chapter 15): dict
    def build_dict():
        res["h"] = {b[0]: b for b in books}

    def look_dict():
        h = res["h"]
        res["found"] = sum(1 for q in queries if q in h)

    def range_dict():                   # no order: every range scans it all
        h = res["h"]
        res["cnt"] = sum(1 for lo, hi in ranges for k in h if lo <= k < hi)

    def put_fresh(c):
        for q in fresh:
            c[q] = None

    tb, tl, tr = best_ms(build_dict), best_ms(look_dict), best_ms(range_dict)
    ti = best_ms_after(lambda: dict(res["h"]), put_fresh)
    row("hash table (dict)", tb, tl, tr, ti, res["found"], res["cnt"])

    agree = all(answers[i] == answers[i % 2] for i in range(2, len(answers)))
    print(f"All {len(answers) // 2} structures found {answers[0]} of the "
          f"{len(queries)} lookups and counted {answers[1]} books\nin the ranges: "
          f"{'yes' if agree else 'no'}")
    print(f"The BST built in arrival order has {height} levels; "
          "a balanced one has 14.")

    # ------------- the 10 most-rated books: heap against sorting -------------
    def top_heap():                     # min-heap of size 10 (Chapter 14)
        heap = []
        for b in books:
            if len(heap) < 10:
                heapq.heappush(heap, b[2])
            elif b[2] > heap[0]:
                heapq.heapreplace(heap, b[2])
        res["top1"] = sorted(heap, reverse=True)

    def top_sort():                     # sort all 10,000 (Chapter 3)
        res["top2"] = sorted((b[2] for b in books), reverse=True)[:10]
```

Kode 16.31 (Lanjutan) *p1610_choose.py: percobaan Subbab 16.8 dalam Python, dengan bisect untuk larik terurut, simpul buatan sendiri untuk list dan pohon, dan dict sebagai tabel hash.*

```
    th, ts = best_ms(top_heap), best_ms(top_sort)
    same = "yes" if res["top1"] == res["top2"] else "no"
    print(f"\nTop 10 by ratings: heap of size 10 {th:.3f} ms, sorting everything "
          f"{ts:.3f} ms\n  (same answer: {same})")

    # ------------- the graph of people credited on the same book -------------
    t0 = time.perf_counter()
    id_of, name, adj = {}, [], []       # name -> vertex number (hashing again)
    for b in books:
        who = []
        for a in b[4].split(","):
            a = a[1:] if a.startswith(" ") else a
            if a not in id_of:
                id_of[a] = len(name)
                name.append(a)
                adj.append([])
            if id_of[a] not in who:
                who.append(id_of[a])
        for u in who:
            for v in who:
                if u != v and v not in adj[u]:
                    adj[u].append(v)
    V = len(name)
    E = sum(len(a) for a in adj) // 2
    comp = [-1] * V                     # label every connected component (BFS)
    comps = largest = 0
    for s in range(V):
        if comp[s] != -1:
            continue
        size, q = 0, deque([s])
        comp[s] = comps
        while q:
            u = q.popleft()
            size += 1
            for v in adj[u]:
                if comp[v] == -1:
                    comp[v] = comps
                    q.append(v)
        largest = max(largest, size)
        comps += 1
    tg = (time.perf_counter() - t0) * 1000
    print(f"\nPeople graph: {V} people, {E} co-credit edges, {comps} components,\n"
          f"  the largest has {largest} people")
    print(f"  adjacency lists hold {2 * E} entries; an adjacency matrix "
          "would need\n"
          f"  {V * V} cells; built and explored in {tg:.3f} ms")
```

Kode 16.31 (Lanjutan) *p1610_choose.py: percobaan Subbab 16.8 dalam Python, dengan bisect untuk larik terurut, simpul buatan sendiri untuk list dan pohon, dan dict sebagai tabel hash.*

```
    s, t = id_of["Stephen King"], id_of["Charles Dickens"]
    parent = [-2] * V                   # BFS from s, remembering parents
    parent[s] = -1
    q = deque([s])
    while q:
        u = q.popleft()
        for v in adj[u]:
            if parent[v] == -2:
                parent[v] = u
                q.append(v)
    path, v = [], t
    while v != -1:
        path.append(name[v])
        v = parent[v]
    path.reverse()
    out = ("  fewest hops from Stephen King to Charles Dickens: "
           f"{len(path) - 1}\n   ")
    for i, p in enumerate(path):        # three names to a line
        sep = "" if i + 1 == len(path) else " ->\n   " if i % 3 == 2 else " ->"
        out += " " + p + sep
    print(out)


if __name__ == "__main__":
    main()
```

```
Catalog: 10000 books; 2000 lookups (half absent); 200 range queries; 1000 inserts

structure                    build ms   lookup us    range us   insert us
unsorted array                  0.124      70.476     190.958       0.020
sorted array + binary search    1.512       0.336       0.451       2.092
singly linked list              1.547     139.348     503.491       0.090
BST (Chapter 11, unbalanced)    6.350       0.556       3.624       0.698
balanced BST                 (none in the Python standard library)
hash table (dict)               0.512       0.034     255.484       0.085
All 5 structures found 1000 of the 2000 lookups and counted 4702 books
in the ranges: yes
The BST built in arrival order has 35 levels; a balanced one has 14.

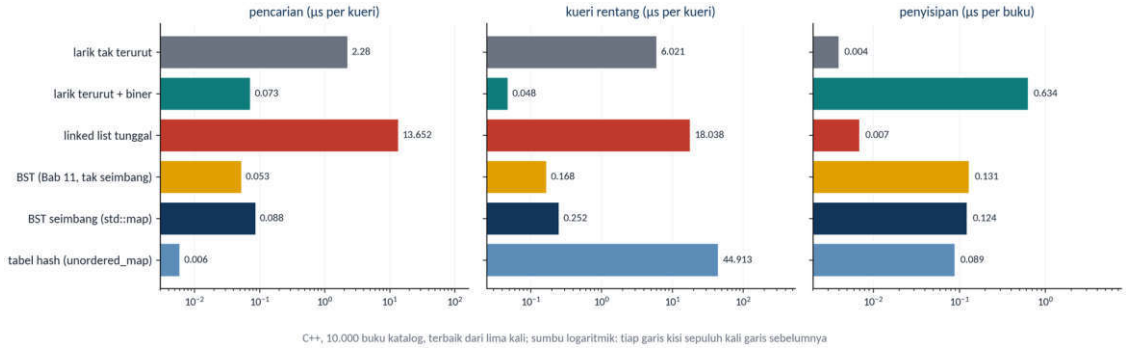
Top 10 by ratings: heap of size 10 0.265 ms, sorting everything 1.114 ms
  (same answer: yes)

