Kuliah 1 — Fondasi: Penyegaran C++, Memori, dan Array

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 1

Dosen: Irfan Subakti — yifana@gmail.com

Hari ini kita membangun lantai dasar perkuliahan ini: apa itu struktur data, bagaimana memori menyimpan data, C++ yang akan kita pakai sepanjang semester, dan struktur pertama dari semuanya, yaitu array. Buku: Bab 1, Subbab 1.1–1.10.

Cakupan Pertemuan Ini

1. **Mengapa struktur data** — Pustaka Ceria, antarmuka dan implementasi, dan mengapa pilihan yang tepat mengalahkan komputer yang lebih cepat.
2. **Memori** — bit, byte, alamat, tipe, urutan *little-endian*, stack dan heap.
3. **Penyegaran C++** — tipe, masukan dan keluaran, keputusan dan perulangan, fungsi, struct, serta operator `&` dan `*`.
4. **Array secara mendalam** — rumus alamat, rekaman, batas indeks, penelusuran, tabel *row-major*, `std::vector`, `ArrayList`, dan `list`.
5. **Cicipan pertama kompleksitas** — akses $O(1)$ lawan penyisipan $O(n)$, big-O, biaya teramortisasi; lalu meja kerja VS Code untuk tiga bahasa.

Setiap program hari ini tersedia dalam C++, Java, dan Python, dan setiap keluaran yang ditampilkan adalah keluaran yang benar-benar dicetak program. Buku: Bab 1.

Titik Berangkat: Perpustakaan Tanpa Rak

Pustaka Ceria, perpustakaan kecil di sebuah kampus, dibuka dengan delapan buku yang ditumpuk di lantai. Cara apa pun dapat menemukan satu buku di antara delapan.

Setahun kemudian koleksinya delapan puluh ribu buku. Membaca setiap sampul kini memakan rata-rata empat puluh ribu kali lihat.

Perpustakaan membeli rak, menomorinya, mengurutkan buku menurut kode, dan menyimpan katalog. Bukunya tidak berubah; yang berubah adalah **penataannya**.

Penataan itulah *struktur data*. Pustaka Ceria menemani kita sepanjang semester: array hari ini, lalu stack, queue, tree, graf, dan tabel hash.

Dari perkuliahan pemrograman sebelumnya kita membawa variabel, perulangan, dan fungsi; yang baru adalah memikirkan berapa banyak nilai yang disimpan program dan bagaimana ia menyimpannya. Buku: Subbab 1.1.1.

Apa Itu 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 dijalankan dengan benar dan efisien. Lima operasi yang lazim adalah **penyisipan**, **penghapusan**, **pencarian**, **penelusuran**, dan **pengurutan**.

Antarmuka — daftar operasi dan janji tiap operasi: “Saya dapat memberi Anda buku B1005, atau mengatakan bahwa buku itu tidak ada.” Tipe yang diuraikan hanya dengan cara ini disebut *tipe data abstrak* (ADT).

Implementasi — byte di memori dan langkah-langkah yang menepati janji itu: deretan terurut yang dicari dengan membagi dua, atau delapan kotak yang dipilih menurut angka terakhir kode.

Antarmuka adalah apa yang perlu diketahui pengunjung; implementasi adalah apa yang perlu diketahui pustakawan. Struktur yang baik membuat pustakawan dapat mengganti implementasi tanpa disadari pengunjung. Buku: Subbab 1.1.2.

Mengapa Pilihan Itu Penting

Tabel 1.1 menghitung banyaknya kali lihat untuk menemukan satu buku di antara n buku dengan tiga penataan yang berbeda.

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

Bila satu kali lihat memakan satu mikrodetik, satu miliar buku memerlukan tujuh belas menit dengan cara pertama dan tiga puluh mikrodetik dengan cara kedua. Komputer yang seratus kali lebih cepat pun tetap kalah. Memilih struktur data yang tepat biasanya lebih berharga daripada membeli komputer tercepat. Buku: Subbab 1.1.3.

Struktur Data di Balik Setiap Halaman Web

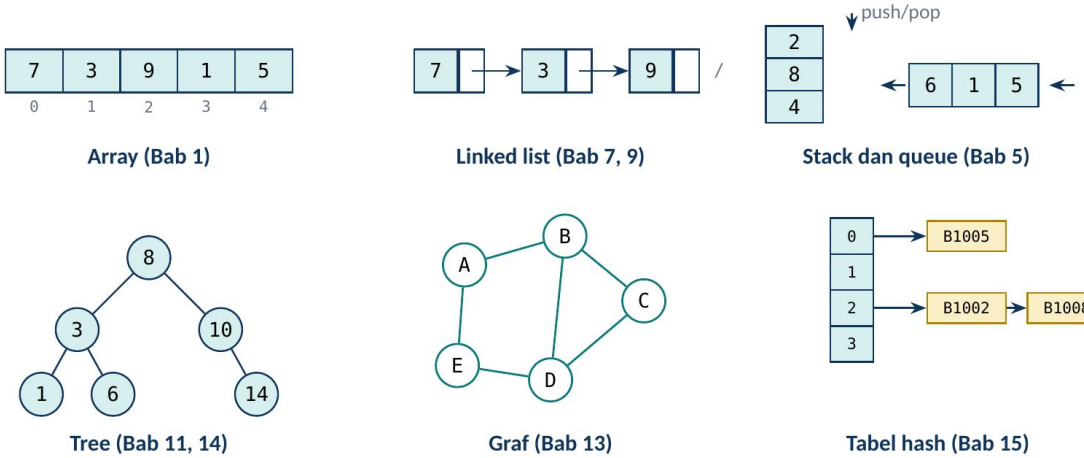
Internet adalah jaringan dari jaringan-jaringan yang berbicara dengan TCP/IP; **World Wide Web** hanyalah satu layanan di atasnya, tempat peramban meminta dokumen kepada server melalui HTTP.

Cache nama pada peramban adalah **tabel hash**; server-server nama membentuk **tree**; router menyimpan permintaan yang menunggu dalam **queue** dan memilih jalur di atas sebuah **graf**.

Server membaca berkas yang blok-bloknya didaftar dalam **array** dan diindeks oleh **tree**; peramban membangun **tree** elemen dan menelusurinya dengan **stack**.

Lima langkah dari satu permintaan halaman, sedikitnya enam struktur data, masing-masing dipilih karena membuat langkah itu murah, dan semuanya ada di buku. Buku: Subbab 1.1.4.

Enam Keluarga Struktur Data



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.

Gambar 1.1: array yang berjajar, linked list berupa kotak-kotak yang menunjuk ke kotak berikutnya, stack dan queue, tree dan heap, graf keterhubungan, dan tabel hash yang menghitung letak sebuah butir. Buku: Subbab 1.1.5.

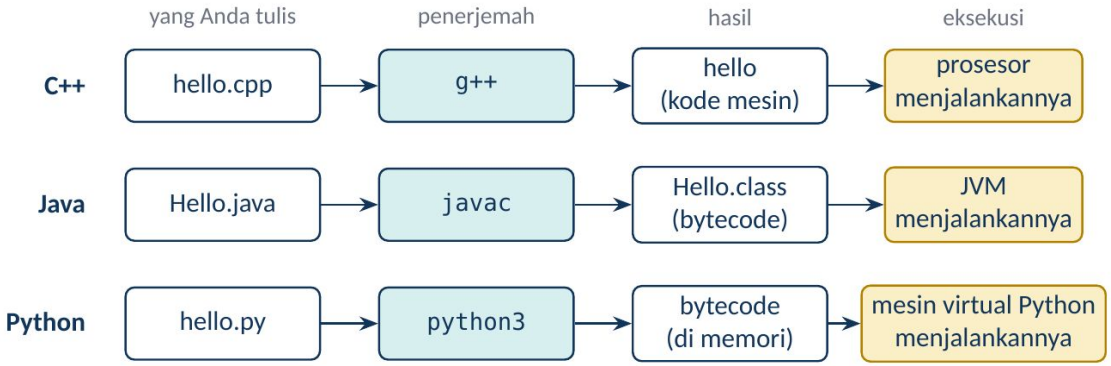
Tiga Bahasa, Satu Gagasan

Tabel 1.2 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%

*C++ tidak menyembunyikan apa pun: array delapan buku benar-benar berjajar di memori. **Java** menaruh setiap objek di heap; **Python** menjadikan setiap variabel nama bagi sebuah objek. Tabel 1.2 mengukur popularitas, bukan mutu. Buku: Subbab 1.2.1.*

Dari Berkas Sumber ke Program yang Berjalan



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.

Gambar 1.2: C++ dikompilasi menjadi kode mesin, Java menjadi bytecode untuk Java Virtual Machine, dan Python menjadi bytecode yang langsung dijalankan oleh mesin virtualnya sendiri. Buku: Subbab 1.2.2.

Bit, Byte, dan Heksadesimal

Bit adalah sakelar, 0 atau 1. Setiap bit tambahan menggandakan banyaknya keadaan, sehingga k bit memiliki banyak keadaan seperti pada Persamaan (1.1).

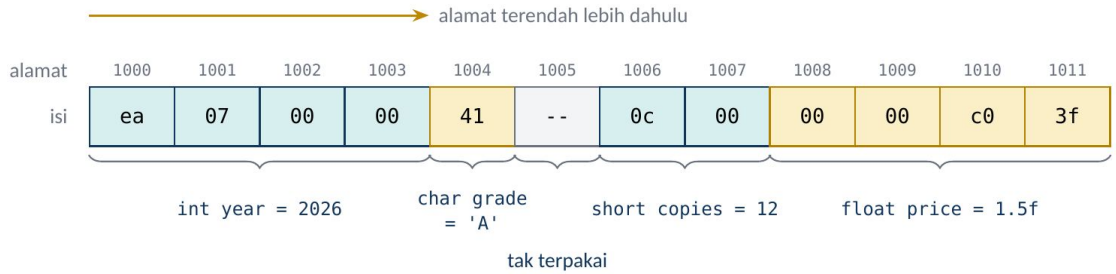
$$2^k \text{ keadaan berbeda} \tag{1.1}$$

Byte adalah delapan bit: $2^8 = 256$ keadaan, yaitu bilangan 0 sampai 255. Byte 11101010 bernilai $128+64+32+8+2=234$.

Heksadesimal menulis empat bit sebagai satu digit, `0`–`9` lalu `a`–`f`: 1110 1010 adalah `ea`, ditulis `0xEA` dalam ketiga bahasa.

*Memori adalah satu deret panjang byte, masing-masing dengan **alamat** bernomor; alamat mana pun dicapai dalam waktu singkat yang sama. Itulah akses acak, dan itulah yang membuat array cepat. Buku: Subbab 1.3.1 dan 1.3.2.*

Tipe Memberi Makna pada Byte



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.

*Pada Gambar 1.3, 2026 adalah `0x000007EA`, disimpan dengan byte paling tidak signifikan lebih dahulu: **little-endian**. Byte 1005 adalah **padding**, agar `short` mulai di alamat genap. Memori tidak tahu apa yang disimpannya; hanya program yang tahu, melalui tipe-tipenya. Buku: Subbab 1.3.3.*

Melihat Byte-Byte Itu Sendiri

Kode 1.1 memeriksa keempat byte dari `int` 2026 satu per satu, mulai dari alamat terendah.

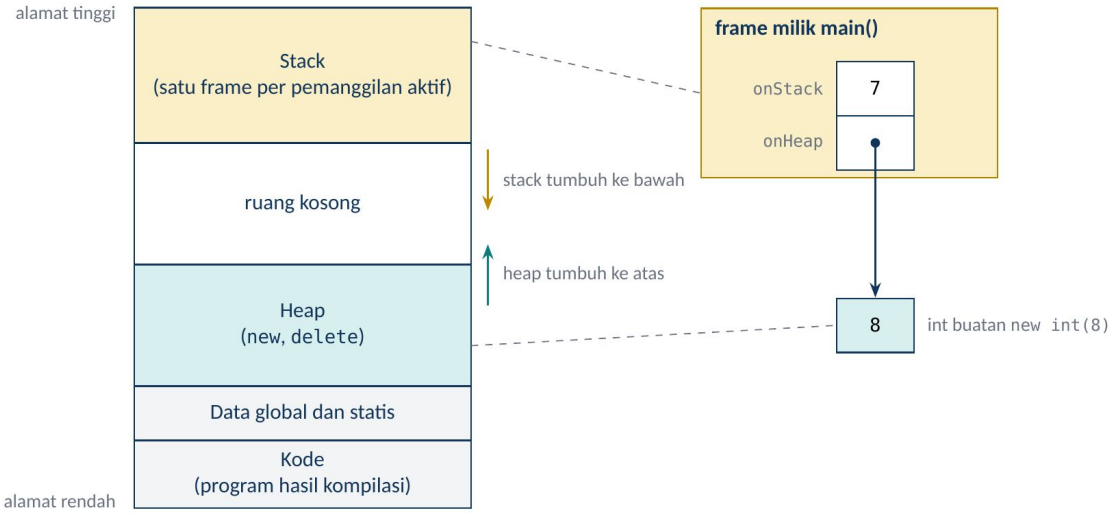
Kode 1.1 Dari `memory_basics.cpp`: byte-byte sebuah `int` dan urutan byte mesin.

```
    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";
```

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

`sizeof` memberi banyaknya byte suatu tipe; `&year` adalah alamat `year`; cast itu membuat kita dapat membaca byte satu per satu. Ukuran yang tercetak adalah ukuran pada mesin 64-bit; standar hanya menetapkan bahwa `char` berukuran satu byte, maka buku selalu bertanya kepada `sizeof`. Buku: Subbab 1.3.4.

Stack dan Heap



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*.

*Gambar 1.4: setiap pemanggilan mendapat **stack frame** yang dilepas ketika fungsi kembali, terakhir masuk pertama keluar; memori dari `new` tinggal di **heap** sampai `delete`, dan setiap `delete` yang terlupa adalah **kebocoran memori**. `memory_basics.cpp` mencetak alamat stack sekitar `0x7ff...` dan alamat heap sekitar `0x55...`. Java dan Python menaruh objek di heap dan membebaskannya dengan **garbage collector**. Buku: Subbab 1.3.5.*

Penyegaran C++: Tipe

Setiap program memiliki kerangka yang sama: `#include`, `using namespace std;`, `int main()`, pernyataan yang diakhiri `;`, dan `return 0`.

Tabel 1.3 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 $2^{64}-1$

Tabel 1.3 memuat ukuran yang dicetak `sizeof` pada mesin 64-bit. Tiga kebiasaan: beri setiap variabel nilai sebelum dibaca; pilih tipe yang cukup besar (luapan terjadi diam-diam); dan ingat bahwa `7 / 2` bernilai `3`, sedangkan `static_cast<double>(7) / 2` bernilai `3.5`. Buku: Subbab 1.4.1 dan 1.4.2.