People graph: 5841 people, 6997 co-credit edges, 3408 components,
  the largest has 605 people
  adjacency lists hold 13994 entries; an adjacency matrix would need
  34117281 cells; built and explored in 8.582 ms
  fewest hops from Stephen King to Charles Dickens: 5
    Stephen King -> Dave McKean -> Richard Dawkins ->
    Thomas Hardy -> Patricia Ingham -> Charles Dickens
```

Waktu Anda akan berbeda dari angka-angka ini: bergantung pada prosesor, kompilator, program lain yang sedang berjalan, dan, di Java, sejauh mana kompilator *just-in-time* sudah “panas”. *Rasio* antarbaris, dan jawabannya, tidak akan berbeda.

16.8.3 Membaca angka-angkanya

Gambar 16.16 memetakan pengukuran C++ pada sumbu logaritmik, tempat setiap garis kisi sepuluh kali garis sebelumnya. **Tabel 16.5** menjajarkan hasil pencarian dan rentang dari ketiga bahasa.



Gambar 16.16 Enam struktur pada katalog Pustaka Ceria (C++, mikrodetik per operasi, terbaik dari lima kali jalan, sumbu logaritmik). Tidak ada struktur yang unggul di setiap kolom.

Tabel 16.5 Mikrodetik per pencarian dan per kueri rentang pada katalog 10.000 buku, dari ketiga jalannya program di atas (angka ditulis seperti dicetak program, dengan titik desimal). Struktur terbaik untuk pencarian termasuk yang terburuk untuk rentang.

Struktur	Cari C++	Cari Java	Cari Python	Rentang C++	Rentang Java	Rentang Python
larik tak terurut	2.280	1.546	70.476	6.021	9.619	190.958
larik terurut + pencarian biner	0.073	0.206	0.336	0.048	0.182	0.451
singly linked list	13.652	19.141	139.348	18.038	29.328	503.491
BST (Bab 11, tak seimbang)	0.053	0.158	0.556	0.168	0.333	3.624
BST seimbang (pustaka)	0.088	0.269	—	0.252	1.763	—
tabel hash (pustaka)	0.006	0.169	0.034	44.913	113.180	255.484

Lima pelajaran dapat dibaca dari tabel dan gambar itu, dan masing-masing menegaskan satu baris **Tabel 16.4**.

1. Untuk pencarian menurut kunci, tabel hash menang telak. Di C++ ia memerlukan 0,006 mikrodetik per pencarian, dua belas kali lebih cepat daripada pencarian biner dan 380 kali lebih cepat daripada pencarian sekuensial. Itulah Persamaan (16.10) yang sedang bekerja: satu hash, satu rantai pendek, berapa pun n. Di Python selisihnya dengan larik terurut lebih kecil, sekitar sepuluh kali, karena setiap langkah Python lebih mahal dan pencarian didominasi oleh interpreter; namun `dict` tetap pencarian tercepat yang dimiliki Python.

2. Untuk rentang, tabel hash adalah struktur terburuk dari semuanya. Tabel hash tidak menyimpan urutan, sehingga "setiap identitas antara a dan $a + 50{,}000$" memaksanya memeriksa kesemua 10.000 kunci untuk setiap kueri: 45 mikrodetik, dibandingkan 0,048 untuk larik terurut, yang cukup memerlukan dua pencarian biner dan satu pengurangan. Inilah tawar-menawar Subbab 16.4.4 dalam angka: struktur yang paling cepat menjawab "adakah ia?" sama sekali tidak dapat menjawab "apa yang di dekatnya?".

3. *Linked list* paling lambat dalam hampir semua hal, kecuali menyisipkan di depan. Pencariannya mengunjungi butir sebanyak larik tak terurut, tetapi sekitar enam kali lebih lambat di C++, karena butir-butir larik bersebelahan di memori, sedangkan simpul-simpul *list* tersebar dan setiap langkah ke simpul berikutnya berpotensi meleset dari *cache* (Bab 7). Satu-satunya kemenangannya adalah penyisipan di depan, 0,007 mikrodetik, seperti dijanjikan **Tabel 16.4**.

4. BST tak seimbang bernasib baik di sini, dan keberuntungan itu layak dipahami. Pencariannya secepat pencarian biner, dan kueri rentangnya cepat karena pohon memangkas seluruh subpohon. Ia membangun pohon 35 tingkat, padahal pohon yang seimbang sempurna

hanya 14, karena identitas-identitas tiba menurut urutan popularitas, yang nyaris acak terhadap identitas. Berikan kunci yang sama dalam urutan *terurut*, maka setiap simpul hanya mendapat anak kanan: 10.000 tingkat, pencarian selambat *linked list* (Bab 11). Pohon seimbang dari pustaka sedikit lebih lambat pada masukan yang ramah ini dan kebal terhadap masukan yang tidak ramah; asuransi itulah yang menjadikannya pilihan bawaan bila Anda memerlukan urutan dan perubahan.

5. Larik terurut unggul sampai datanya berubah. Ia memiliki kueri rentang terbaik dan pencarian terbaik kedua, tanpa pointer sama sekali. Namun setiap penyisipan menggeser rata-rata separuh larik: 0,634 mikrodetik per buku, penyisipan paling lambat dalam tabel C++, dan selisihnya tumbuh linear terhadap n. Untuk katalog yang dibangun ulang sekali setiap malam ia ideal; untuk katalog yang berubah sepanjang hari, pohon seimbang adalah kompromi yang lebih baik.

***Heap*.** Sepuluh buku dengan penilaian terbanyak memerlukan 0,006 ms dengan *heap* berukuran 10 dan 0,420 ms dengan pengurutan penuh di C++: tujuh puluh kali lebih sedikit, seperti diramalkan Persamaan (16.7) untuk $\lg 10 \approx 3.3$ lawan $\lg 10{,}000 \approx 13.3$ ditambah ongkos menyalin dan mengurutkan semuanya. Di Python *heap* sekitar empat kali lebih cepat. Pada jalannya Java keduanya berdekatan, karena gelung *heap* hanya berjalan lima kali, terlalu sedikit bagi kompilator *just-in-time* untuk mengoptimasinya; mengukur waktu kode Java dengan andal memerlukan fase pemanasan, yang disediakan alat *benchmark* produksi seperti JMH.

Graf. Kolom orang yang dicantumkan di katalog memuat 5.841 orang berbeda (pengarang, ilustrator, penyunting, penerjemah); menghubungkan dua orang setiap kali mereka dicantumkan pada buku yang sama menghasilkan 6.997 sisi. Graf yang serenggang itu adalah tepat untuk apa *adjacency list* diciptakan: *list*-nya memuat $2|E| = 13{,}994$ entri, sedangkan matriks ketetanggaan akan memerlukan

$$|V|^2 = 5841^2 = 34{,}117{,}281 \text{ sel} \qquad \text{dibandingkan} \qquad |V| + 2|E| = 19{,}835 \text{ entri}, \tag{16.12}$$

lebih dari 1.700 kali memorinya, hampir seluruhnya nol. Menurut Persamaan (16.12), bahkan dengan satu bit per sel, matriksnya memakan sekitar 4 megabita untuk mencatat kurang dari tujuh ribu hubungan. BFS dari setiap orang yang belum dikunjungi memberi label 3.408 komponen terhubung; yang terbesar menyatukan 605 orang, dari Stephen King sampai Charles Dickens, yang terpisah lima lompatan di sepanjang rantai yang dicetak program. Seluruh graf, termasuk membangun *hash map* dari nama ke nomor simpul, memerlukan kurang dari 4 ms di C++. Satu tugas memakai tiga struktur buku ini sekaligus: tabel hash untuk mengubah nama menjadi nomor simpul, *adjacency list* untuk menyimpan graf, dan *queue* untuk menelusurinya.

16.8.4 Pilihannya, dalam satu paragraf

Namai operasi dominannya. Jika operasinya "butir ke-i", pakai larik. Jika "cari menurut kunci" dan tidak lebih, pakai tabel hash. Jika "cari menurut kunci" ditambah apa pun tentang urutan, pakai pohon pencarian seimbang, atau larik terurut bila datanya jarang berubah. Jika "yang terbesar (atau terkecil) berikutnya", pakai *heap*. Jika butir-butirnya saling terhubung dan pertanyaannya tentang hubungan-hubungan itu, pakai graf dengan *adjacency list* dan penelusuran yang sesuai dengan pertanyaannya: BFS untuk langkah paling sedikit, DFS atau Kahn untuk struktur dan urutan, Dijkstra dengan *heap* untuk rute berbobot. Pakai *stack* dan *queue* bila urutan pemrosesannya terakhir-masuk-pertama-keluar atau pertama-masuk-pertama-keluar, dan *linked*

list bila Anda memegang sebuah posisi dan harus menyisipkan atau menghapus di sana. Kemudian, seperti yang dilakukan subbab ini, ukurlah; dan bila hasil pengukuran bertentangan dengan tabel, cari tahu sebabnya sebelum memercayai salah satunya.

16.9 Perbandingan Soal-Soal dalam Bab Ini