Masukan dan Keluaran

Kode 1.2 Dari `io_sum.cpp`: membaca sebuah cacah lalu sebanyak itu bilangan dengan `cin`, dan meringkasnya.

```
    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;
```

Kode 1.2 membaca sebuah cacah, lalu sebanyak itu bilangan; `<<` dan `>>` menunjuk arah aliran data, dan `if (!(cin >> n) || n <= 0)` membaca sekaligus memeriksa. Jalankan sebagai `./io_sum < week_loans.txt` untuk pekan `12 9 15 4 20 11 6`: total 77, rata-rata 11.00, terkecil 4, terbesar 20. Java memakai `Scanner`, Python `sys.stdin.read().split()`. Buku: Subbab 1.4.3.

Keputusan, Perulangan, dan Fungsi

`if`/`else`, `switch` (akhiri setiap `case` dengan `break` atau `return`), `while`, `do … while`, dan `for`; kondisi dengan `==`, `!=`, `&&`, `||`, `!`.

`for (int i = 0; i < n; i++)` berjalan tepat n kali dengan `i` $= 0, \ldots, n-1$: persis indeks-indeks sah sebuah array berisi n elemen.

Fungsi memiliki signatur, `long lateFee(int daysKept, char memberType)`; C++ mengoper parameter **berdasarkan nilai**, sehingga fungsi menerima salinan.

Struct mengelompokkan nilai-nilai yang berkaitan di bawah satu nama; titik memilih satu *field*: `first.title`. Array berisi struct adalah tabel rekaman yang paling sederhana.

`break` langsung keluar dari perulangan; `continue` melompat ke putaran berikutnya. `for` berbasis rentang, `for (const Loan& loan : loans)`, mengunjungi setiap elemen tanpa indeks. Buku: Subbab 1.4.4–1.4.6.

Contoh Terkerjakan 1: Biaya Keterlambatan — Aturannya

Kode 1.3 Dari `late_fee.cpp`: fungsi biaya keterlambatan dengan `if` dan `switch`.

```
// 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;
    }
}
```

Masa pinjam tujuh hari; keterlambatan dikenai Rp1.000 per hari pada pekan pertama dan Rp2.000 per hari sesudahnya; mahasiswa membayar separuh, tamu dua kali lipat. Pada Kode 1.3, `-1` berarti "jenis anggota tidak dikenal". Akhiran `L` pada `1000L` membuat hitungan berlangsung dalam `long`. Buku: Subbab 1.4.6.

Contoh Terkerjakan 1: Perulangan atas Peminjaman

`Loan loans[] = { {"Laskar Pelangi", 5, 'S'}, …, {"Sejarah Nusantara", 9, 'X'} };` memuat enam peminjaman; Kode 1.4 menjumlahkan biayanya dan melewati rekaman yang buruk.

Kode 1.4 Fungsi `main` pada `late_fee.cpp`: array berisi struct dan `for` berbasis rentang dengan `continue`.

```
    long total = 0;
    for (const Loan& loan : loans) {
        long fee = lateFee(loan.daysKept, loan.memberType);
        if (fee < 0) {
            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';
```

`struct Loan` di atasnya memiliki tiga field: `title`, `daysKept`, dan `memberType`. Buku: Subbab 1.4.6.

Contoh Terkerjakan 1: Periksa Dua Baris dengan Tangan

```
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
```

Fisika Dasar, tamu, 16 hari: terlambat $16-7=9$ hari; dasar $7\times1000+(9-7)\times2000=11.000$; dua kali lipat: 22.000.

Negeri 5 Menara, mahasiswa, 20 hari: terlambat 13 hari; dasar $7000+6\times2000=19.000$; separuhnya: 9.500.

Sejarah Nusantara, jenis `'X'`: `lateFee` mengembalikan -1; perulangan melaporkannya lalu lanjut, sehingga satu rekaman buruk tidak merusak total.

Menjalankan program dengan tangan adalah kebiasaan paling berguna yang dapat diajarkan perkuliahan ini. Buku: Subbab 1.4.6.

Aturan yang Sama dalam Python

Kode 1.5 adalah versi Python: sebuah `dataclass` untuk peminjaman dan pernyataan `match` untuk jenis anggota.

Kode 1.5 Dari `late_fee.py`: fungsi biaya keterlambatan dengan `match`.

```python
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
```

Operator `/` pada Python selalu menghasilkan bilangan real, maka separuh biaya ditulis `base // 2`. Java memakai `record` dan `switch`. Ketiganya mencetak tujuh baris yang sama. `match` memerlukan Python 3.10 atau yang lebih baru. Buku: Subbab 1.4.6.

Operator Alamat dan Pointer Pertama

`&age` adalah **alamat** `age`; `int* p = &age;` mendeklarasikan pointer; `*p` mengikutinya (**dereferensi**). Kode 1.6 mengoper `age` dengan dua cara.

Kode 1.6 Dari `address_of.cpp`: salinan tidak dapat mengubah variabel pemanggil; alamat dapat.

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

void birthdayByAddress(int* p) {           // p menyimpan alamat milik pemanggil
    *p = *p + 1;
}
    birthdayByValue(age);
    cout << "after birthdayByValue(age), age = " << age << '\n';
    birthdayByAddress(&age);
    cout << "after birthdayByAddress(&age), age = " << age << '\n';
```

```
after *p = 21, age = 21
after birthdayByValue(age), age = 21
after birthdayByAddress(&age), age = 22
```

*`sizeof(p)` bernilai 8: setiap alamat memakan delapan byte pada mesin 64-bit. Java dan Python tidak pernah menampilkan alamat, tetapi variabel bertipe objek menyimpan **referensi**, dan metode yang menerimanya dapat mengubah objek itu. Bab 6 menuntaskan ceritanya. Buku: Subbab 1.4.7.*

Array: Satu Deret Slot Bernomor

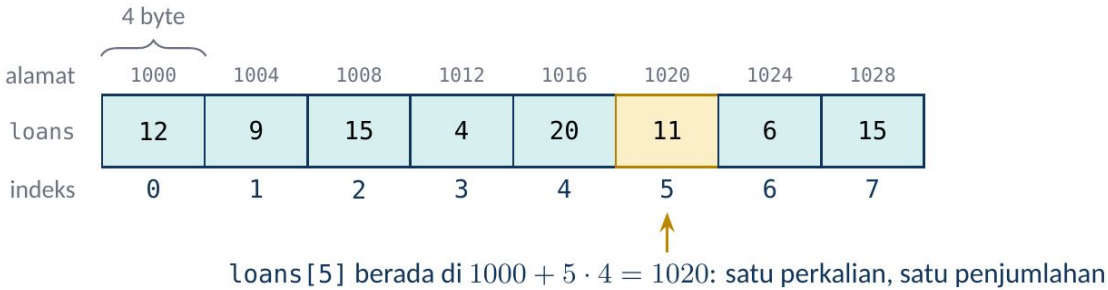
DEFINISI. *Array* adalah sejumlah tetap elemen bertipe sama, disimpan berurutan dalam satu blok memori yang bersambung, masing-masing dikenali oleh sebuah *indeks* bilangan bulat. Indeks pertama adalah 0, sehingga array berisi n elemen memiliki indeks $0, 1, \ldots, n-1$.

`int loans[8] = {12, 9, 15, 4, 20, 11, 6, 15};` — ukurannya diketahui saat program dikompilasi dan tidak pernah berubah.

`int zeros[5] = {0};` membuat setiap elemen bernilai nol; `int loans[] = {…};` membiarkan kompilator yang menghitung.

`loans[3] = 5;` *menaruh 5 di tempat keempat: indeks 3 adalah slot keempat. Buku: Subbab 1.5.1.*

Rumus Alamat



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

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

Persamaan (1.2) dengan $b = 1000$ dan $s = 4$ pada Gambar 1.5: `loans[5]` mulai di $1000 + 5 \cdot 4 = 1020$. Satu perkalian dan satu penjumlahan, baik array itu berisi delapan elemen maupun delapan juta. Buku: Subbab 1.5.1.

Mengapa Indeks Dimulai dari 0

Indeks bukan “elemen keberapa dalam hitungan”, melainkan **berapa elemen dari awal**: elemen pertama berada di $b + 0 \cdot s = b$.

Bila indeks dimulai dari 1, setiap akses memerlukan satu pengurangan tambahan, $b + (i - 1) \cdot s$, untuk selamanya.

Dalam C++, `loans[i]` sama persis dengan `*(loans + i)`, dan `&loans[i]` sama persis dengan `loans + i`: menambahkan i pada pointer memajukannya sejauh i *elemen*.

INGAT. Array berisi n elemen **tidak** memiliki elemen berindeks n. Elemen terakhir berindeks $n - 1$.

Kurung siku hanyalah ejaan yang lebih ramah untuk “mulai dari alamat dasar, maju i elemen, ikuti alamatnya”. Buku: Subbab 1.5.2.

Array Berisi Rekaman: Katalog

Kode 1.7 menyimpan setiap *field* di dalam rekaman itu sendiri, sehingga satu `Book` adalah blok padat sebesar $6+100+60+30=196$ byte.

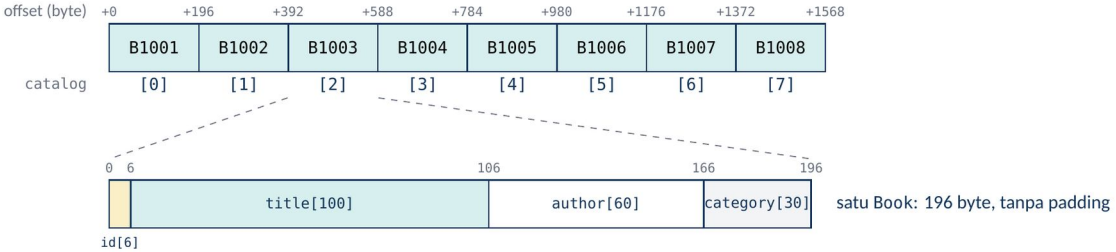
Kode 1.7 Dari `book.h`: rekaman `Book` dengan array karakter berukuran tetap, dan fungsi bantu yang tidak pernah menulis melewati sebuah *field*.

```
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';
}
```

`std::string` akan menyimpan karakternya di tempat lain, di heap. Harga array berukuran tetap adalah batas: judul paling banyak 99 karakter, karena satu `char` menampung penutup `'\0'`. `safeCopy` memotong teks yang lebih panjang. Buku: Subbab 1.5.3.

Delapan Buku Berjajar



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.

Gambar 1.6: field-nya mulai pada offset 0, 6, 106, dan 166; buku-buku yang berurutan berjarak 196 byte, seperti diramalkan Persamaan (1.2) dengan $s = 196$. Tidak ada padding, karena `char` boleh mulai di alamat mana pun. Buku: Subbab 1.5.3.

Mengukur Letak Awal Setiap Buku

Kode 1.8 Dari `catalog_1d.cpp`: *offset* byte setiap elemen, dan pemeriksaan bahwa `&catalog[i]` sama dengan `catalog + i`.

```
    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");
    }
```

```
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
```

Kode 1.8 mengukur jarak setiap elemen dari `catalog[0]`. Di Java, `new Book[8]` membuat delapan referensi, dan setiap `new Book(...)` adalah objek tersendiri di heap; list Python pun demikian. Rumusnya tetap menemukan referensi dalam satu langkah; satu langkah lagi mengikutinya ke buku. Buku: Subbab 1.5.3.

Tetap di Dalam Batas

C++ tidak memeriksa: `loans[8]` membaca apa saja yang ada di $b{+}8{\cdot}4$ — **perilaku tak terdefinisi**. Pengakses berpemeriksa `at(i)` melempar `out_of_range` (Kode 1.9).

Kode 1.9 Dari `bounds.cpp`: pengindeksan di dalam batas, dan `at()` yang menolak indeks di luar rentang.

```
    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.at(3) = 40
v.at(5) refused: vector::_M_range_check: __n (which is 5) >= this->size() (which is 5)
```

Java melempar `ArrayIndexOutOfBoundsException` dan Python `IndexError`, tepat di baris kesalahannya. Python juga menerima indeks negatif: `a[-1]` adalah elemen terakhir; di C++, `a[-1]` adalah memori tepat sebelum array. Buku: Subbab 1.5.4.

Menelusuri Array: Jumlah dan Maksimum

Kode 1.10 Dari `array_ops.cpp`: jumlah, dan posisi elemen terbesar.

```
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;
}
```

Kode 1.10: penelusuran mengunjungi setiap elemen tepat sekali, dari indeks 0 sampai $n-1$*. Parameter* `const int a[]` *hanyalah pointer, sehingga panjangnya dikirim sebagai* `n`*;* `sizeof loans / sizeof loans[0]` *hanya benar di* `main`*.* `const` *berjanji tidak mengubah array. Buku: Subbab 1.5.5.*

Menelusuri Array: Mencari dan Menghitung

Kode 1.11 Dari `array_ops.cpp`: posisi pertama sebuah nilai, dan banyaknya kemunculan nilai itu.

```
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;
}
```

Pada Kode 1.11, `findFirst` mengembalikan -1 untuk "tidak ditemukan", nilai yang tidak mungkin menjadi indeks sah: kontrak setiap pencarian di Bab 2. Buku: Subbab 1.5.5.

Menelusuri Array: Membalik di Tempat

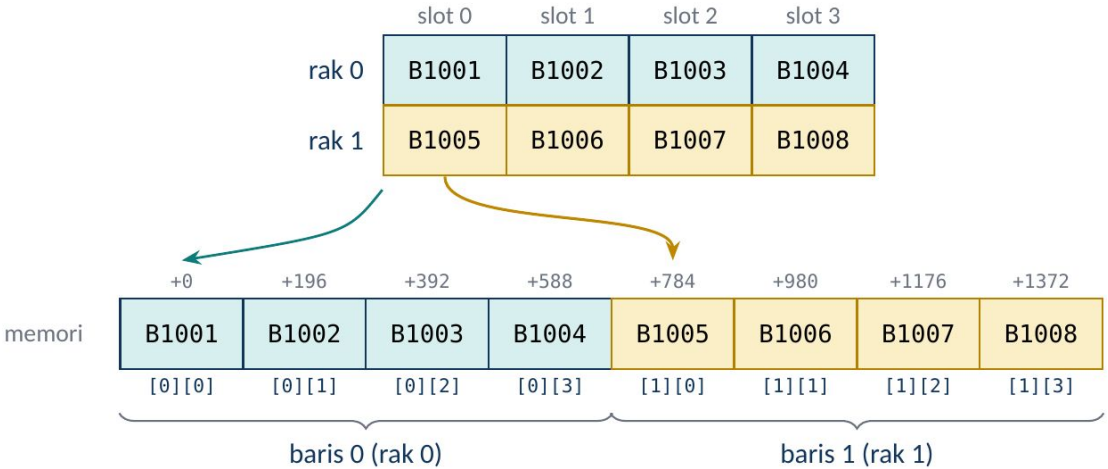
Kode 1.12 Dari `array_ops.cpp`: membalik array dengan dua indeks yang bertemu di tengah.

```
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;
    }
}
```

```
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]
```

Kode 1.12: $\lfloor n/2 \rfloor$ *pertukaran, masing-masing melalui variabel sementara* `t`*. Python menukar dalam satu baris,* `a[i], a[j] = a[j], a[i]`*; fungsi Java dan Python tidak memerlukan* `n`*, karena array mengenal* `a.length` *dan list mengenal* `len(a)`*. Buku: Subbab 1.5.5.*

Array Dua Dimensi, Row-Major



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.

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

Gambar 1.7: C++ menyimpan baris 0 lebih dahulu, lalu baris 1. Elemen $[r][c]$ didahului r baris utuh berisi C elemen dan c elemen dari barisnya sendiri, maka berlaku Persamaan (1.3). Banyaknya baris tidak muncul. Buku: Subbab 1.5.6.

Contoh Terkerjakan 2: Alamat dalam Tabel

`int m[6][9]` mulai di $b = 2000$ dengan $s = 4$; jadi $R = 6$ dan $C = 9$.

Row-major, Persamaan (1.3): `m[4][7]` berada di $2000 + (4 \cdot 9 + 7) \cdot 4 = 2000 + 43 \cdot 4 = 2172$.

Column-major, seperti penyimpanan Fortran, Persamaan (1.4): $2000 + (7 \cdot 6 + 4) \cdot 4 = 2000 + 46 \cdot 4 = 2184$.

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

Elemen mana yang mulai di $2000 + 4 \cdot 50$? Posisi 50 adalah baris $\lfloor 50/9 \rfloor = 5$, kolom $50 - 45 = 5$: `m[5][5]`.

Kode yang menghitung alamat dengan tangan, atau yang mengoper array antarbahasa, harus tahu urutan mana yang dipakai. Buku: Subbab 1.5.6 dan Penyelesaian 1.4.

Rumusnya, Diperiksa oleh Mesin

Kode 1.13 Dari `shelves_2d.cpp`: *offset* terukur setiap `shelf[s][p]` dibandingkan dengan $(s \cdot \text{SLOTS} + p) \cdot \text{sizeof(Book)}$.

```
    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");
        }
    }
```

```
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
```

Kode 1.13 membandingkan setiap offset terukur dengan Persamaan (1.3). `shelf[1][2]` berisi B1007: $(1 \cdot 4 + 2) \cdot 196 = 1176$, persis offset terukurnya. `Book[][]` di Java atau list berisi list di Python adalah array berisi array baris; panjang barisnya bahkan boleh berbeda (jagged). Buku: Subbab 1.5.6.

Tabel Bilangan

Kode 1.14 Dari `loan_table.cpp`: total baris, total kolom, dan total keseluruhan dengan dua perulangan bersarang.

```
    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;
    }
```

```
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
```

Pada Kode 1.14, perulangan luar menelusuri baris, perulangan dalam menelusuri kolom satu baris. `columnTotal` memerlukan satu total berjalan per kolom, maka ia berupa array yang dinolkan dengan `= {0}`. Perulangan berjalan baris demi baris, sesuai urutan elemen di memori; itu cocok dengan cache line 64 byte prosesor. Buku: Subbab 1.5.7.

Jebakan Python: Satu Baris, Tiga Kali

`*` pada list mengulang *referensi*: `[[0] * 4] * 3` adalah tiga referensi ke baris yang **sama** (Kode 1.15).

Kode 1.15 `aliasing_rows.py`: `[[0] * 4] * 3` menghasilkan tiga referensi ke satu baris; *list comprehension* menghasilkan tiga baris.

```python
# 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
```

Idiom yang benar, `[[0] * 4 for _ in range(3)]`, menjalankan `[0] * 4` tiga kali dan membuat tiga baris tersendiri. Buku: Subbab 1.5.7.

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

Array mentah lupa panjangnya di dalam fungsi, tidak dapat disalin dengan `=`, dan ukurannya tetap sejak kompilasi.

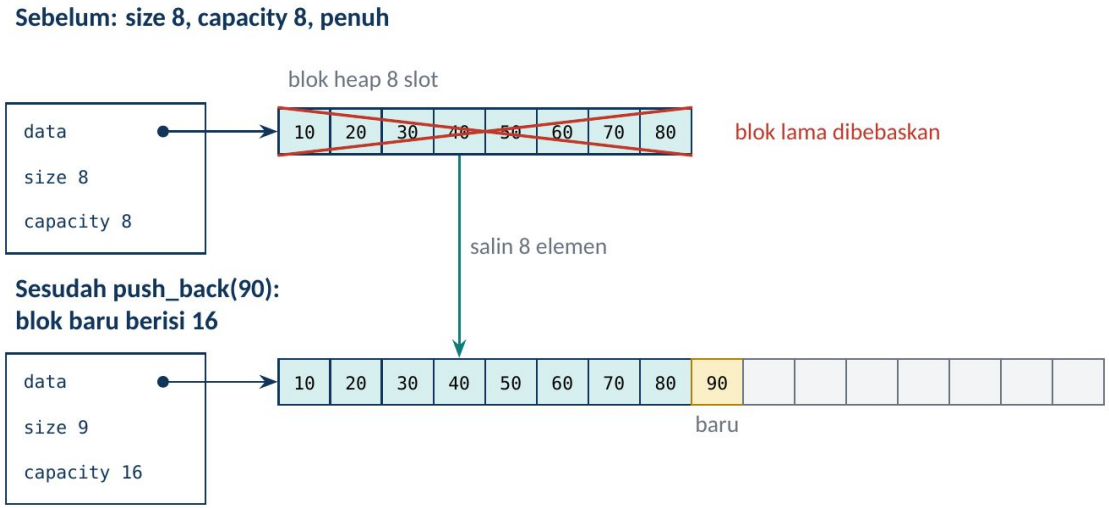
`std::array<T, N>` mengenal `size()`, dapat disalin dan dibandingkan, menyediakan `at(i)`, dan tidak menambah biaya apa pun.

`std::vector<T>` dapat tumbuh: `push_back`, `pop_back`, `size()`, `v[i]`, `v.at(i)`. Ia menyimpan satu blok di heap, sebuah **ukuran** (*size*), dan sebuah **kapasitas**.

Bila bloknya penuh, `push_back` meminta blok yang lebih besar, menyalin setiap elemen, membebaskan blok lama, dan baru kemudian menambahkan elemen baru.

Bila Anda tahu kira-kira berapa elemen yang akan datang, `v.reserve(n)` menetapkan kapasitas sekali saja dan menghindari setiap pertumbuhan ulang. Buku: Subbab 1.5.8.

Satu Kali Tumbuh, Digambarkan



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.

Gambar 1.8: vector berkapasitas 8 menerima elemen kesembilan; blok berisi 16 dialokasikan, 8 elemen disalin, blok lama dibebaskan, lalu 90 ditambahkan. Buku: Subbab 1.5.8.

Contoh Terkerjakan 3: Mengamati Vector Tumbuh

Kode 1.16 Dari `vector_growth.cpp`: 100 kali `push_back`, mencetak satu baris setiap kali kapasitasnya berubah.

```
    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();
        }
    }
```

```
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)
```

Dengan GCC kapasitasnya berlipat dua: 1, 2, 4, …, 128. Pustaka Microsoft mengalikan dengan 1,5; argumen di halaman berikut berlaku untuk faktor tetap mana pun. Buku: Subbab 1.5.8.

Mengapa Melipatduakan Itu Murah

Untuk mencapai n elemen, vector tumbuh pada ukuran $1, 2, 4, \ldots$ sampai pangkat terbesar $2^k < n$, dan setiap kali menyalin sebanyak itu:

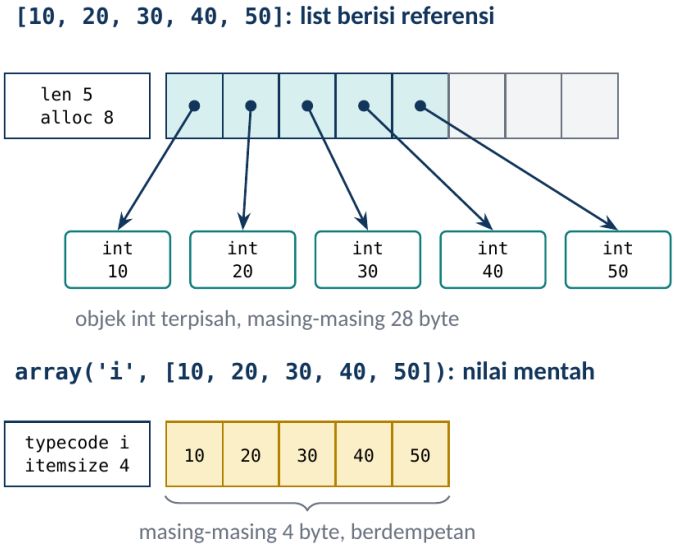
$$1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 < 2n \tag{1.5}$$

Untuk $n = 100$: $1 + 2 + 4 + 8 + 16 + 32 + 64 = 127 < 200$, tepat hitungan pada Kode 1.16.

Tumbuh satu slot setiap kali akan menyalin $n(n-1)/2$: hampir 5.000 untuk $n = 100$, lima miliar untuk $n = 100.000$.

Persamaan (1.5): dalam jangka panjang, kurang dari dua salinan per `push_back`*, sebuah konstanta.* `ArrayList` *Java tumbuh dengan faktor 1,5 dari kapasitas awal 10 (202 salinan untuk 100 elemen, di bawah* $3n$*). Buku: Subbab 1.5.8 dan 1.5.9.*

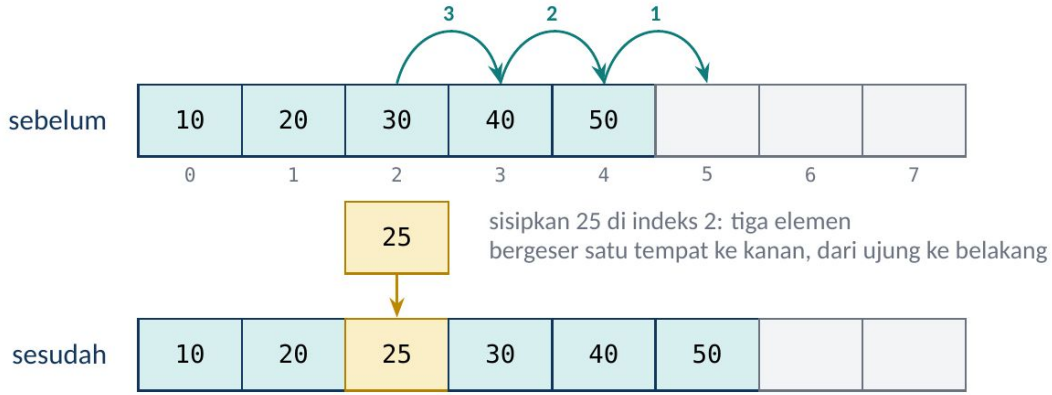
List Python dan Modul `array`



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

Gambar 1.9: list menyimpan referensi delapan byte ke objek bilangan bulat 28 byte; `array('i')` memadatkan nilai mentah 4 byte. `ArrayList<Integer>` di Java juga menyimpan referensi. Buku: Subbab 1.5.9 dan 1.5.10.

Penyisipan: Memberi Ruang



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.

Gambar 1.10: untuk menyisipkan 25 di indeks 2, geser 50, lalu 40, lalu 30 satu tempat ke kanan, dari ujung ke belakang; menggeser dari depan akan menimpa 40 sebelum sempat diselamatkan. Buku: Subbab 1.6.3.

Penyisipan dan Penghapusan, Secara Tepat

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
```

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
```

Pseudokode 1.1 berjalan dari ujung mundur ke `pos`; Pseudokode 1.2 adalah bayangan cerminnya, menggeser elemen ke kiri dari celah ke depan. Buku: Subbab 1.6.3.

Menghitung Pergeseran

Baris 2 pada Pseudokode 1.1 dijalankan sekali untuk setiap i dari n turun sampai $pos+1$, seperti dinyatakan Persamaan (1.6):

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

Di ujung, $pos = n$: tanpa pergeseran. Di depan, $pos = 0$: semua n. Dirata-ratakan atas $n+1$ posisi, Persamaan (1.7):

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

Tulis $0+1+\cdots+n$ maju dan mundur lalu jumlahkan kolom demi kolom: $n+1$ kolom bernilai n, jadi dua kali jumlahnya adalah $n(n+1)$. Penghapusan menggeser $n-1-pos$ elemen, rata-rata $(n-1)/2$. Buku: Subbab 1.6.3.

Contoh Terkerjakan 4: Penyisipan, Dihitung

Kode 1.17 Dari `insert_shift.cpp`: penyisipan dengan menggeser, mengembalikan banyaknya pergeseran.