Tabel 16.6 menjajarkan kesembilan soal. Seperti di Bab 8, bacalah kolom terakhir dari atas ke bawah: sebagian besar penyelesaian keliru yang menggoda lolos pada contoh yang menjadi alasan kode itu dibuat. "Jarak" DFS benar untuk empat dari tujuh cabang yang terjangkau; Kahn tanpa pemeriksaan penutup sempurna pada setiap masukan tanpa siklus; `getMin` dari elemen terakhir benar pada *heap* yang diberikan; hash yang dihitung dalam `int` bekerja untuk kunci pendek; dan penghapusan dengan mengosongkan slot bekerja sampai lintasan *probe* suatu kunci melewati slot yang dikosongkan itu.

Tabel 16.6 Kesembilan soal bab ini: struktur yang dipakai masing-masing, ongkosnya, dan masukan tempat penyelesaian kelirunya gagal.

Soal	Struktur	Waktu	Memori tambahan	Masukan yang digagalkan penyelesaian keliru
16.1	*max-heap* larik	$O(\lg n)$ ambil dan naikkan; $O(n)$ bangun, `getMin`, periksa	$O(1)$	*heap* yang elemen terakhirnya bukan minimum; larik berisi 0 atau 1 elemen
16.2	dua *hash map*	rata-rata $O(n)$	$O(n)$	setiap tumpukan yang tiket pertamanya bukan potongan pertama
16.3	*adjacency list* + *queue*	$O(\|V\| + \|E\|)$	$O(\|V\|)$	setiap graf tempat DFS mencapai sebuah simpul lewat jalan jauh lebih dahulu
16.4	*adjacency list* + *queue*	$O(\|V\| + \|E\|)$	$O(\|V\|)$	graf dengan siklus berarah
16.5	*adjacency list* + *min-heap*	$O((\|V\| + \|E\|)\lg\|V\|)$	$O(\|V\| + \|E\|)$	jalan langsung yang lebih lama daripada jalan memutar
16.6	*min-heap* berukuran k	$O(n \lg k)$	$O(k)$	masukan apa pun, bila *heap*-nya *max-heap*
16.7	*max-heap* + *min-heap*	$O(\lg n)$ per nilai	$O(n)$	nilai keempat contohnya, tanpa penyeimbangan
16.8	tabel hash berantai	rata-rata $O(1 + \alpha)$	$O(n + m)$	kunci yang cukup panjang untuk meluapkan `int`
16.9	*linear probing*	rata-rata $O(1/(1-\alpha)^2)$	$O(m)$	pencarian yang lintasannya melewati slot terhapus

Ongkos-ongkos itu terkelompok menjadi tiga keluarga. **Satu lintasan menuruni pohon** berongkos $O(\lg n)$, menurut Persamaan (16.2): setiap operasi *heap*, setiap pencarian di pohon seimbang. **Satu kali lewat atas sebuah struktur** berongkos $O(n)$ atau $O(|V| + |E|)$: membangun *heap*, BFS, Kahn, penelusuran rencana perjalanan, mencari minimum *max-heap*. **Satu lompatan yang dihitung** berongkos rata-rata $O(1)$: setiap operasi tabel hash pada *load factor* yang sedang, menurut Persamaan (16.10) dan Persamaan (16.11). Melihat keluarga mana yang memuat sebuah operasi adalah sebagian besar dari analisisnya.

16.10 Profitina dalam Praktik

Profitina adalah platform visibilitas AI (GEO), berjalan di produksi di app.profitina.com, yang mengukur seberapa siap situs web sebuah merek untuk dikutip oleh mesin jawaban AI dan membantu memperbaiki temuannya. Ketiga struktur dalam bab ini memiliki tempat yang jelas

dalam produk semacam itu; kita melukiskan bentuk setiap persoalan dan kelas penyelesaian yang cocok dengannya, bukan cara Profitina sendiri dibangun.

Situs web adalah graf. Ketika sebuah situs dipindai, halaman-halamannya adalah simpul dan tautan internalnya adalah busur. Pertanyaan-pertanyaan pemindaian kesiapan adalah pertanyaan graf. Halaman mana yang sama sekali dapat dicapai dari beranda? Itu penelusuran dari satu simpul, dan halaman yang tidak dicapai penelusuran mana pun adalah halaman yatim yang kecil kemungkinannya ditemukan perayap mesin jawaban. Berapa klik jarak sebuah halaman penting dari beranda? Itu jarak BFS, persis Soal 16.3, dan halaman yang terkubur banyak klik dalamnya cenderung jarang dirayapi. Perayapan itu sendiri adalah BFS yang *queue*-nya berupa *frontier*, halaman yang sudah ditemukan tetapi belum diambil, disertai himpunan "sudah terlihat" agar tidak ada halaman yang diambil dua kali.

Himpunan "sudah terlihat" adalah tabel hash, dan kuncinya perlu dirawat. Halaman yang sama dapat bersembunyi di balik banyak ejaan: `https://Example.com/a`, `https://example.com/a/`, `https://example.com/a?utm_source=x`. Sebelum sebuah URL masuk ke himpunan itu, ia *dikanonikkan*, diubah menjadi satu bentuk baku (nama host huruf kecil, aturan garis miring penutup, parameter pelacakan dibuang), barulah di-hash; bila tidak, himpunan pada Soal 16.8 akan menyimpan tiga entri untuk satu halaman dan perayapan mengambilnya tiga kali.

Pemindaian ulang adalah *priority queue*. Profitina memindai ulang situs setiap merek secara otomatis dengan irama yang bergantung pada tingkat langganan, dan membunyikan peringatan ketika skor berubah. Penjadwal semacam itu menyimpan, untuk setiap situs, waktu jatuh tempo pemindaian berikutnya, dan berulang kali bertanya "pemindaian mana yang paling segera jatuh tempo?": *min-heap* berkunci waktu jatuh tempo, struktur Soal 16.5 dan 16.6. Ketika pengguna meminta pemindaian ulang seketika, waktu jatuh tempo situs itu turun: `decreaseKey` pada Soal Latihan 16.1, atau, lebih sederhana, entri baru dengan penghapusan malas seperti pada Soal 16.5.

Daftar teratas adalah *heap* berukuran k. Papan peringkat komunitas dan daftar "unggahan teratas minggu ini" memerlukan k yang terbaik dari banyak butir, dan memerlukannya sering; *heap* berukuran k (Soal 16.6) menjawabnya tanpa mengurutkan semua orang.

Server Profitina ditulis dalam Python, sehingga struktur-struktur ini muncul sebagai `dict`, `set`, `collections.deque`, dan `heapq`, baris-baris **Tabel 16.3**, dan seperti bagian produk lainnya semuanya dijaga oleh rangkaian pengujian otomatis lengkap, jauh lebih dari seribu pengujian, yang harus lulus sebelum setiap rilis. Versi latihan yang dapat dijalankan dari gagasan bab ini adalah Profitina Laundry, studi kasus seri buku ini: Soal Latihan 16.14 membangun meja layanannya dari *hash map* pesanan dan *priority queue* waktu janji.

16.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Memakai ujung *priority queue* yang salah. `std::priority_queue` adalah *max-heap*; `PriorityQueue` dan `heapq` adalah *min-heap*. Kode yang disalin antarbahasa diam-diam terbalik (Soal 16.6). Tuliskan arahnya dalam komentar di tempat *heap* dideklarasikan.

Memelihara *max-heap* untuk mencari k terbesar. Untuk menyimpan k yang terbesar Anda harus membuang yang terkecil di antara mereka, sehingga *heap*-nya harus *min-heap* (Soal 16.6).

Lupa menyimpan akar sebelum menimpanya. `extractMax` harus menyalin `heap[0]` ke variabel lokal sebelum daun terakhir masuk (Soal 16.1).

Memperbaiki ke arah yang salah. Kunci yang lebih besar hanya dapat naik; kunci yang lebih kecil hanya dapat tenggelam. `increaseKey` melakukan *sift-up*, `decreaseKey` melakukan *sift-down* (Soal 16.1, Soal Latihan 16.1).

Aritmetika *unsigned* yang *underflow*. `arr.size() - 2` bernilai sangat besar bila larik berisi kurang dari dua elemen. Bergelunglah atas anak-anak mulai indeks 1, atau bandingkan sebelum mengurangi (Soal 16.1).

Mencari minimum *max-heap* di satu tempat. Minimum itu berada di suatu tempat di antara $\lceil n/2 \rceil$ daun, tidak pernah di indeks tetap (Soal 16.1).

Menyebut `std::map` sebagai *hash map*. Ia adalah pohon pencarian seimbang dengan operasi $O(\lg n)$ dan iterasi terurut; tabel hash-nya adalah `std::unordered_map` (Soal 16.2).

Mengandalkan urutan iterasi tabel hash. Urutan itu tidak ditentukan dan berubah setelah *rehash*. Bergelunglah atas masukan, atau urutkan, bila urutan keluaran penting (Soal 16.2).

Memakai kedalaman DFS sebagai jarak. Hanya BFS, dengan *queue*-nya, yang menemukan rute dengan sisi paling sedikit (Soal 16.3).

Menandai simpul ketika keluar dari *queue* alih-alih ketika masuk. Jaraknya tetap benar, tetapi sebuah simpul dapat masuk *queue* berkali-kali; tandai saat masuk.

Memercayai pengurutan topologis tanpa menghitung. Bila simpul yang dikeluarkan kurang dari $|V|$, ada siklus (Soal 16.4).

Mengunci jarak Dijkstra pada penglihatan pertama. Sebuah jarak baru final ketika simpulnya keluar dari *heap* sebagai yang terkecil (Soal 16.5). Dan Dijkstra mensyaratkan bobot yang tidak negatif.