```cpp
// 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;
}
```

```
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
```

Kode 1.17 adalah Pseudokode 1.1 dengan penghitung, dijalankan pada array berisi 10 sampai 10.000 elemen. Tabelnya adalah Persamaan (1.6) yang sedang bekerja: kalikan ukurannya dengan sepuluh, maka kerja di depan atau di tengah ikut berlipat sepuluh; hanya ujung yang gratis. `v.insert`, `list.add(pos, x)`, *dan* `lst.insert(pos, x)` *melakukan penggeseran yang sama di balik satu baris. Buku: Subbab 1.6.3.*

Big-O, Secara Informal

DEFINISI. Kita menulis $f(n) = O(g(n))$ bila ada konstanta positif c dan n_0 sehingga Persamaan (1.8) berlaku: sejak ukuran tertentu, f tumbuh tidak lebih cepat daripada suatu kelipatan konstan dari g.

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

Definisi ini mengabaikan **faktor konstan** ($n/2$ dan n sama-sama $O(n)$) dan **masukan kecil**; yang tersisa adalah *bentuk* pertumbuhannya.

$n/2 \le 1 \cdot n$ untuk $n \ge 1$, jadi $n/2 = O(n)$. Perulangan penjumlahan memerlukan sekitar $3n + 2 \le 5n$ langkah untuk $n \ge 1$: $O(n)$.

Dua bentuk hari ini: $O(1)$, konstan, dan $O(n)$, linear — lipatduakan n dan kerjanya ikut berlipat dua. Bab 2 mendefinisikan notasi ini dengan ketat. Buku: Subbab 1.6.4.

Biaya Operasi Array

Tabel 1.4 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)

*Tabel 1.4 adalah yang pertama dari banyak tabel serupa; setiap struktur data adalah pilihan tentang baris mana yang dibuat murah. Baris terakhir adalah $O(1)$ **teramortisasi**: jaminan atas setiap rangkaian operasi yang panjang, bukan "biasanya cepat". Buku: Subbab 1.6.4 dan 1.6.5.*

Program Pertama Anda

Kode 1.18 `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;
}
```

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

Meja kerja: VS Code dengan ekstensi C/C++, Java, dan Python, GCC (MSYS2 UCRT64 di Windows), JDK 25, Python 3.14. Kode 1.18 dikompilasi dengan `g++ -std=c++17 -O2 -Wall -Wextra -o hello hello.cpp`; `201703` *adalah nilai* `__cplusplus` *di bawah C++17. Buku: Subbab 1.7.7 dan 1.7.8.*

Profitina dalam Praktik

Satu deret skor tetap per halaman: pemindaian situs menilai setiap halaman terhadap daftar sinyal yang tetap; slot k selalu berarti sinyal k, dibaca dalam $O(1)$.

Tabel halaman kali sinyal: total baris menunjukkan kinerja satu halaman, total kolom kinerja seluruh situs pada satu sinyal — bentuk tabel peminjaman.

Riwayat yang hanya tumbuh di ujung: deret skor mingguan adalah array yang dapat tumbuh; menambah di ujung $O(1)$ teramortisasi, dan tidak ada yang pernah disisipkan di tengah.

Gagal dengan lantang: bila model AI lokal di balik Content Engine tidak tersedia, aplikasi mengatakannya — prinsip `at()` dalam skala terkecil.

Umpan yang terbaru-lebih-dahulu menambah di ujung dan membaca mundur, alih-alih menyisipkan di indeks 0 dan membayar n pergeseran setiap kali. Buku: Subbab 1.8.

Kesalahan yang Sering Terjadi

`i <= n` atau mulai dari 1: mengunjungi `a[n]` yang tidak ada dan melewatkan `a[0]`. Kondisinya selalu `i < n`.

Array yang belum diisi, atau `sizeof` **di dalam fungsi** untuk mencari panjang: byte sisa, dan `8/4=2` berapa pun panjang sebenarnya.

Kehilangan nilai saat menukar, atau **menggeser ke arah yang salah**: pakai variabel sementara; sisipkan dari ujung ke belakang.

Baris dan kolom tertukar; "array 2-D" Java atau Python dikira satu blok; `[[0] * c] * r` di Python.

Referensi disalin lalu dianggap salinan; menyisipkan di depan dalam perulangan ($n^2/2$ pergeseran); luapan dan pembagian bilangan bulat; `delete` yang terlupa.

Setiap kesalahan di buku disertai perbaikannya. Buku: Subbab 1.9.

Ringkasan: Lima Hal yang Perlu Diingat

1. Struktur data adalah penataan data beserta operasinya; pilihan yang tepat mengubah n langkah menjadi $\log_2 n$ atau 1 (Tabel 1.1).

2. Memori adalah deret byte beralamat; tipe menetapkan berapa byte dan bagaimana membacanya; variabel lokal di stack, hasil `new` di heap.

3. Elemen i sebuah array berada di $b + i \cdot s$ (Persamaan (1.2)), sehingga aksesnya $O(1)$; tabel C++ disimpan *row-major* (Persamaan (1.3)).

4. Penyisipan atau penghapusan di tengah menggeser $n - pos$ elemen, $O(n)$; menambah di ujung vector yang berlipat dua adalah $O(1)$ teramortisasi (Persamaan (1.5)).

5. C++ tidak memeriksa batas (`at()` memeriksa); Java dan Python selalu memeriksa, dan Python menerima indeks negatif.

Meja kerjanya adalah VS Code dengan C++, Java, dan Python, diperiksa dengan perintah-perintah verifikasi di buku. Buku: Subbab 1.10.

Pertemuan Berikutnya

Bab 2 — Pencarian. Pencarian sekuensial, sentinel, pencarian biner dan interpolasi, serta notasi *O* secara lengkap.

Kontrak “sebuah posisi, atau −1” milik `findFirst` menjadi kontrak setiap pencarian.

Sebelum itu: pasang meja kerja, jalankan `hello` dalam ketiga bahasa, dan kerjakan Soal Latihan 1.3, 1.4, 1.7, dan 1.9.

Katalog Pustaka Ceria yang terurut menurut kode adalah tempat pencarian biner membagi dua jalannya menuju sebuah buku dalam tujuh belas kali lihat di antara delapan puluh ribu. Buku: Subbab 1.11 dan 1.12.

Kuliah 2 — Pencarian

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 2

Dosen: Irfan Subakti — yifana@gmail.com

Hari ini kita menjawab pertanyaan tertua yang pernah diajukan kepada sebuah koleksi — adakah di sini, dan di mana? — dengan enam cara, dan belajar memilih di antaranya berdasarkan angka, bukan tebakan. Buku: Bab 2, Subbab 2.1–2.15.

Cakupan Pertemuan Ini

1. **Masalahnya dan satuan biayanya** — sebuah posisi atau -1; pemeriksaan; kasus terbaik, terburuk, dan rata-rata.
2. **Pencarian sekuensial** — dengan dan tanpa sentinel, dan berapa tepatnya biaya masing-masing.
3. **Pencarian biner** — jejaknya, invariannya, buktinya, $\lfloor \lg n \rfloor + 1$ pemeriksaan, bentuk rekursifnya.
4. **Jebakan meleset satu** — interval tertutup dan setengah terbuka, *lower bound* dan *upper bound*, titik tengah yang meluap.
5. **Pencarian interpolasi, lompat, dan dua dimensi** — dan apa kata katalog nyata berisi 10.000 buku tentang semuanya.
6. **Pustaka standar, batas bawah pencarian dengan perbandingan, Profitina Laundry.**

Satu gagasan mengalir sepanjang pertemuan: makin banyak yang Anda ketahui tentang susunan data, makin sedikit bagian yang perlu Anda lihat. Buku: Subbab 2.1.

Dari Mana Kita Melanjutkan

Bab 1 memberi kita meja kerja dalam tiga bahasa dan gambaran memori: sel i sebuah larik berada di alamat Persamaan (2.1).

$$address(A[i]) = base + i \cdot w, \tag{2.1}$$

Jadi **setiap sel dicapai dalam satu langkah**. Setiap pencarian hari ini adalah strategi untuk memutuskan sel *mana* yang dilihat.

Buku-buku Pustaka Ceria kini tersimpan dalam larik. Seorang peminjam menanyakan sebuah kode: adakah bukunya, dan di mana?

Pencarian adalah operasi yang paling sering dijalankan dalam komputasi: setiap masuk log, setiap pemindaian kode batang, setiap mesin jawab yang mengutip halaman. Buku: Subbab 2.1.

Masalah Pencarian

> **DEFINISI.** Diberikan n rekaman $A[0], \ldots, A[n-1]$ dan kunci pencarian x, kembalikan indeks i yang kuncinya sama dengan x, atau laporkan bahwa indeks seperti itu tidak ada. Pencarian yang menemukan kunci disebut **berhasil**; yang tidak, **gagal**.

Kembalikan **posisi**, jangan hanya FOUND: pemanggil memerlukannya untuk mencetak, meminjamkan, atau memindahkan buku. "Tidak ada" adalah -1, yang tidak pernah menjadi indeks sah.

Satuan biayanya adalah **pemeriksaan**: satu kali melihat satu elemen.

Kita menanyakan kasus **terbaik**, **terburuk** (jaminannya), dan **rata-rata**, lalu meringkasnya dengan $O(\cdot)$; $\lg n$ berarti $\log_2 n$.

$\lg 8 = 3$ karena $8 \to 4 \to 2 \to 1$ memerlukan tiga kali pembagian dua. $O(\lg n)$ bertambah satu setiap kali n berlipat dua. Buku: Subbab 2.2.1 dan 2.2.2.

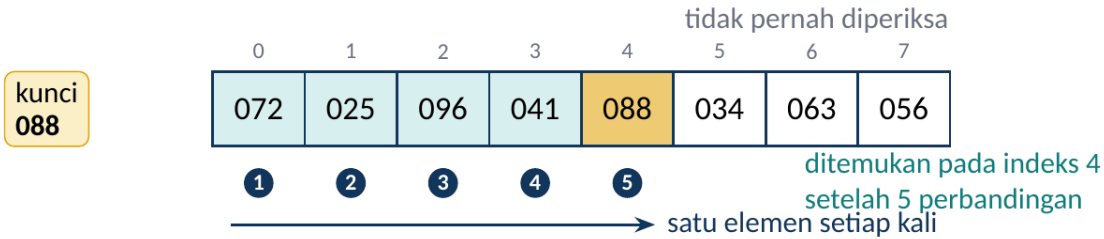
Rak Komputer Pustaka Ceria

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

Tabel 2.1 *adalah contoh berjalan kita. Buku datang dalam* ***urutan kedatangan****, 072, 025, 096, 041, 088, 034, 063, 056; urutan terurut baru ada setelah seseorang mengurutkannya, pokok bahasan Bab 3. Pencarian mana yang mungkin bergantung pada urutan mana yang kita miliki. Buku: Subbab 2.2.3.*

Pencarian Sekuensial



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.

***Gambar 2.1**: tanpa urutan yang dapat dimanfaatkan, lihat kartu pertama, lalu kedua, dan berhenti pada kecocokan atau di ujung. Cara ini bekerja pada larik apa pun, itulah kekuatannya; ia tidak dapat melewatkan apa pun, itulah kelemahannya. Buku: Subbab 2.3.1.*

Prosedurnya, dan Penanda Lawan Indeks

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
```

Versi pertama yang ditulis kebanyakan orang memasang **penanda** dan berjalan sampai ujung: selalu n perbandingan, dan ia lupa *di mana* kunci ditemukan.

Pseudokode 2.1 menyembuhkan keduanya: kembali dari dalam gelung, dan kembalikan indeksnya.

Berhenti lebih awal untuk keberadaan, jangan pernah untuk menghitung: "berapa kali 041 dipinjam?" harus melihat setiap entri.

Log peminjaman 041, 088, 025, 041, 063, 041, 096, 088 memuat buku 041 tiga kali, di posisi 0, 3, dan 5. Buku: Subbab 2.3.1 dan 2.3.2.

Pencarian Sekuensial dalam C++

Kode 2.1 Dari `sequential_search.cpp`: keluar lebih awal dengan indeksnya, dan gelung penghitung yang harus berjalan sampai ujung.

```cpp
// 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 *pada* [8,10,6,−2,11,7,1,100]*: kunci 8 memerlukan 1 perbandingan, 6 memerlukan 3, 100 memerlukan 8, dan 5 yang tidak ada memerlukan kedelapannya. Pada rak dalam urutan kedatangan, kode 088 memerlukan 5, persis seperti diramalkan* ***Gambar 2.1****. Buku: Subbab 2.3.3.*

Berapa Biaya Pencarian Sekuensial?

Kasus terbaik: kunci ada di $A[0]$: satu pemeriksaan, $O(1)$. **Kasus terburuk:** di ujung atau tidak ada: n pemeriksaan, $O(n)$.

Rata-rata, berhasil, setiap posisi berpeluang sama, Persamaan (2.2):

$$C_{\text{succ}}(n) = \frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}\cdot\frac{n(n+1)}{2} = \frac{n+1}{2}. \tag{2.2}$$

Rata-rata bila kunci ada dengan peluang p, Persamaan (2.3):

$$C(n,p) = p\cdot\frac{n+1}{2} + (1-p)\cdot n. \tag{2.3}$$

Delapan buku: rata-rata 4,5 pemeriksaan; sepuluh ribu: 5.000,5. Pada $p = 1/2$ sekitar $3/4\,n$. Lipatduakan katalognya dan rata-rata pencarian ikut berlipat dua: itulah arti $O(n)$ dalam praktik. Buku: Subbab 2.3.4.

Kapan Pencarian Sekuensial Alat yang Tepat

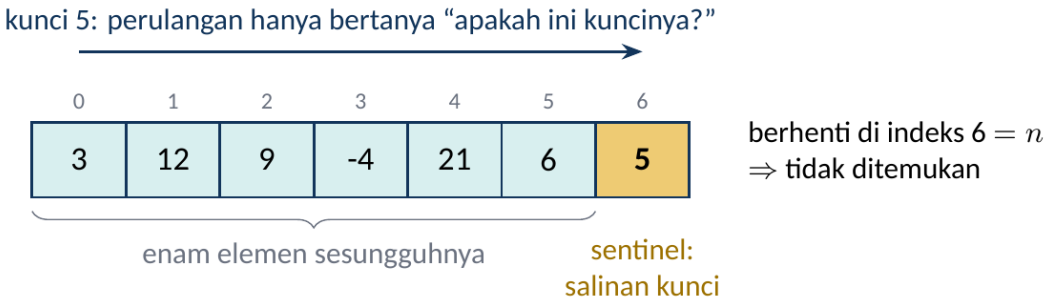
> **WAWASAN KUNCI.** Pencarian sekuensial adalah satu-satunya pencarian dalam bab ini yang bekerja pada data **tidak terurut**, dan pada data yang hanya dapat dibaca dari depan ke belakang — berkas, aliran data, *linked list*. Bila datanya kecil, dicari sekali saja, atau tidak dapat diurutkan, inilah alat yang tepat. Bila datanya besar dan sering dicari, inilah alat yang harus diganti.

Jumlah $1 + 2 + \cdots + n = n(n+1)/2$: tulis maju dan mundur lalu jumlahkan kolom demi kolom.

Pencarian sekuensial yang gagal *selalu* memerlukan tepat n pemeriksaan.

Bab 7 mencari di dalam linked list, tempat pencarian sekuensial menjadi satu-satunya pilihan. Buku: Subbab 2.3.4.

Sentinel: Satu Uji Padahal Tadinya Dua



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.

***Gambar 2.2**: setiap langkah **Pseudokode 2.1** bertanya “apakah $i < n$?” dan “apakah $A[i]$ kuncinya?”. Tanam salinan kunci di sel cadangan berindeks n, maka gelung pasti berhenti; sesudahnya, $i < n$ berarti ditemukan dan $i = n$ berarti tidak ada. Buku: Subbab 2.4.1.*

SENTINEL-SEARCH

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
```

Pseudokode 2.2 *menyimpan isi sel cadangan lebih dahulu dan memulihkannya sebelum kembali, sehingga larik ditinggalkan seperti semula. Buku: Subbab 2.4.1.*

Menghitung Uji

Kode 2.2 Dari `sentinel_search.cpp`: gelung sentinel, dengan satu-satunya uji yang dihitung.