Lupa menyeimbangkan dua *heap*. Median adalah pernyataan tentang posisi; ukuran keduanya harus tetap berselisih paling banyak satu (Soal 16.7).

Menghitung hash dalam `int` *signed*. Nilainya meluap (perilaku tak terdefinisi di C++) dan dapat menghasilkan indeks negatif. Pakailah aritmetika *unsigned* dan reduksi modulo m di setiap langkah, atau `Math.floorMod` di Java (Soal 16.8).

Menghapus dari tabel *open addressing* dengan mengosongkan slot. Pakailah *tombstone* (Soal 16.9).

Membiarkan tabel hash penuh. Ongkosnya tumbuh seperti $1 + \alpha$ untuk *chaining* dan seperti $1/(1-\alpha)^2$ untuk *linear probing*; lakukan *rehash* sebelum α membesar (Persamaan (16.10), Persamaan (16.11)).

Memulai rantai dari tempat yang salah. Butir pertama masukan bukanlah butir pertama jawaban (Soal 16.2).

Memilih struktur sebelum menamai operasinya. Namai operasi dominannya lebih dahulu, lalu lihat **Tabel 16.4**, lalu ukur (Subbab 16.8).

16.12 Ringkasan Bab

- Soal tentang struktur data diselesaikan dengan sembilan langkah (Subbab 16.2): nyatakan ulang, pilih struktur beserta alasannya, gambarlah, tulis invariannya, daftarkan kasus tepi, atur urutan perubahan dengan aman, hitung ongkosnya, tentukan perlakuan terhadap masukan buruk, dan uji dengan pemeriksa memori serta kembaran yang lambat.

- Setiap struktur menjanjikan sebuah invarian, dan setiap operasi harus membiarkan janji itu tetap benar; sebagian besar galat dalam bab ini adalah operasi yang tidak melakukannya.
- *Heap* (Soal 16.1, 16.6, 16.7): anak-anak indeks i adalah $2i+1$ dan $2i+2$, Persamaan (16.1); tingginya $\lfloor \lg n \rfloor$, Persamaan (16.2), sehingga sisip, ambil, dan naikkan berongkos $O(\lg n)$; minimum *max-heap* berada di antara daun, Persamaan (16.3); membangun dari bawah berongkos $O(n)$, Persamaan (16.4); k terbesar memerlukan *min-heap* berukuran k, Persamaan (16.7), dengan sekitar $k\ln(n/k)$ penggantian pada masukan acak, Persamaan (16.8); median berjalan memerlukan *max-heap* dan *min-heap* yang ukurannya seimbang.
- Tabel hash (Soal 16.2, 16.8, 16.9): *load factor* $\alpha = n/m$, Persamaan (16.9), menentukan ongkosnya, $\Theta(1+\alpha)$ untuk *chaining*, Persamaan (16.10), dan sekitar $\frac{1}{2}(1 + 1/(1-\alpha)^2)$ *probe* untuk pencarian gagal dengan *linear probing*, Persamaan (16.11); hitung hash tanpa *overflow*; hapus dengan *tombstone* pada *open addressing*; dan ingat bahwa `std::map` adalah pohon, bukan tabel hash.
- Graf (Soal 16.3, 16.4, 16.5): BFS menemukan rute dengan sisi paling sedikit dalam $O(|V| + |E|)$, Persamaan (16.5); algoritma Kahn menemukan urutan topologis atau membuktikan adanya siklus; Dijkstra dengan *heap* menemukan rute tercepat dengan bobot tidak negatif dalam $O((|V| + |E|)\lg|V|)$, Persamaan (16.6); graf renggang tempatnya di *adjacency list*, Persamaan (16.12).
- Memilih struktur (Subbab 16.8): namai operasi dominannya dan lihat **Tabel 16.4**. Pada katalog nyata, tabel hash paling cepat untuk pencarian dan paling lambat untuk rentang, larik terurut terbaik untuk rentang dan paling lambat untuk disisipi, *linked list* paling lambat dalam hampir segala hal kecuali penyisipan di depan, dan *heap* tujuh puluh kali lebih cepat daripada pengurutan untuk sepuluh teratas.

16.13 Soal Latihan

Soal Latihan 16.1 sampai 16.6 dan 16.14 meminta kode; penyelesaiannya adalah program-program di Subbab 16.14. Yang lain meminta penalaran dan dijawab dengan kata-kata.

1. **(Menurunkan kunci.)** Tulislah `decreaseKey(i, newValue)` untuk *max-heap* Soal 16.1. Ke arah mana perbaikannya berjalan, dan mengapa? Terapkan `decreaseKey(0, 2)` pada `[8, 5, 6, 4, 5, 3, 2, 1, 3, 1]`.
2. **(Menggabungkan k list terurut.)** Gabungkan *list* terurut {3, 9, 14}, {1, 10}, {2, 4, 8, 20}, dan satu *list* kosong menjadi satu *list* terurut dengan *min-heap* yang memuat paling banyak satu entri per *list*. Berapa ongkosnya untuk k *list* dengan N nilai seluruhnya?
3. **(Dua sisi.)** Graf disebut *bipartit* bila simpul-simpulnya dapat diwarnai dengan dua warna sehingga setiap sisi menghubungkan warna yang berbeda. Putuskan hal itu dengan BFS. Apakah peta cabang Soal 16.3 bipartit? Apakah persegi (empat simpul dalam satu siklus) bipartit?
4. **(Komponen.)** Hitung komponen terhubung sebuah graf dengan DFS iteratif yang memakai *stack* eksplisit. Berapa komponen yang dimiliki peta cabang?
5. **(Karakter unik pertama.)** Temukan karakter pertama dalam sebuah string yang muncul tepat sekali, dalam dua kali lewat dengan *hash map*. Cobalah "borrowed books" dan "aabb".
6. **(Dua harga.)** Diberikan harga 35, 80, 42, 17, 60, 23 dan nilai sebuah voucer, temukan dua harga yang jumlahnya sama dengan nilai itu dalam satu kali lewat dengan *hash set*. Cobalah 77 dan 100. Mengapa sebuah harga tidak boleh dipasangkan dengan dirinya sendiri?
7. **(Heap dengan tangan.)** Sisipkan 20, 5, 15, 22, 40, 11, 30 satu per satu ke *max-heap* kosong. Berikan larik setelah setiap penyisipan, lalu larik setelah satu `extractMax`.

8. **(Membangun dengan tangan.)** Terapkan `buildMaxHeap` pada {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}. Daftarkan pemanggilan `heapifyDown` secara berurutan beserta larik setelah masing-masing.
9. **(BFS dan DFS dengan tangan.)** Untuk graf berarah dengan busur 0→1, 0→2, 1→3, 2→3, 3→4, 2→4, daftarkan urutan BFS dan urutan DFS rekursif dari 0 (tetangga dalam urutan naik), serta jarak BFS setiap simpul.
10. **(Dijkstra dan jalan negatif.)** Pada graf dengan busur A→B 2, A→C 5, C→B -4, berapa jarak ke B yang dilaporkan algoritma Dijkstra, dan berapa jarak terpendek yang sebenarnya? Langkah invarian mana yang gagal?
11. **(Probing dengan tangan.)** Dengan $m = 7$, $h(k) = k \bmod 7$, dan *linear probing*, sisipkan 12, 19, 5, 26, 33. Kemudian hapus 19 dengan *tombstone* dan cari 26 serta 40. Berikan setiap barisan *probe*.
12. **(Struktur yang mana?)** Sebutkan struktur dari **Tabel 16.4** untuk setiap tugas Pustaka Ceria berikut, dengan alasan satu kalimat: (a) data anggota, dicari menurut nomor keanggotaan di meja layanan sepanjang hari; (b) daftar pinjaman yang terlambat, selalu diproses mulai dari yang paling lama terlambat; (c) katalog rak, dicetak menurut nomor panggil setiap malam dan berubah beberapa kali seminggu; (d) tautan "peminjam buku ini juga meminjam" antarbuku; (e) riwayat *undo* penyunting katalog; (f) buku-buku yang kodenya berada di antara B2000 dan B2999.
13. **(Matriks atau list?)** Sebuah kota memiliki 20.000 persimpangan dan 52.000 ruas jalan. Berapa sel yang dimiliki matriks ketetanggaan, berapa entri yang dimiliki *adjacency list*, dan berapa bita yang diperlukan masing-masing dengan bilangan bulat 4 bita (dan, untuk matriks, dengan satu bit per sel)? Mana yang akan Anda pilih untuk BFS?
14. **(Profitina Laundry.)** Meja layanan Profitina Laundry menerima pesanan, masing-masing dengan nomor pesanan, nama pelanggan, dan jam janji. Mesin cuci selalu mengambil pesanan dengan jam janji paling awal (bila sama, menurut nomor pesanan), dan meja layanan harus menjawab "bagaimana status pesanan x?" seketika. Pilih kedua strukturnya, lalu tulis `receive`, `washNext`, dan `status`. Jalankan: terima 501 (Ani, 17.00), 502 (Budi, 10.00), 503 (Citra, 12.00); cuci berikutnya; terima 504 (Dewi, 09.00, pesanan kilat); cuci berikutnya; tampilkan status 503, 502, dan 999.

16.14 Penyelesaian Soal Latihan

Soal Latihan 16.1 sampai 16.6 dan 16.14. **Kode 16.32**, **Kode 16.33**, dan **Kode 16.34** memuat semua fungsi yang diminta dan mencetak baris-baris yang sama.

Kode 16.32 `ex16_solutions.cpp`: penyelesaian soal-soal latihan pemrograman Bab 16.