```cpp
// 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;
}
```

```
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)
```

***Kode 2.2**: untuk 5 yang tidak ada, gelung biasa melakukan $2n + 1 = 13$ uji, gelung sentinel $n + 1 = 7$. Sentinel memangkas uji, bukan pemeriksaan, dan keduanya tetap $O(n)$. Ia memerlukan sel cadangan yang dapat ditulisi dan tidak boleh dipakai dua thread pada satu larik. Pada katalog nyata ia tidak berpengaruh di C++ maupun Java dan menghemat sekitar 9% di Python. Buku: Subbab 2.4.2 dan 2.4.3.*

Pencarian Biner: Memanfaatkan Urutan

Pada rak yang terurut, buka bagian tengah: 056. Setiap buku di kirinya berkode lebih kecil, jadi 088 tidak mungkin di sana. **Separuh rak lenyap dalam satu kali lihat.**

Begitulah cara Anda mencari kata di kamus kertas.

> **DEFINISI. Pencarian biner** mencari pada larik *terurut* $A[0..n-1]$ dengan menjaga interval $A[low..high]$ yang pasti memuat x bila x ada. Ia memeriksa $A[mid]$: sama, ditemukan; lebih kecil, interval menjadi $A[mid+1..high]$; lebih besar, $A[low..mid-1]$. Interval kosong ($low > high$) berarti tidak ada.

Pencarian biner adalah algoritma terpenting dalam bab ini. Buku: Subbab 2.5.1.

Jejak dengan Tangan



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.

***Gambar 2.3**: tiga pemeriksaan menemukan 17 dalam* [3,9,11,12,15,17,23,31,35]*; pencarian sekuensial memerlukan enam. Buku: Subbab 2.5.2.*

Contoh Terkerjakan 1: Mencari 17, lalu 10

17: $low = 0$, $high = 8$, $mid = 4$, $A[4] = 15 < 17$, jadi $low \leftarrow 5$. Lalu $mid = 6$, $A[6] = 23 > 17$, jadi $high \leftarrow 5$. Lalu $mid = 5$: ditemukan.

10: $mid = 4$, $15 > 10$, $high \leftarrow 3$; $mid = 1$, $9 < 10$, $low \leftarrow 2$; $mid = 2$, $11 > 10$, $high \leftarrow 1$.

Kini $low = 2 > high = 1$: kosong, 10 tidak ada — dan $low = 2$ tepat merupakan tempat 10 akan disisipkan.

```
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
```

Program mencetak jejak tangan itu baris demi baris. Titik sisip yang tertinggal di low adalah benih lower bound yang kita bahas nanti. Buku: Subbab 2.5.2 dan 2.5.6.

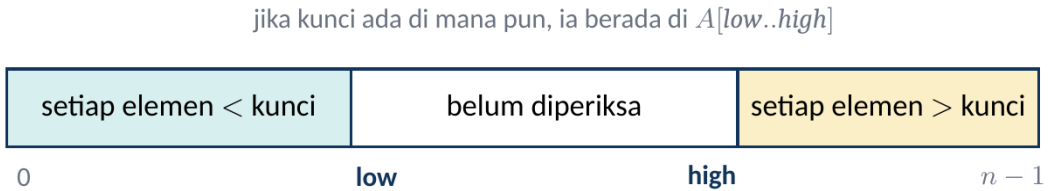
BINARY-SEARCH

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
```

Pseudokode 2.3 *bekerja pada interval tertutup* $[low, high]$. *Baris 4 menghitung titik tengah sebagai* $low + \lfloor (high - low)/2 \rfloor$, *dengan alasan yang segera kita temui. Pencarian biner pertama terbit pada 1946; yang benar untuk setiap n baru muncul pada 1962. Buku: Subbab 2.5.3.*

Invarian Gelung



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.

***Gambar 2.4**: pada awal setiap iterasi, semua isi $A[0..low-1]$ lebih kecil daripada x dan semua isi $A[high+1..n-1]$ lebih besar; jadi bila x ada, ia berada di $A[low..high]$. Buku: Subbab 2.5.3.*

Membuktikan Pencarian Biner Benar

Inisialisasi. $low = 0$, $high = n - 1$: kedua daerah luar kosong, dan pernyataan tentang setiap elemen daerah kosong selalu benar.

Pemeliharaan. Bila $A[mid] < x$, keterurutan membuat seluruh $A[low..mid]$ lebih kecil daripada x, sehingga $low \leftarrow mid + 1$ menjaga invarian. Kasus lainnya adalah bayangan cerminnya.

Terminasi. Baris 6 mengembalikan kecocokan yang sungguhan, atau $low > high$: daerah tengah kosong, x tidak ada, dan low adalah titik sisipnya.

Gelungnya berakhir. $high - low + 1$ adalah bilangan bulat tak negatif yang mengecil pada setiap iterasi yang tidak kembali.

Penangkal bug pencarian biner bukanlah kecerdikan, melainkan invarian. Di kelas-kelas Bentley, hanya sekitar satu dari sepuluh pemrogram profesional yang menulis pencarian biner yang benar. Buku: Subbab 2.5.3.

Berapa Banyak Pemeriksaan?

Interval berisi m elemen menyisakan paling banyak $\lfloor m/2 \rfloor$ setelah satu pemeriksaan, sehingga diperoleh rekurens Persamaan (2.4):

$$T(0) = 0, \qquad T(m) = 1 + T(\lfloor m/2 \rfloor) \quad (m \geq 1). \tag{2.4}$$

Dibuka: ukurannya $n, \lfloor n/2 \rfloor, \lfloor n/4 \rfloor, \ldots$, selama $2^k \leq n$, sehingga Persamaan (2.5):

$$T(n) = \lfloor \lg n \rfloor + 1. \tag{2.5}$$

CONTOH. $n = 9$: paling banyak $\lfloor 3{,}17 \rfloor + 1 = 4$ pemeriksaan. $n = 10^4$: 14. Sejuta: 20. Semiliar: 30 — dibandingkan semiliar untuk pencarian sekuensial.

Melipatduakan n menambah satu pemeriksaan pada pencarian biner dan melipatduakan pencarian sekuensial. Pencarian yang berhasil rata-rata memerlukan sekitar $\lg n - 1$ pemeriksaan; pada 10^4 kunci kita mengukur 12,4. Buku: Subbab 2.5.4.

Pencarian Biner secara Rekursif

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)
```

Pseudokode 2.4*: benar untuk interval ini bila benar untuk setiap interval yang lebih pendek — bukti dengan induksi, tema Bab 10. Paling banyak* $\lfloor \lg n \rfloor + 2$ *stack frame; pemanggilan ekor dapat diubah secara mekanis menjadi gelung. Buku: Subbab 2.5.5.*

Pemanggilan Rekursif, Dihitung

Kode 2.3 Dari `binary_search.cpp`: bentuk rekursif, dengan menghitung pemanggilan.

```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);
}
```

```
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
```

***Kode 2.3**: pencarian yang gagal melakukan satu pemanggilan terakhir pada interval kosong yang tidak memeriksa apa pun, sehingga 10 memerlukan 3 pemeriksaan dalam 4 pemanggilan; 40 memerlukan 4 pemeriksaan, maksimum yang dijanjikan Persamaan (2.5) untuk $n = 9$, dalam 5 pemanggilan. Buku: Subbab 2.5.6.*

Interval Tertutup dan Setengah Terbuka

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	$high - low + 1$	$high - low$

***Tabel 2.2**: kedua konvensi benar; mencampurnya tidak, dan hampir setiap bug pencarian biner adalah campuran. Setengah terbuka punya dua keunggulan kecil: banyaknya elemen adalah high − low, dan larik kosong tidak memerlukan kasus khusus. Buku: Subbab 2.6.1.*

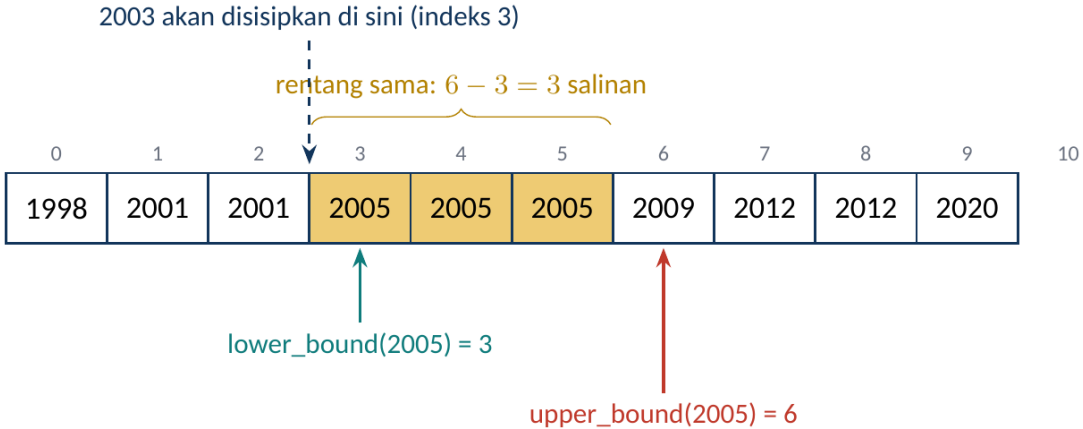
Empat Jebakan

1. **Syarat gelung yang salah.** Tertutup dengan `low < high` tidak pernah memeriksa elemen terakhir: mencari 5 dalam [5] gagal. Setengah terbuka dengan `low <= high` memeriksa $A[n]$.
2. **Pembaruan yang tidak menyusutkan.** `low = mid` pada interval dua elemen tidak mengubah apa pun: gelung berjalan selamanya.
3. **Titik tengah yang meluap.** `(low + high) / 2` dalam aritmetika 32 bit.
4. **Mengabaikan duplikat.** **Pseudokode 2.3** mengembalikan *salah satu* salinan kunci yang berulang, dan yang mana bergantung pada n.

Semuanya pernah muncul dalam kode yang diterbitkan; langkah pemeliharaan dalam bukti kita gagal tepat pada jebakan 2. Buku: Subbab 2.6.2.

Lower Bound dan *Upper Bound*

DEFINISI. Untuk $A[0..n-1]$ terurut, ***lower bound*** adalah i terkecil dengan $A[i] \geq x$, dan ***upper bound*** adalah i terkecil dengan $A[i] > x$ (keduanya n bila tidak ada). Salinan-salinan x menempati $A[lower..upper-1]$; banyaknya $upper - lower$; bila x tidak ada, *lower bound* adalah tempat ia harus disisipkan.



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.

***Gambar 2.5**: ketiga salinan 2005 menempati* [3,6)*; untuk 2003 yang tidak ada, kedua batasnya 3. Buku: Subbab 2.6.3.*

LOWER-BOUND

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)
```

> **WAWASAN KUNCI.** *Lower bound* dan *upper bound* menjawab "adakah?" ($upper > lower$), "salinan pertama?" ($lower$), "terakhir?" ($upper - 1$), "berapa banyak?" ($upper - lower$), "di mana tempatnya?" ($lower$), dan hitungan rentang $lower(b) - lower(a)$.

Pseudokode 2.5 *tidak pernah menguji kesamaan. Invariannya: $A[0..low-1]$ semuanya lebih kecil daripada x dan $A[high..n-1]$ semuanya paling sedikit x; ketika $low = high$, daerah yang belum diketahui lenyap. Paling banyak $\lceil \lg(n+1) \rceil$ pemeriksaan, dan hampir sebanyak itu setiap kali. Buku: Subbab 2.6.3.*

Lower Bound dalam C++

Kode 2.4 Dari `bounds.cpp`: *lower bound* pada interval setengah terbuka; *upper bound* hanya berbeda pada `<=`.

```cpp
// 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;
}
```

```
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
```

Kode 2.4 *adalah pencarian biner yang pertama-tama perlu Anda raih. Buku: Subbab 2.6.3 dan 2.6.5.*

Titik Tengah yang Meluap

Pada 2006 Joshua Bloch melaporkan bahwa `int mid = (low + high) / 2;` dalam pustaka Java telah salah selama sekitar dua dasawarsa.

Bila $low + high > 2^{31} - 1 = 2.147.483.647$, jumlahnya melingkar menjadi bilangan negatif, begitu pula indeksnya.

Perbaikannya, Persamaan (2.6), tidak pernah membentuk bilangan yang lebih besar daripada $high$:

$$mid = low + \left\lfloor \frac{high - low}{2} \right\rfloor, \tag{2.6}$$

```
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
```

Di C++ luapan bilangan bertanda adalah perilaku tak terdefinisi, maka `bounds.cpp` menghitung dalam 64 bit apa yang akan disimpan `int` 32 bit; Java meluapkan `int` sungguhan; bilangan bulat Python tidak pernah meluap, tetapi kebiasaan baik ini tidak memakan biaya. Buku: Subbab 2.6.4.

Pencarian Interpolasi: Tebak Letaknya

Mencari *Tanaka* di buku telepon, Anda membukanya kira-kira tiga perempat jalan: Anda memakai **nilai**, bukan hanya urutan.

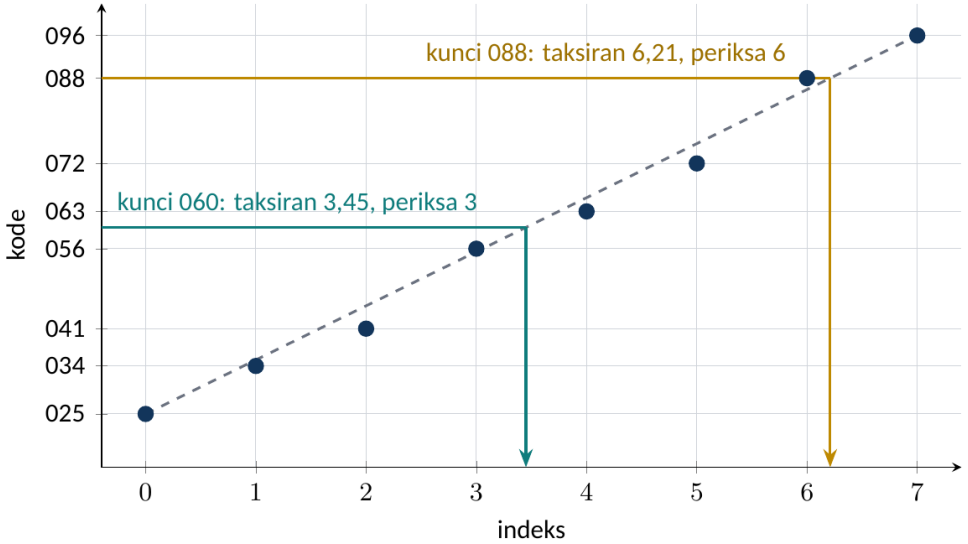
Bila kunci naik lurus dari $A[low]$ ke $A[high]$, kunci berada pada posisi Persamaan (2.7):

$$pos = low + \left\lfloor \frac{x - A[low]}{A[high] - A[low]} \cdot (high - low) \right\rfloor. \tag{2.7}$$

$x = A[low]$ memberi low; $x = A[high]$ memberi $high$. Pencarian biner selalu memakai pecahan $1/2$.