```cpp
// ex16_solutions.cpp - solutions to the programming exercises of Chapter 16.
#include <functional>
#include <iostream>
#include <queue>
#include <stdexcept>
#include <string>
#include <tuple>
#include <unordered_map>
#include <unordered_set>
#include <vector>
using namespace std;

void print(const string& label, const vector<int>& v) {
    cout << label;
    for (size_t i = 0; i < v.size(); i++) cout << (i ? " " : "") << v[i];
    cout << "\n";
}

// ---------- Exercise 16.1: decreaseKey in a max-heap (a smaller key sinks) ----
void siftDown(vector<int>& h, size_t i) {
    while (true) {
        size_t l = 2 * i + 1, r = 2 * i + 2, big = i;
        if (l < h.size() && h[l] > h[big]) big = l;
        if (r < h.size() && h[r] > h[big]) big = r;
        if (big == i) return;
        swap(h[i], h[big]);
        i = big;
    }
}

void decreaseKey(vector<int>& h, size_t i, int newValue) {
    if (i >= h.size()) throw out_of_range("decreaseKey: no element at that index");
    if (newValue > h[i]) throw invalid_argument("decreaseKey: new value is larger");
    h[i] = newValue;
    siftDown(h, i);
}
```

Kode 16.32 (Lanjutan) *ex16_solutions.cpp: penyelesaian soal-soal latihan pemrograman Bab 16.*

```
// ---------- Exercise 16.2: merge k sorted lists with a min-heap ----------
vector<int> mergeSorted(const vector<vector<int>>& lists) {
    using E = tuple<int, size_t, size_t>;          // (value, which list, position)
    priority_queue<E, vector<E>, greater<E>> heap;
    for (size_t j = 0; j < lists.size(); j++)
        if (!lists[j].empty()) heap.push({lists[j][0], j, 0});
    vector<int> out;
    while (!heap.empty()) {
        auto [v, j, p] = heap.top();
        heap.pop();
        out.push_back(v);
        if (p + 1 < lists[j].size()) heap.push({lists[j][p + 1], j, p + 1});
    }
    return out;
}

// ---------- Exercise 16.3: is the graph bipartite? (BFS two-coloring) ----------
bool bipartite(const vector<vector<int>>& adj) {
    vector<int> color(adj.size(), -1);
    for (size_t s = 0; s < adj.size(); s++) {
        if (color[s] != -1) continue;
        color[s] = 0;
        queue<int> q;
        q.push(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int v : adj[u]) {
                if (color[v] == -1) {
                    color[v] = 1 - color[u];       // the other side
                    q.push(v);
                } else if (color[v] == color[u]) {
                    return false;                  // an edge inside one side
                }
            }
        }
    }
    return true;
}
```

Kode 16.32 (Lanjutan) *ex16_solutions.cpp: penyelesaian soal-soal latihan pemrograman Bab 16.*

```
// ---------- Exercise 16.4: count connected components (DFS, a stack) ----------
int components(const vector<vector<int>>& adj) {
    vector<bool> seen(adj.size(), false);
    int count = 0;
    for (size_t s = 0; s < adj.size(); s++) {
        if (seen[s]) continue;
        count++;
        vector<int> stack = {(int)s};
        seen[s] = true;
        while (!stack.empty()) {
            int u = stack.back();
            stack.pop_back();
            for (int v : adj[u])
                if (!seen[v]) {
                    seen[v] = true;
                    stack.push_back(v);
                }
        }
    }
    return count;
}

// ---------- Exercise 16.5: the first character that occurs only once ----------
char firstUnique(const string& s) {
    unordered_map<char, int> count;
    for (char c : s) count[c]++;                  // pass 1: count
    for (char c : s)                              // pass 2: in the original order
        if (count[c] == 1) return c;
    return '\0';
}

// ---------- Exercise 16.6: two prices that add up to a voucher ----------
bool twoSum(const vector<int>& price, int target, int& a, int& b) {
    unordered_set<int> seen;
    for (int p : price) {
        if (seen.count(target - p)) {             // its partner came earlier
            a = target - p;
            b = p;
            return true;
        }
        seen.insert(p);
    }
    return false;
}
```

Kode 16.32 (Lanjutan) *ex16_solutions.cpp: penyelesaian soal-soal latihan pemrograman Bab 16.*

```
// ---------- Exercise 16.14: the Profitina Laundry service desk ----------
struct Order {
    int number;
    string customer;
    int readyBy;                                    // promised hour, 0..23
    string status;
};

class LaundryDesk {
    unordered_map<int, Order> byNumber;             // look up any order in O(1)
    using Key = pair<int, int>;                     // (promised hour, order number)
    priority_queue<Key, vector<Key>, greater<Key>> next;   // earliest promise first
public:
    void receive(int number, const string& customer, int readyBy) {
        byNumber[number] = {number, customer, readyBy, "waiting"};
        next.push({readyBy, number});
    }
    void washNext() {
        if (next.empty()) {
            cout << "  nothing to wash\n";
            return;
        }
        auto [hour, number] = next.top();
        next.pop();
        byNumber[number].status = "washed";
        cout << "  wash order " << number << " (" << byNumber[number].customer
             << ", ready by " << hour << ":00)\n";
    }
    void status(int number) const {
        auto it = byNumber.find(number);
        if (it == byNumber.end()) cout << "  order " << number << ": unknown\n";
        else cout << "  order " << number << ": " << it->second.status << "\n";
    }
};

int main() {
    cout << "Exercise 16.1\n";
    vector<int> h = {8, 5, 6, 4, 5, 3, 2, 1, 3, 1};
    decreaseKey(h, 0, 2);
    print("  after decreaseKey(0, 2): ", h);

    cout << "Exercise 16.2\n";
    print("  merged: ", mergeSorted({{3, 9, 14}, {1, 10}, {2, 4, 8, 20}, {}}));
```

Kode 16.32 (Lanjutan) *ex16_solutions.cpp: penyelesaian soal-soal latihan pemrograman Bab 16.*

```
    cout << "Exercise 16.3\n";
    vector<vector<int>> branches = {{1, 2, 3}, {0, 6, 4}, {0, 5}, {0, 6},
                                    {1, 5}, {2, 4}, {1, 3}, {}};
    vector<vector<int>> square = {{1, 3}, {0, 2}, {1, 3}, {2, 0}};
    cout << boolalpha << "  branch map bipartite: " << bipartite(branches)
         << "\n  a square bipartite: " << bipartite(square) << "\n";

    cout << "Exercise 16.4\n";
    cout << "  components of the branch map: " << components(branches) << "\n";

    cout << "Exercise 16.5\n";
    for (string s : {"borrowed books", "aabb"}) {
        char c = firstUnique(s);
        cout << "  \"" << s << "\" -> " << (c ? string(1, c) : string("none"))
             << "\n";
    }

    cout << "Exercise 16.6\n";
    int a, b;
    vector<int> price = {35, 80, 42, 17, 60, 23};
    for (int target : {77, 100}) {
        if (twoSum(price, target, a, b))
            cout << "  " << target << " = " << a << " + " << b << "\n";
        else
            cout << "  " << target << ": no pair\n";
    }

    cout << "Exercise 16.14\n";
    LaundryDesk desk;
    desk.receive(501, "Ani", 17);
    desk.receive(502, "Budi", 10);
    desk.receive(503, "Citra", 12);
    desk.washNext();
    desk.receive(504, "Dewi", 9);                  // an express order
    desk.washNext();
    desk.status(503);
    desk.status(502);
    desk.status(999);
    return 0;
}
```

Kode 16.33 `Ex16Solutions.java`: penyelesaian yang sama dalam Java.