Gagasan ini berasal dari Peterson pada 1957. Buku: Subbab 2.7.1.

Rak Itu Hampir Berupa Garis Lurus



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.

***Gambar 2.6**: kedelapan kode terletak dekat garis putus-putus dari* (0,025) *ke* (7,096) *— tepat seperti andaian Persamaan (2.7). Buku: Subbab 2.7.2.*

Contoh Terkerjakan 2: Interpolasi dengan Tangan

Kunci 088, $low = 0, high = 7$:

$$pos = 0 + \left\lfloor \frac{88 - 25}{96 - 25} \cdot 7 \right\rfloor = \left\lfloor \frac{63}{71} \cdot 7 \right\rfloor = \lfloor 6{,}21 \rfloor = 6$$

$A[6] = 088$: ditemukan dengan **satu** pemeriksaan; pencarian biner memerlukan tiga.

Kunci 060: $\lfloor (35/71) \cdot 7 \rfloor = \lfloor 3{,}45 \rfloor = 3$; $A[3] = 056 < 060$, jadi $low \leftarrow 4$. Kini $060 < A[4] = 063$: di luar interval, tidak ada setelah satu pemeriksaan.

Pemeriksaan rentang $A[low] \le x \le A[high]$ menjaga taksiran tetap di dalam larik; penjaga kedua mencegah pembagian dengan nol bila $A[low] = A[high]$. Buku: Subbab 2.7.2.

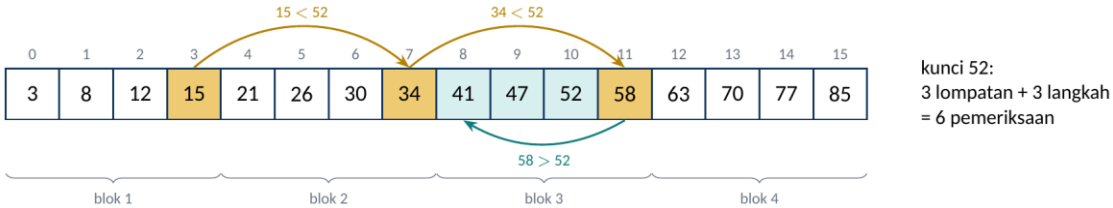
INTERPOLATION-SEARCH

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
```

***Pseudokode 2.6**: pada kunci yang tersebar merata, rata-ratanya sekitar lglgn pemeriksaan (4,26 pada sejuta kunci, dibandingkan 18,8 untuk pencarian biner); pada kunci yang menggerombol ia merosot menjadi $O(n)$. Setiap langkah memerlukan pembagian, dan kuncinya harus bilangan. Buku: Subbab 2.7.2–2.7.4.*

Pencarian Lompat: Melompat per Blok



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.

***Gambar 2.7**: lompat m tempat sekaligus, memeriksa elemen terakhir setiap blok (15, 34, 58); karena $58 > 52$, kunci ada di blok 3, dan berjalan singkat melalui 41, 47, 52 menemukannya. Berguna bila bergerak mundur itu mahal: pita, blok berkas yang besar. Buku: Subbab 2.8.1.*

JUMP-SEARCH

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
```

***Pseudokode 2.7**: program-programnya juga langsung berhenti bila elemen terakhir sebuah blok sama dengan kunci, dan langkah berjalannya melewati elemen itu. Buku: Subbab 2.8.1.*

Memilih Ukuran Blok

Sekitar n/m lompatan lalu paling banyak $m - 1$ langkah memberi kasus terburuk Persamaan (2.8):

$$f(m) = \frac{n}{m} + m - 1. \tag{2.8}$$

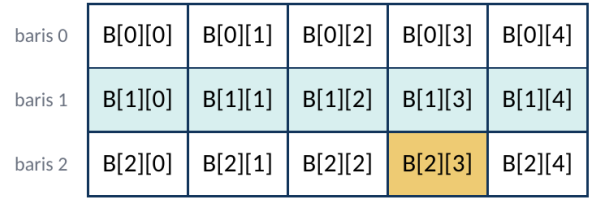
$f'(m) = -n/m^2 + 1 = 0$ memberi keseimbangan Persamaan (2.9):

$$m^* = \sqrt{n}, \qquad f(m^*) = 2\sqrt{n} - 1. \tag{2.9}$$

```
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
```

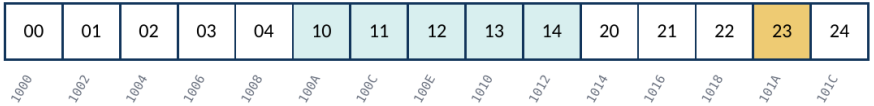
Tanpa kalkulus: n/m dan m memiliki hasil kali tetap n, dan dua bilangan positif dengan hasil kali tetap memiliki jumlah terkecil bila keduanya sama. Jauh lebih baik daripada n, jauh lebih buruk daripada $\lfloor \lg n \rfloor + 1$. Buku: Subbab 2.8.2 dan 2.8.3.

Larik Dua Dimensi di Memori



$$\&B[2][3] = \texttt{0x1000} + (2 \cdot 5 + 3) \cdot 2 = \texttt{0x1000} + 26 = \texttt{0x101A}$$

15 elemen yang sama di memori, baris demi baris (int 2 byte, alamat dasar 0x1000):



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.

$$address(B[i][j]) = base + (i \cdot C + j) \cdot w, \tag{2.10}$$

***Gambar 2.8**: C dan C++ menyimpan kisi baris demi baris. Persamaan (2.10) mengubah $B[i][j]$ menjadi elemen ke-$(i \cdot C + j)$ lalu menerapkan Persamaan (2.1). Fortran, MATLAB, dan R memakai urutan kolom-mayor. Buku: Subbab 2.9.1.*

Contoh Terkerjakan 3: Dua Alamat

`int A[10]` di 0x1000, `int` berukuran 2 byte: $\&A[7] = 0\text{x}1000 + 7 \cdot 2 = 0\text{x}1000 + 14 = 0\text{x}100\text{E}$.

`int B[3][5]` di 0x1000: elemen ke-$(2 \cdot 5 + 3) = 13$, *offset* byte $13 \cdot 2 = 26 = 0\text{x}1\text{A}$, jadi $\&B[2][3] = 0\text{x}101\text{A}$.

Dengan `int` 4 byte masa kini: $0\text{x}1000 + 28 = 0\text{x}101\text{C}$ dan $0\text{x}1000 + 52 = 0\text{x}1034$.

> **KELIRU YANG LAZIM.** Menambahkan *banyaknya* elemen alih-alih *offset* byte, $0\text{x}1000 + 7$, melupakan faktor w.

`search_2d.cpp` mencetak keempat alamat ini persis. Buku: Subbab 2.9.1.

Tiga Cara Mencari di Kisi

Kisi tidak terurut: telusuri baris demi baris, $O(RC)$ pemeriksaan, sesuai urutan memori.

Baris dan kolom sama-sama terurut: metode tangga, $O(R + C)$, **Pseudokode 2.8**.

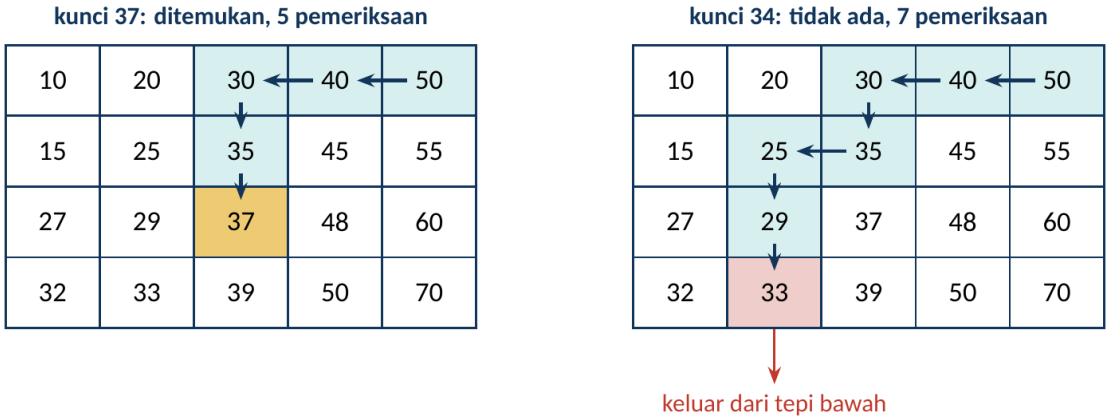
Seluruh kisi terurut baris-mayor: satu pencarian biner atas $0..RC - 1$, dengan $r = \lfloor k/C \rfloor$ dan $c = k \bmod C$.

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)
```

Mulai dari pojok kanan atas: elemen yang lebih besar menyingkirkan sisa kolomnya, yang lebih kecil menyingkirkan sisa barisnya. Buku: Subbab 2.9.2.

Metode Tangga, Dijejaki



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.

***Gambar 2.9**: untuk 37, jalur 50, 40, 30, 35, 37 menemukan kunci dalam lima pemeriksaan; untuk 34, jalurnya melangkah ke kiri dan ke bawah sampai keluar kisi di bawah 33, setelah tujuh. Buku: Subbab 2.9.2 dan 2.9.3.*

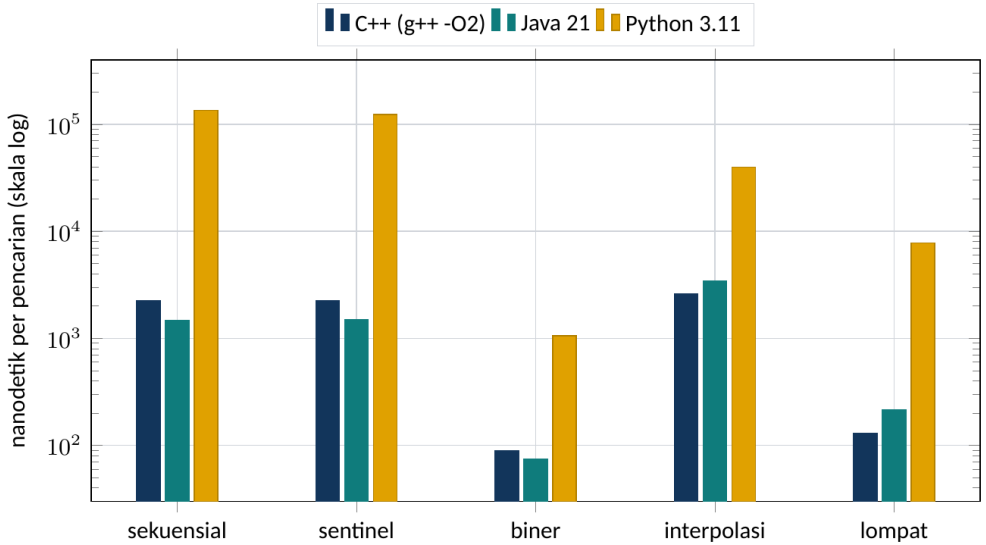
Mengukur pada Katalog Nyata

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

***Tabel 2.3**: katalog goodbooks-10k berisi sepuluh ribu buku nyata; 1.000 identitas yang ada dan 905 yang tidak ada, terbaik dari sembilan kali jalan. Hitungan pemeriksaan cocok dengan teori: 4.996 dan 10.000 untuk pencarian sekuensial, 12,4 dan 13,3 untuk pencarian biner, tidak pernah lebih dari 14. Buku: Subbab 2.10.1 dan 2.10.2.*

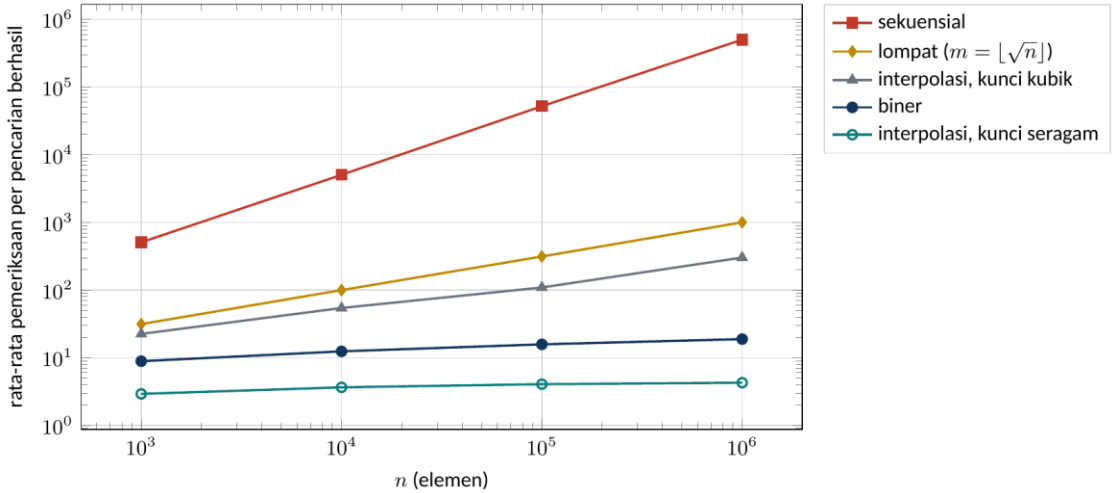
Pemeriksaan Bukanlah Waktu



Gambar 2.10 Nanodetik per pencarian pada katalog 10.000 buku, menurut metode pencarian dan bahasa, pada skala logaritmik, diukur pada September 2026.

***Gambar 2.10**: pencarian biner sekitar 25 kali lebih cepat daripada sekuensial di C++, bukan 600, karena penelusuran membaca memori secara berurutan. Interpolasi gagal pada identitas yang menggerombol (211 pemeriksaan). Pencarian biner di Python mengalahkan pencarian sekuensial di C++. Buku: Subbab 2.10.3.*

Bagaimana Hitungan Pemeriksaan Tumbuh



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.

***Gambar 2.11**: sekuensial naik dengan kemiringan 1, lompat dengan kemiringan $1/2$, biner sekitar 3,3 per langkah sepuluh kali lipat; interpolasi pada kunci seragam hampir tidak naik, tetapi pada kunci kubik ia melonjak dari 22 ke 301. Buku: Subbab 2.10.4.*

Mencari dengan Pustaka Standar

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+)

***Tabel 2.4**: dalam kode produksi, panggillah pustakanya — tetapi ketahui persis apa yang dijanjikan setiap panggilan. C++ bekerja pada rentang setengah terbuka dan menjamin sekitar $\log_2 n$ perbandingan; `list.index` pada Python melempar `ValueError` alih-alih mengembalikan -1. Buku: Subbab 2.11.*

Java: Satu `int`, Dua Jawaban

Kode 2.5 Dari `LibrarySearch.java`: `Arrays.binarySearch` dan hasil negatifnya, serta penyisipan di titik sisip.