```java
// Ex16Solutions.java - solutions to the programming exercises of Chapter 16.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;

public class Ex16Solutions {

    static String join(List<Integer> v) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < v.size(); i++)
            sb.append(i > 0 ? " " : "").append(v.get(i));
        return sb.toString();
    }

    // ---------- Exercise 16.1: decreaseKey in a max-heap (a smaller key sinks) --
    static void siftDown(List<Integer> h, int i) {
        while (true) {
            int l = 2 * i + 1, r = 2 * i + 2, big = i;
            if (l < h.size() && h.get(l) > h.get(big)) big = l;
            if (r < h.size() && h.get(r) > h.get(big)) big = r;
            if (big == i) return;
            int t = h.get(i);
            h.set(i, h.get(big));
            h.set(big, t);
            i = big;
        }
    }

    static void decreaseKey(List<Integer> h, int i, int newValue) {
        if (i < 0 || i >= h.size())
            throw new IndexOutOfBoundsException(
                    "decreaseKey: no element at that index");
        if (newValue > h.get(i))
            throw new IllegalArgumentException("decreaseKey: new value is larger");
        h.set(i, newValue);
        siftDown(h, i);
    }
```

Kode 16.33 (Lanjutan) *Ex16Solutions.java: penyelesaian yang sama dalam Java.*

```
    // ---------- Exercise 16.2: merge k sorted lists with a min-heap ----------
    static List<Integer> mergeSorted(int[][] lists) {
        // entries are {value, which list, position in that list}
        PriorityQueue<int[]> heap = new PriorityQueue<>((x, y) ->
                x[0] != y[0] ? Integer.compare(x[0], y[0])
              : x[1] != y[1] ? Integer.compare(x[1], y[1])
              : Integer.compare(x[2], y[2]));
        for (int j = 0; j < lists.length; j++)
            if (lists[j].length > 0) heap.add(new int[] {lists[j][0], j, 0});
        List<Integer> out = new ArrayList<>();
        while (!heap.isEmpty()) {
            int[] e = heap.poll();
            out.add(e[0]);
            int j = e[1], p = e[2];
            if (p + 1 < lists[j].length)
                heap.add(new int[] {lists[j][p + 1], j, p + 1});
        }
        return out;
    }

    // ---------- Exercise 16.3: is the graph bipartite? (BFS two-coloring) ------
    static boolean bipartite(int[][] adj) {
        int[] color = new int[adj.length];
        java.util.Arrays.fill(color, -1);
        for (int s = 0; s < adj.length; s++) {
            if (color[s] != -1) continue;
            color[s] = 0;
            ArrayDeque<Integer> q = new ArrayDeque<>();
            q.add(s);
            while (!q.isEmpty()) {
                int u = q.poll();
                for (int v : adj[u]) {
                    if (color[v] == -1) {
                        color[v] = 1 - color[u];   // the other side
                        q.add(v);
                    } else if (color[v] == color[u]) {
                        return false;              // an edge inside one side
                    }
                }
            }
        }
        return true;
    }
```

Kode 16.33 (Lanjutan) *Ex16Solutions.java: penyelesaian yang sama dalam Java.*

```
    // ---------- Exercise 16.4: count connected components (DFS, a stack) ------
    static int components(int[][] adj) {
        boolean[] seen = new boolean[adj.length];
        int count = 0;
        for (int s = 0; s < adj.length; s++) {
            if (seen[s]) continue;
            count++;
            ArrayDeque<Integer> stack = new ArrayDeque<>();
            stack.push(s);
            seen[s] = true;
            while (!stack.isEmpty()) {
                int u = stack.pop();
                for (int v : adj[u])
                    if (!seen[v]) {
                        seen[v] = true;
                        stack.push(v);
                    }
            }
        }
        return count;
    }

    // ---------- Exercise 16.5: the first character that occurs only once ------
    static String firstUnique(String s) {
        HashMap<Character, Integer> count = new HashMap<>();
        for (char c : s.toCharArray()) count.merge(c, 1, Integer::sum);    // pass 1
        for (char c : s.toCharArray())                                     // pass 2
            if (count.get(c) == 1) return String.valueOf(c);
        return "none";
    }

    // ---------- Exercise 16.6: two prices that add up to a voucher ----------
    static int[] twoSum(int[] price, int target) {
        HashSet<Integer> seen = new HashSet<>();
        for (int p : price) {
            if (seen.contains(target - p)) return new int[] {target - p, p};
            seen.add(p);
        }
        return null;
    }

    // ---------- Exercise 16.14: the Profitina Laundry service desk ----------
    static class LaundryDesk {
        record Order(int number, String customer, int readyBy) {}
```

Kode 16.33 (Lanjutan) *Ex16Solutions.java: penyelesaian yang sama dalam Java.*

```
        final HashMap<Integer, Order> byNumber = new HashMap<>();    // O(1) lookup
        final HashMap<Integer, String> status = new HashMap<>();
        // entries are {promised hour, order number}; earliest promise first
        final PriorityQueue<int[]> next = new PriorityQueue<>((x, y) ->
                x[0] != y[0] ? Integer.compare(x[0], y[0])
                             : Integer.compare(x[1], y[1]));

        void receive(int number, String customer, int readyBy) {
            byNumber.put(number, new Order(number, customer, readyBy));
            status.put(number, "waiting");
            next.add(new int[] {readyBy, number});
        }

        void washNext() {
            if (next.isEmpty()) {
                System.out.println("  nothing to wash");
                return;
            }
            int[] top = next.poll();
            status.put(top[1], "washed");
            System.out.println("  wash order " + top[1] + " ("
                    + byNumber.get(top[1]).customer() + ", ready by " + top[0]
                    + ":00)");
        }

        void showStatus(int number) {
            System.out.println("  order " + number + ": "
                    + status.getOrDefault(number, "unknown"));
        }
    }

    public static void main(String[] args) {
        System.out.println("Exercise 16.1");
        List<Integer> h = new ArrayList<>(List.of(8, 5, 6, 4, 5, 3, 2, 1, 3, 1));
        decreaseKey(h, 0, 2);
        System.out.println("  after decreaseKey(0, 2): " + join(h));

        System.out.println("Exercise 16.2");
        System.out.println("  merged: " + join(mergeSorted(
                new int[][] {{3, 9, 14}, {1, 10}, {2, 4, 8, 20}, {}})));
```

Kode 16.33 (Lanjutan) *Ex16Solutions.java: penyelesaian yang sama dalam Java.*

```
        System.out.println("Exercise 16.3");
        int[][] branches = {{1, 2, 3}, {0, 6, 4}, {0, 5}, {0, 6},
                            {1, 5}, {2, 4}, {1, 3}, {}};
        int[][] square = {{1, 3}, {0, 2}, {1, 3}, {2, 0}};
        System.out.println("  branch map bipartite: " + bipartite(branches)
                + "\n  a square bipartite: " + bipartite(square));

        System.out.println("Exercise 16.4");
        System.out.println("  components of the branch map: "
                + components(branches));

        System.out.println("Exercise 16.5");
        for (String s : new String[] {"borrowed books", "aabb"})
            System.out.println("  \"" + s + "\" -> " + firstUnique(s));

        System.out.println("Exercise 16.6");
        int[] price = {35, 80, 42, 17, 60, 23};
        for (int target : new int[] {77, 100}) {
            int[] ab = twoSum(price, target);
            if (ab != null)
                System.out.println("  " + target + " = " + ab[0] + " + " + ab[1]);
            else
                System.out.println("  " + target + ": no pair");
        }

        System.out.println("Exercise 16.14");
        LaundryDesk desk = new LaundryDesk();
        desk.receive(501, "Ani", 17);
        desk.receive(502, "Budi", 10);
        desk.receive(503, "Citra", 12);
        desk.washNext();
        desk.receive(504, "Dewi", 9);                  // an express order
        desk.washNext();
        desk.showStatus(503);
        desk.showStatus(502);
        desk.showStatus(999);
    }
}
```

Kode 16.34 `ex16_solutions.py`: penyelesaian yang sama dalam Python.

```python
# ex16_solutions.py - solutions to the programming exercises of Chapter 16.
import heapq
from collections import Counter, deque


# ---------- Exercise 16.1: decreaseKey in a max-heap (a smaller key sinks) ----
def sift_down(h, i):
    while True:
        l, r, big = 2 * i + 1, 2 * i + 2, i
        if l < len(h) and h[l] > h[big]:
            big = l
        if r < len(h) and h[r] > h[big]:
            big = r
        if big == i:
            return
        h[i], h[big] = h[big], h[i]
        i = big


def decrease_key(h, i, new_value):
    if not 0 <= i < len(h):
        raise IndexError("decreaseKey: no element at that index")
    if new_value > h[i]:
        raise ValueError("decreaseKey: new value is larger")
    h[i] = new_value
    sift_down(h, i)


# ---------- Exercise 16.2: merge k sorted lists with a min-heap ----------
def merge_sorted(lists):
    # entries are (value, which list, position in that list)
    heap = [(lst[0], j, 0) for j, lst in enumerate(lists) if lst]
    heapq.heapify(heap)
    out = []
    while heap:
        v, j, p = heapq.heappop(heap)
        out.append(v)
        if p + 1 < len(lists[j]):
            heapq.heappush(heap, (lists[j][p + 1], j, p + 1))
    return out
```

Kode 16.34 (Lanjutan) *ex16_solutions.py: penyelesaian yang sama dalam Python.*

```
# ---------- Exercise 16.3: is the graph bipartite? (BFS two-coloring) ----------
def bipartite(adj):
    color = [-1] * len(adj)
    for s in range(len(adj)):
        if color[s] != -1:
            continue
        color[s] = 0
        q = deque([s])
        while q:
            u = q.popleft()
            for v in adj[u]:
                if color[v] == -1:
                    color[v] = 1 - color[u]         # the other side
                    q.append(v)
                elif color[v] == color[u]:
                    return False                    # an edge inside one side
    return True


# ---------- Exercise 16.4: count connected components (DFS, a stack) ----------
def components(adj):
    seen = [False] * len(adj)
    count = 0
    for s in range(len(adj)):
        if seen[s]:
            continue
        count += 1
        stack = [s]
        seen[s] = True
        while stack:
            u = stack.pop()
            for v in adj[u]:
                if not seen[v]:
                    seen[v] = True
                    stack.append(v)
    return count


# ---------- Exercise 16.5: the first character that occurs only once ----------
def first_unique(s):
    count = Counter(s)                              # pass 1: count
    for c in s:                                     # pass 2: in the original order
        if count[c] == 1:
            return c
    return "none"
```

Kode 16.34 (Lanjutan) *ex16_solutions.py: penyelesaian yang sama dalam Python.*

```
# ---------- Exercise 16.6: two prices that add up to a voucher ----------
def two_sum(price, target):
    seen = set()
    for p in price:
        if target - p in seen:                      # its partner came earlier
            return target - p, p
        seen.add(p)
    return None


# ---------- Exercise 16.14: the Profitina Laundry service desk ----------
class LaundryDesk:
    def __init__(self):
        self.by_number = {}             # number -> [customer, hour, status]
        self.next = []                  # min-heap of (promised hour, number)

    def receive(self, number, customer, ready_by):
        self.by_number[number] = [customer, ready_by, "waiting"]
        heapq.heappush(self.next, (ready_by, number))

    def wash_next(self):
        if not self.next:
            print("  nothing to wash")
            return
        hour, number = heapq.heappop(self.next)
        self.by_number[number][2] = "washed"
        customer = self.by_number[number][0]
        print(f"  wash order {number} ({customer}, ready by {hour}:00)")

    def status(self, number):
        order = self.by_number.get(number)
        print(f"  order {number}: {order[2] if order else 'unknown'}")


def main():
    print("Exercise 16.1")
    h = [8, 5, 6, 4, 5, 3, 2, 1, 3, 1]
    decrease_key(h, 0, 2)
    print("  after decreaseKey(0, 2):", " ".join(map(str, h)))

    print("Exercise 16.2")
    print("  merged:", " ".join(map(str, merge_sorted([[3, 9, 14], [1, 10],
                                                       [2, 4, 8, 20], []]))))
```

Kode 16.34 (Lanjutan) *ex16_solutions.py: penyelesaian yang sama dalam Python.*

```
    print("Exercise 16.3")
    branches = [[1, 2, 3], [0, 6, 4], [0, 5], [0, 6], [1, 5], [2, 4], [1, 3], []]
    square = [[1, 3], [0, 2], [1, 3], [2, 0]]
    print("  branch map bipartite:", str(bipartite(branches)).lower())
    print("  a square bipartite:", str(bipartite(square)).lower())

    print("Exercise 16.4")
    print("  components of the branch map:", components(branches))

    print("Exercise 16.5")
    for s in ("borrowed books", "aabb"):
        print(f'  "{s}" -> {first_unique(s)}')

    print("Exercise 16.6")
    price = [35, 80, 42, 17, 60, 23]
    for target in (77, 100):
        ab = two_sum(price, target)
        print(f"  {target} = {ab[0]} + {ab[1]}" if ab else f"  {target}: no pair")

    print("Exercise 16.14")
    desk = LaundryDesk()
    desk.receive(501, "Ani", 17)
    desk.receive(502, "Budi", 10)
    desk.receive(503, "Citra", 12)
    desk.wash_next()
    desk.receive(504, "Dewi", 9)                    # an express order
    desk.wash_next()
    desk.status(503)
    desk.status(502)
    desk.status(999)