```
        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);
```

```
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]
```

Kode 2.5*: hasil negatif r berarti "tidak ada, dan tempatnya di $-r-1$"; tambahan -1 mencegah kunci yang tidak ada tampak seperti "ditemukan di indeks 0". Java tidak punya upper bound: tulis* ***Pseudokode 2.5*** *dengan* `<=`*. Buku: Subbab 2.11.2.*

Python: Modul `bisect`

Kode 2.6 Dari `library_search.py`: *lower bound* dan *upper bound* lewat `bisect_left` dan `bisect_right`, `insort`, dan pencarian rekaman dengan `key=`.

```
    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)
    i = bisect.bisect_left(shelf, 63, key=lambda book: book[0])  # Python 3.10+
    print("record with code 063:", shelf[i][1])
```

```
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
```

***Kode 2.6**: `insort` menemukan tempatnya dalam $O(\lg n)$, tetapi penyisipannya tetap menggeser setiap elemen sesudahnya, $O(n)$. Argumen `key=` (Python 3.10+) diterapkan pada elemen, maka kita memberikan kode 63 apa adanya. Buku: Subbab 2.11.3.*

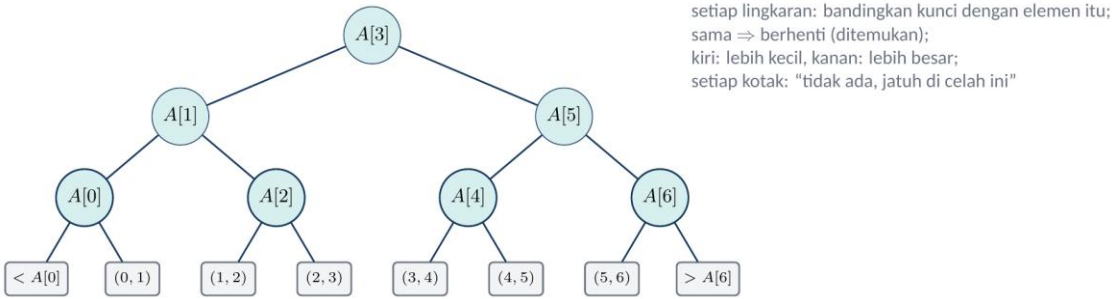
Memilih Metode

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	$O(1)$

***Tabel 2.5**: tidak terurut dan dicari sekali, sekuensial; sering dicari, urutkan sekali atau pakai hash; terurut, pencarian biner, dalam bentuk lower bound untuk duplikat dan rentang; interpolasi hanya pada bilangan yang sudah diukur tersebar merata; lompat hanya bila gerak mundur mahal. Buku: Subbab 2.12.1.*

Tak Ada Perbandingan yang Mengalahkan Pencarian Biner



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.

$$h \geq \lceil \lg(n+1) \rceil. \tag{2.11}$$

Gambar 2.12*: kunci yang tidak ada dapat jatuh ke* $n+1$ *celah, sehingga pohon keputusan memerlukan* $n+1$ *daun dan* $2^h \geq n+1$*, yaitu Persamaan (2.11). Karena* $\lceil lg(n+1) \rceil = \lfloor lgn \rfloor + 1$*, pencarian biner optimal. Untuk mengalahkannya diperlukan aritmetika atas kunci, atau hashing (Bab 15). Buku: Subbab 2.12.2.*

Profitina dalam Praktik

Deret waktu sudah terurut, maka "pemindaian antara 1 Juli dan 30 September" adalah dua pencarian *lower bound*, $O(\lg n)$, sepanjang apa pun riwayatnya.

"N baru sejak terakhir Anda melihat" adalah $n - upper(t)$; **peringkat di papan skor** adalah sebuah titik sisip.

Penyaringan tautan berulang kali bertanya "sudahkah kita melihat ini?" setiap detik: $\lfloor \lg n \rfloor + 1$ pemeriksaan, bukan n.

Simpanlah data dalam urutan yang cocok dengan pertanyaan yang akan diajukan, dan setiap pertanyaan menjadi pencarian batas.

Rincian implementasi Profitina adalah urusannya sendiri; bentuk masalah-masalah ini bersifat umum. Buku: Subbab 2.13.1.

Studi Kasus: Profitina Laundry

Kode 2.7 Dari `laundry_orders.cpp`: pesanan antara 10:00 dan 12:00, ditemukan dengan dua *lower bound*.

```
    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);
```

```
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
```

***Kode 2.7**: pesanan datang menurut urutan waktu dengan nomor berurutan, sehingga daftarnya terurut dengan sendirinya. Waktu diubah menjadi menit sejak tengah malam sebelum dicari, karena sebagai teks `"9:30" > "10:05"`. Pencarian menurut nama pelanggan adalah penelusuran, karena nama tidak terurut. Buku: Subbab 2.13.2.*

Kesalahan yang Sering Terjadi

Pencarian biner pada data tidak terurut, **mencampur konvensi interval**, atau `low = mid` pada interval tertutup.

`(low + high) / 2`; mengharapkan salinan *pertama* dari pencarian biner biasa; salah membaca hasil negatif Java.

Mengembalikan penanda alih-alih posisi; berhenti lebih awal saat menghitung; sentinel pada data bersama atau hanya-baca.

Interpolasi tanpa penjaga, atau dipercaya pada data yang belum diukur (17 kali lebih buruk pada katalog nyata).

Menambahkan banyaknya elemen pada alamat; menelusuri kisi kolom demi kolom; membandingkan waktu sebagai teks; mengira pemeriksaan sama dengan waktu.

Lima belas kesalahan, masing-masing dengan perbaikannya, tercantum di buku. Bab 4 mengerjakan soal-soal persis semacam ini. Buku: Subbab 2.14.

Ringkasan: Lima Hal yang Perlu Diingat

1. Kembalikan posisi atau -1; hitung pemeriksaan. Pencarian sekuensial rata-rata $(n+1)/2$ dan n bila tidak ada: $O(n)$.

2. Pencarian biner menjaga invarian **Gambar 2.4** dan memerlukan paling banyak $\lfloor \lg n \rfloor + 1$ pemeriksaan (Persamaan (2.5)).

3. Satu konvensi interval, pembaruan yang menyusutkan, dan titik tengah aman Persamaan (2.6); raihlah *lower bound* dan *upper bound* lebih dahulu.

4. Interpolasi bernilai $\lg\lg n$ pada kunci merata dan $O(n)$ pada kunci menggerombol; pencarian lompat memerlukan sekitar $2\sqrt{n}$.

5. Tidak ada pencarian berbasis perbandingan yang mengalahkan $\lceil \lg(n+1) \rceil$ (Persamaan (2.11)); ukurlah, karena pemeriksaan meramalkan pertumbuhan, bukan kecepatan.

Pencarian biner, interpolasi, dan lompat semuanya mengandaikan data sudah diurutkan. Buku: Subbab 2.15.

Pertemuan Berikutnya

Bab 3 — Pengurutan. Algoritma pengurutan klasik, dari *bubble sort* sampai *quicksort*, *merge sort*, dan *heapsort*, diukur pada katalog yang sama.

Mengurutkan sekali agar dapat mencari biner berkali-kali adalah salah satu tawar-menawar terbaik dalam komputasi.

Sebelum itu: kerjakan Soal Latihan 2.4, 2.6, 2.7, dan 2.11, dalam C++ dan satu bahasa lain.

Soal Latihan 2.6 menyembunyikan beberapa bug dalam satu pencarian biner pendek: temukan masing-masing dengan empat jebakan hari ini. Buku: Subbab 2.16 dan 2.17.

Kuliah 3 — Pengurutan

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 3

Dosen: Irfan Subakti — yifana@gmail.com

Hari ini kita melunasi utang yang ditinggalkan pencarian biner: sebelas cara menata data, dari pengurutan yang ditemukan siapa saja sampai yang tertanam di dalam pustaka yang kita pakai setiap hari. Buku: Bab 3, Subbab 3.1–3.17.

Cakupan Pertemuan Ini

1. **Istilah yang kita perlukan** — urutan terurut, pertukaran, perbandingan dan pertukaran, inversi; di tempat, stabil, adaptif.
2. **Pengurutan sederhana** — bubble sort (dengan keluar dini), exchange sort, selection sort, dan insertion sort; mengapa keempatnya kuadratik.
3. **Lebih cepat dengan lompatan jauh** — Shell sort dan barisan celahnya.
4. **Bagi dan kuasai** — merge sort dan quicksort dengan partisi Lomuto dan Hoare; heap sort, pandangan pertama.
5. **Melewati batas** $n \lg n$ — counting sort dan radix sort.
6. **Tabel lengkap, hasil pengukuran, pengurutan pustaka tahun 2026, Profitina Laundry.**

Setiap algoritma mengurutkan enam bilangan yang sama, sehingga jejaknya dapat dijajarkan. Buku: Subbab 3.1.

Titik Berangkat

Pencarian biner menemukan satu buku di antara sejuta dalam sekitar dua puluh kali lihat — tetapi hanya pada data yang **sudah terurut**. Pada rak yang tidak terurut ia diam-diam mengembalikan jawaban yang salah.

Setiap pengurutan hari ini mengajarkan gagasan yang muncul kembali kelak: pertukaran bertetangga, keadaptifan, bagi dan kuasai (Bab 10), heap (Bab 14).

Larik berjalan kita, dengan posisi bernomor mulai dari 0:

$$A = [22, 10, 15, 3, 8, 2]$$

Bab 2 mengurutkan lariknya dengan pustaka dan berjanji menjelaskan bagaimana pengurutan dilakukan. Buku: Subbab 3.1.

Apa Itu Pengurutan

DEFINISI. Diberikan n butir $a_0, \ldots, a_{n-1}$ dan sebuah urutan pada kuncinya, **pengurutan** menghasilkan susunan ulang $a'_0, \ldots, a'_{n-1}$ dari butir-butir yang sama sehingga Persamaan (3.1) berlaku. Urutan menurun memakai $\geq$.

$$a'_0 \leq a'_1 \leq \cdots \leq a'_{n-1} \tag{3.1}$$

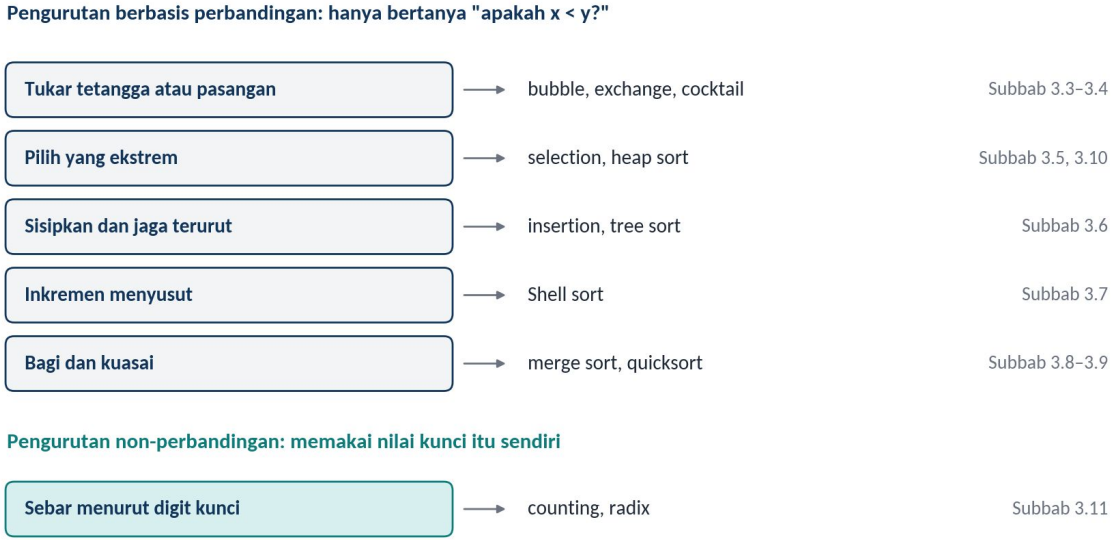
Dua syarat: *terurut*, dan *susunan ulang* — enam salinan bilangan terkecil memang terurut, tetapi bukan hasil pengurutan.

Kunci adalah bagian butir yang dilihat oleh urutan: judul hari ini, kategori besok.

Data yang terurut membuat pencarian, penemuan duplikat, pengelompokan, penggabungan, dan pemeringkatan menjadi murah.

Knuth menghabiskan sebagian besar satu jilid setebal 780 halaman untuk pengurutan; gagasan-gagasannya bersih, dipahami dengan baik, dan terdokumentasi dengan baik. Buku: Subbab 3.2.1 dan 3.2.2.

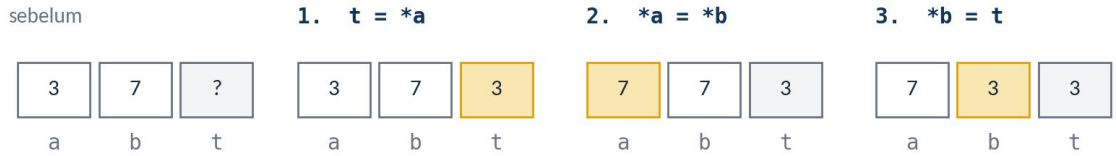
Enam Keluarga Metode Pengurutan



Gambar 3.1 Keluarga metode pengurutan dalam bab ini. Lima yang pertama membandingkan kunci; yang terakhir langsung memakai digit-digit kunci.

Gambar 3.1: penukaran, seleksi, penyisipan, penambahan menyusut, bagi dan kuasai, serta distribusi. Lima yang pertama membandingkan kunci; yang terakhir langsung memakai digit-digit kunci. Buku: Subbab 3.2.3.

Satu Gerakan yang Diperlukan Setiap Pengurutan



Tiga penugasan dan satu variabel cadangan t: nilai yang pertama ditimpa harus disimpan lebih dulu.

Gambar 3.2 Menukar dua nilai dengan variabel cadangan. Setiap kotak yang disorot adalah kotak yang berubah pada langkah itu.

Gambar 3.2: `a = b; b = a;` *menghancurkan satu-satunya salinan nilai pertama. Variabel sementara* `t` *menyelamatkannya: tiga penugasan. Buku: Subbab 3.2.4.*

Menukar Melalui Pointer

Kode 3.1: fungsi yang menerima dua `int` menerima salinan; untuk menukar variabel pemanggil, ia menerima **alamat**-nya.

Kode 3.1 Dari `swap_demo.cpp`: menukar dua `int` melalui pointer.