if __name__ == "__main__":
    main()
```

```
Exercise 16.1
  after decreaseKey(0, 2): 6 5 3 4 5 2 2 1 3 1
Exercise 16.2
  merged: 1 2 3 4 8 9 10 14 20
Exercise 16.3
  branch map bipartite: false
  a square bipartite: true
Exercise 16.4
  components of the branch map: 2
Exercise 16.5
  "borrowed books" -> w
  "aabb" -> none
Exercise 16.6
  77 = 35 + 42
  100: no pair
Exercise 16.14
  wash order 502 (Budi, ready by 10:00)
  wash order 504 (Dewi, ready by 9:00)
  order 503: waiting
  order 502: washed
  order 999: unknown
```

16.1. Kunci yang lebih kecil hanya dapat merusak invarian terhadap anak-anaknya, tidak pernah terhadap induknya, sehingga perbaikannya berjalan *ke bawah*: `siftDown` dari indeks itu. Pada *heap* yang diberikan, akar menjadi 2, tenggelam melewati 6 (anak yang lebih besar) lalu melewati 3 (di indeks 5, lebih besar daripada 2 di indeks 6): `6 5 3 4 5 2 2 1 3 1`. Ongkosnya $O(\lg n)$ menurut Persamaan (16.2).

16.2. *Heap* memuat nilai terkecil yang tersisa dari setiap *list*, beserta *list* dan posisi asalnya. Setiap *pop* mengeluarkan minimum global, dan nilai berikutnya dari *list* yang sama menggantikannya. *Heap* tidak pernah memuat lebih dari k entri, sehingga setiap dari N nilai berongkos $O(\lg k)$: $O(N \lg k)$ seluruhnya. *List* yang kosong tidak pernah dimasukkan ke *heap*.

16.3. Warnai simpul awal dengan 0 dan setiap simpul yang baru ditemukan dengan warna kebalikan dari penemunya; sisi di antara dua simpul berwarna sama membuktikan adanya siklus ganjil, dan graf yang bersiklus ganjil tidak dapat diwarnai dengan dua warna. Peta cabang memuat siklus lima simpul Pusat, Gubeng, Rungkut, Wonokromo, Darmo, Pusat, sehingga tidak bipartit; persegi bipartit. Ongkosnya $O(|V| + |E|)$.

16.4. Setiap penelusuran dari simpul yang belum terlihat menandai satu komponen utuh; banyaknya penelusuran yang dimulai adalah banyaknya komponen. Peta cabang memiliki 2: tujuh cabang yang terhubung, dan Benowo sendirian.

16.5. Lewat pertama menghitung setiap karakter di *hash map*; lewat kedua menyusuri string dalam urutan aslinya dan mengembalikan karakter pertama yang hitungannya 1. Pada “borrowed books”, b, o, r muncul lebih dari sekali dan w adalah yang tunggal pertama. “aabb” tidak memilikinya. Dua kali lewat, $O(L)$ untuk string sepanjang L.

16.6. Untuk setiap harga p, pasangannya $t - p$ harus sudah muncul *sebelumnya*, sehingga kita mencarinya sebelum memasukkan p. Untuk 77: pada 42, himpunannya memuat {35, 80}, dan $77 - 42 = 35$ ada di sana. Untuk 100 tidak ada pasangan. Mencari sebelum memasukkan juga yang mencegah sebuah harga dipasangkan dengan dirinya sendiri: untuk voucer 70 dan satu harga 35, pencarian 35 terjadi ketika 35 belum berada di himpunan. (Bab 4 menyelesaikan soal yang sama dengan dua indeks pada larik *terurut* dalam $O(n \lg n)$; *hash set* mengerjakannya rata-rata dalam $O(n)$ tanpa pengurutan.)

16.7. Setelah setiap penyisipan: `[20]`; `[20, 5]`; `[20, 5, 15]`; `[22, 20, 15, 5]` (22 naik melewati 5 dan 20); `[40, 22, 15, 5, 20]` (40 naik melewati 20 dan 22); `[40, 22, 15, 5, 20, 11]`; `[40, 22, 30, 5, 20, 11, 15]` (30 naik melewati 15). `extractMax` mengembalikan 40; 15 pindah ke akar dan tenggelam melewati 30: `[30, 22, 15, 5, 20, 11]`.

16.8. Dengan $n = 10$, indeks internalnya 4 turun sampai 0. `heapifyDown(4)`: 16 lebih besar daripada anaknya 7, tidak ada yang dikerjakan. `heapifyDown(3)`: 2 lawan 14 dan 8, tukar dengan 14: `4 1 3 14 16 9 10 2 8 7`. `heapifyDown(2)`: 3 lawan 9 dan 10, tukar dengan 10: `4 1 10 14 16 9 3 2 8 7`. `heapifyDown(1)`: 1 lawan 14 dan 16, tukar dengan 16, lalu 1 lawan 7, tukar: `4 16 10 14 7 9 3 2 8 1`. `heapifyDown(0)`: 4 lawan 16 dan 10, tukar dengan 16; lalu lawan 14 dan 7, tukar dengan 14; lalu lawan 2 dan 8, tukar dengan 8: `16 14 10 8 7 9 3 2 4 1`. Ini contoh dari Cormen dkk. [9], dan hasilnya sama dengan hasil mereka.

16.9. BFS dari 0: mengambil 0 menemukan 1 dan 2 (jarak 1); mengambil 1 menemukan 3 (jarak 2); mengambil 2 mendapati 3 sudah terlihat dan menemukan 4 (jarak 2, dari 2); urutannya 0, 1, 2, 3, 4 dengan jarak 0, 1, 1, 2, 2. DFS rekursif dari 0 berjalan 0, 1, 3, 4, lalu mundur ke 0 dan mengunjungi 2, yang tetangganya 3 dan 4 sudah dikunjungi: urutannya 0, 1, 3, 4, 2.

16.10. Dijkstra mengambil A (0), merelaksasi B menjadi 2 dan C menjadi 5, mengambil B pada 2 dan menyatakannya selesai; lalu mengambil C pada 5, yang busurnya menawarkan B $5 - 4 = 1$, tetapi B sudah selesai, sehingga jarak yang dilaporkan 2. Jarak sebenarnya 1, melalui A→C→B. Argumen invarian “setiap rute lain harus meninggalkan wilayah selesai pada waktu yang paling sedikit sama besar, dan panjang jalan tidak pernah negatif” gagal pada klausa terakhirnya: rute

melalui C berangkat pada 5 lalu *berkurang* sesudahnya. Bellman–Ford menangani bobot negatif [9].

16.11. 12 → 5; 19 → 5 terisi, 6; 5 → 5 terisi, 6 terisi, 0; 26 → 5, 6, 0 terisi, 1; 33 → 5, 6, 0, 1 terisi, 2. Tabelnya: 0: 5, 1: 26, 2: 33, 5: 12, 6: 19. Hapus 19: slot 6 menjadi *tombstone*. Cari 26: *probe* 5 (12), 6 (*tombstone*, lanjut), 0 (5), 1 (26): ketemu. Cari 40 (40 mod 7 = 5): 5, 6, 0, 1, 2, 3 (kosong): tidak ada setelah enam *probe*. Bila slot 6 dikosongkan, pencarian 26 akan berhenti di 6 dan melewatkannya.

16.12. (a) Tabel hash: pencarian menurut nomor saja, sepanjang hari. (b) *Heap* (*priority queue*) berkunci lama keterlambatan: selalu yang terbesar berikutnya. (c) Larik terurut: dicetak berurutan setiap malam, jarang berubah, sehingga ongkos penyisipan tidak berarti. (d) Graf dengan *adjacency list*: hubungan antarbutir. (e) *Stack*: *undo* mengambil perubahan terbaru lebih dahulu. (f) Kueri rentang: larik terurut atau pohon pencarian seimbang dengan `lower_bound`, tidak pernah tabel hash.

16.13. Matriks: $20,000^2 = 4 \times 10^8$ sel, 1,6 GB dengan bilangan bulat 4 bita, 50 MB dengan satu bit per sel. *List*: $2 \times 52{,}000 = 104{,}000$ entri ditambah 20.000 kepala *list*, jauh di bawah 1 MB. BFS pada *list* berongkos $O(|V| + |E|) \approx 124{,}000$ langkah, pada matriks $|V|^2 = 4 \times 10^8$. Pilihlah *list*; menurut Persamaan (16.12), matriks hanya sepadan untuk graf yang padat.

16.14. *Hash map* dari nomor pesanan ke pesanan menjawab `status` dalam $O(1)$; *min-heap* berisi (jam janji, nomor pesanan) memberikan pesanan berikutnya untuk dicuci dalam $O(\lg n)$. Budi (10.00) dicuci pertama; pesanan kilat Dewi (09.00), yang diterima sesudahnya, melompati Citra (12.00) dan dicuci kedua; 503 masih menunggu, 502 sudah dicuci, dan 999 tidak dikenal. Keluarannya adalah blok terakhir keluaran program di atas.

Daftar Pustaka

[1] Free Software Foundation, “Program Instrumentation Options,” in *Using the GNU Compiler Collection (GCC)*, GCC 16.2, Aug. 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://gcc.gnu.org/onlinedocs/

[2] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in *Proc. USENIX Annual Technical Conf.*, Boston, MA, USA, 2012, pp. 309–318.

[3] Valgrind Developers, *Valgrind User Manual*, release 3.27.1, May 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://valgrind.org/docs/manual/

[4] B. Stroustrup, *A Tour of C++*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2022.

[5] cppreference.com, “std::priority_queue,” “std::map,” “std::unordered_map,” and “std::push_heap.” Accessed: Sep. 24, 2026. [Online]. Available: https://en.cppreference.com/w/cpp

[6] Oracle, “Class PriorityQueue,” “Class HashMap,” “Class TreeMap,” and “Class ArrayDeque,” *Java SE 25 API Specification*. Accessed: Sep. 24, 2026. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/

[7] Python Software Foundation, “heapq — Heap queue algorithm,” “bisect,” “collections.deque,” and “Mapping Types — dict,” *Python 3.14 Documentation*. Accessed: Sep. 24, 2026. [Online]. Available: https://docs.python.org/3/

[8] R. W. Floyd, “Algorithm 245: Treesort 3,” *Communications of the ACM*, vol. 7, no. 12, p. 701, 1964.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[10] A. B. Kahn, “Topological sorting of large networks,” *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[11] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[12] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

[13] Z. Zajac, “goodbooks-10k: A new dataset for book recommendations,” 2017. [Online]. Available: https://github.com/zygmuntz/goodbooks-10k. License CC BY-SA 4.0. Accessed: Sep. 24, 2026.

[14] J. W. J. Williams, “Algorithm 232: Heapsort,” *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, 1964.

[15] E. F. Moore, “The shortest path through a maze,” in *Proc. Int. Symp. on the Theory of Switching*, Cambridge, MA, USA: Harvard Univ. Press, 1959, pp. 285–292.

[16] W. W. Peterson, “Addressing for random-access storage,” *IBM Journal of Research and Development*, vol. 1, no. 2, pp. 130–146, 1957.

[17] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011.

Heap dan *heap sort* berasal dari Williams [14], dan pembangunan *bottom-up* dalam waktu linear dari Floyd [8]; penelusuran melebar sebagai metode lintasan terpendek muncul dalam karya Moore tentang labirin [15], algoritma Dijkstra dalam catatannya tahun 1959 [11], dan pengurutan topologis jaringan besar dalam Kahn [10]. *Open addressing* dengan *linear probing* diperkenalkan oleh Peterson [16] dan dianalisis oleh Knuth [12], yang rumus-rumusnya dituliskan kembali oleh Persamaan (16.11). Cormen dkk. [9] serta Sedgewick dan Wayne [17] membahas setiap struktur dalam bab ini lengkap dengan buktinya; perilaku pustaka mengikuti cppreference [5], spesifikasi API Java [6], dan dokumentasi Python [7], sedangkan aturan bahasa C++ mengikuti Stroustrup [4]. Katalog di Subbab 16.8 adalah himpunan data goodbooks-10k [13]. Pemeriksa memori dijelaskan dalam manual GCC [1], manual Valgrind [3], dan makalah AddressSanitizer [2].