```
void swapInts(int* a, int* b) {        // menerima dua alamat, bukan dua nilai
    int t = *a;                         // amankan nilai pertama
    *a = *b;
    *b = t;
}
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);
```

Java tidak dapat menukar dua variabel lokal di dalam metode, tetapi dapat menukar dua slot larik yang diterimanya. **Python** menulis `a, b = b, a`.

Dalam C++ sungguhan kita memanggil `std::swap(x, y)`, yang menerima referensi dan melakukan tiga penugasan yang sama; sejak bubble sort, program C++ kita memakainya. Buku: Subbab 3.2.4.

Menghitung Kerja

Perbandingan bertanya “apakah x sebelum y?”; **pertukaran** menukar dua butir; **pergeseran** atau **pemindahan** menyalin satu butir. Satu pertukaran kira-kira tiga pemindahan.

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

Tabel 3.1: pada semiliar butir dan 10^9 langkah per detik, $n \lg n$ memerlukan 30 detik dan $n^2/2$ memerlukan 16 tahun. Melipatduakan n membuat pengurutan kuadratik empat kali lebih lama dan pengurutan $n \lg n$ sedikit lebih dari dua kali. Buku: Subbab 3.2.5.

Inversi: Seberapa Tidak Terurut?

DEFINISI. Inversi adalah pasangan (i, j) dengan $i < j$ dan $a_i > a_j$; Persamaan (3.2) menghitung banyaknya.

$$inv(a) = \left| \{(i, j) : 0 \le i < j \le n-1, a_i > a_j\} \right|. \tag{3.2}$$

Terurut: 0 inversi. Terbalik: semua $n(n-1)/2$ pasangan.

Hitungan terkerjakan pada A**:** 22 mengalahkan lima bilangan sesudahnya, 10 mengalahkan tiga (3, 8, 2), 15 mengalahkan tiga, 3 mengalahkan satu, 8 mengalahkan satu:

$$5+3+3+1+1+0=13$$

Ingat angka 13: ia muncul lagi sebagai banyaknya pertukaran bubble sort dan pergeseran insertion sort. Buku: Subbab 3.2.5.

Tiga Sifat

> **DEFINISI.** Sebuah pengurutan **di tempat** bila ia hanya memakai memori tambahan berukuran konstan di samping lariknya. Ia **stabil** bila butir-butir berkunci sama keluar dalam urutan masukannya. Ia **adaptif** bila ia bekerja lebih sedikit pada masukan yang hampir terurut.

Di tempat penting bila memori terbatas atau datanya sangat besar.

Stabil penting setiap kali rekaman diurutkan menurut satu kunci setelah ditata menurut kunci lain: katalog menurut id, lalu menurut kategori.

Adaptif penting karena data nyata sering *hampir* terurut: daftar kemarin ditambah beberapa entri baru.

Subbab 3.12 memperlihatkan pengurutan tidak stabil yang mengacak katalog. Buku: Subbab 3.2.6.

Bubble Sort

Telusuri dari kiri ke kanan, tukar tetangga yang urutannya salah. Satu **lintasan** membawa elemen terbesar ke ujung, seperti gelembung yang naik di dalam soda.

Invarian: setelah lintasan ke-p, p posisi terakhir memuat p elemen terbesar, terurut.

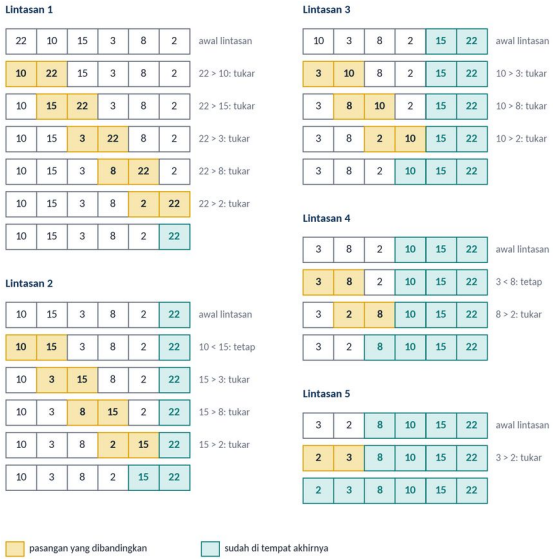
Keluar dini: lintasan tanpa pertukaran berarti larik sudah terurut, menurut Persamaan (3.1).

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
```

Pseudokode 3.1: penanda `swapped` *diatur ulang pada awal setiap lintasan. Buku: Subbab 3.3.1 dan 3.3.2.*

Jejak Bubble Sort



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.

Gambar 3.3: lintasan 1 membawa 22 ke ujung, lintasan 2 membawa 15; lima lintasan, lima belas perbandingan. Bilangan kecil merayap ke kiri satu langkah per lintasan. Buku: Subbab 3.3.3.

Bubble Sort dalam C++

Kode 3.2 Dari `bubble_sort.cpp`: kedua gelung, penanda arah, dan keluar dini.

```
    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;
            }
                 << (swapped ? "" : "  no swap, stop") << "\n";
```

```
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
```

Kode 3.2 mencetak larik setelah setiap lintasan, cocok dengan Gambar 3.3. Pada masukan terurut ia berhenti setelah satu lintasan berisi 5 perbandingan; pada masukan terbalik setiap perbandingan menjadi pertukaran. Java menulis pertukarannya dengan `t`; Python menamai penghitungnya `pass_no`, karena `pass` adalah kata kunci. Buku: Subbab 3.3.4.

Tepatnya Berapa Banyak Kerja

Kasus terburuk, Persamaan (3.3); untuk $n = 6, 15$; untuk $n = 100.000$, sekitar lima miliar:

$$C_{\text{worst}}(n) = \sum_{p=1}^{n-1} (n-p) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}. \tag{3.3}$$

Kasus terbaik dengan keluar dini: $n-1$ perbandingan. **Pertukaran**: pertukaran bertetangga menghapus tepat satu inversi, Persamaan (3.4):

$$\text{pertukaran oleh bubble sort} = inv(A). \tag{3.4}$$

Rata-rata: setiap pasangan terbalik pada separuh dari semua urutan, Persamaan (3.5):

$$\overline{inv}(n) = \frac{1}{2} \cdot \frac{n(n-1)}{2} = \frac{n(n-1)}{4}. \tag{3.5}$$

> **WAWASAN KUNCI.** Algoritma apa pun yang hanya menukar elemen *bertetangga* memerlukan rata-rata $n(n-1)/4$ pertukaran: kuadratik. Untuk mengalahkan n^2, pindahkan elemen sejauh-jauhnya.

Bubble sort bersifat di tempat dan stabil: ia hanya menukar pada `>` yang tegas. Buku: Subbab 3.3.5 dan 3.3.6.

Exchange Sort

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]
```

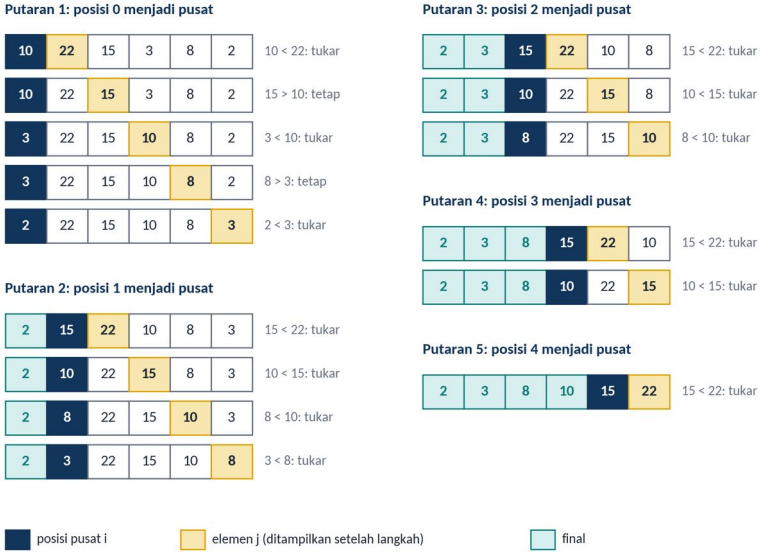
Posisi i dibandingkan dengan **setiap** posisi sesudahnya dan mengambil nilai yang lebih kecil yang dijumpainya.

$n(n-1)/2$ perbandingan pada *setiap* kasus: tanpa keluar dini, tidak adaptif.

13 pertukarannya pada A adalah kebetulan, bukan hukum: pertukaran jarak jauh dapat mengubah banyak inversi sekaligus.

*Pseudokode 3.2 bersifat di tempat tetapi **tidak stabil**: lompatan jauh dapat membawa sebuah elemen melewati kunci yang sama. Buku: Subbab 3.4.*

Jejak Exchange Sort



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.

Gambar 3.4: pada putaran 1 posisi 0 berubah tiga kali, $22 \rightarrow 10 \rightarrow 3 \rightarrow 2$*, sebelum nilai terkecil menetap di sana. Buku: Subbab 3.4.*

Selection Sort: Cari Dulu, Tukar Sekali

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]
```

```
  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
```

Pseudokode 3.3 adalah pengurutan yang dibangun dari pencarian: setiap putaran adalah penelusuran linear Bab 2. Selalu $n(n-1)/2$ *perbandingan, tetapi* ***paling banyak*** $n-1$ ***pertukaran*** *— 3 pada* A *dibandingkan 13. Baik bila memindahkan itu mahal (rekaman besar, memori flash); tidak adaptif, tidak stabil. Buku: Subbab 3.5.1–3.5.3.*

Jejak Selection Sort



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.

Gambar 3.5: 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. Buku: Subbab 3.5.1.

Insertion Sort: Kartu di Tangan

Pseudokode 3.4 *Insertion sort*: tumbuhkan awalan terurut dengan menyisipkan satu kunci setiap kali, menggeser elemen yang lebih besar ke kanan.

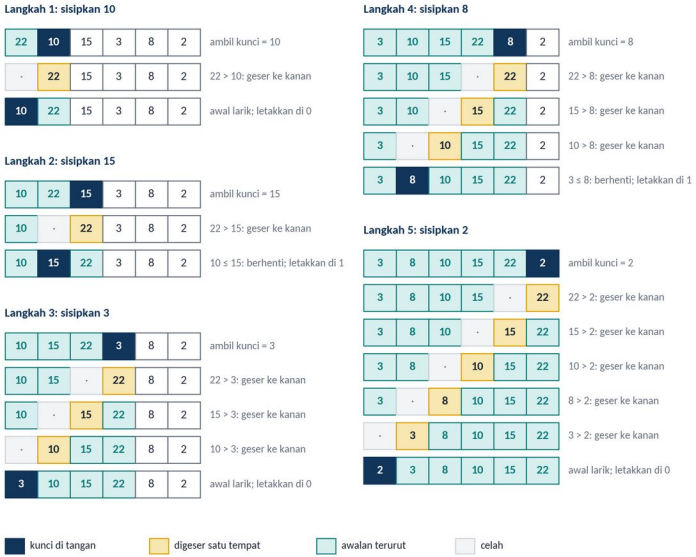
```
INSERTION-SORT(A, n)
 1  for i = 1 to n − 1
 2      key = A[i]                          // kartu di tangan kita
 3      j = i − 1
 4      while j ≥ 0 and A[j] > key          // uji j ≥ 0 lebih dulu
 5          A[j+1] = A[j]                   // geser yang lebih besar ke kanan
 6          j = j − 1
 7      A[j+1] = key                        // jatuhkan kunci ke lubang
```

Invarian: pada awal langkah i, $A[0..i-1]$ memuat elemen-elemen aslinya, terurut.

> **UJI INDEKS LEBIH DAHULU.** `while (a[j] > key && j >= 0)` membaca `a[-1]`: perilaku tak terdefinisi di C++, eksepsi di Java, dan di Python diam-diam elemen *terakhir*. Tulislah `j >= 0 && a[j] > key`.

Pseudokode 3.4 mengandalkan evaluasi hubung-pendek: `&&` dan `and` melewatkan operan kanan bila operan kiri salah. Buku: Subbab 3.6.1 dan 3.6.2.

Jejak Insertion Sort



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.

Gambar 3.6: setiap langkah mengambil sebuah kunci (biru tua), menggeser elemen-elemen yang lebih besar di awalan terurut satu tempat ke kanan, lalu menjatuhkan kunci ke celahnya. Buku: Subbab 3.6.1.

Insertion Sort dalam Python

Kode 3.3 Dari `insertion_sort.py`: gelung menguji `j >= 0` sebelum membaca `a[j]`, dan berhenti pada kunci yang sama.

```python
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
```

```
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]
```

Kode 3.3: 15 perbandingan dan 13 pergeseran pada A — lagi-lagi 13 inversi. Versi C++ dan Java mencetak baris-baris yang sama. Buku: Subbab 3.6.3.

Mengapa Insertion Sort Terbaik di Antara yang Sederhana

Setiap pergeseran membereskan tepat satu inversi: pergeseran $= inv(A)$. Setiap langkah menambah paling banyak satu perbandingan penghenti, Persamaan (3.6):

$$n-1 \le C_{\text{insertion}}(A) \le inv(A) + n - 1 \le \frac{n(n-1)}{2} + n - 1. \tag{3.6}$$

Masukan terurut: $n-1$ perbandingan, tanpa pergeseran. Terbalik: $n(n-1)/2$ untuk keduanya. Acak: sekitar $n^2/4$.

Kerjanya sebanding dengan $n + inv(A)$: itulah arti tepat **adaptif**.

Di tempat, stabil, mungil, dan cepat pada larik kecil atau hampir terurut — itulah sebabnya setiap pengurutan pustaka yang serius menutup kerjanya dengan insertion sort pada potongan-potongan kecil. Buku: Subbab 3.6.4.

Shell Sort: Lompatan Jauh Lebih Dahulu

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
```

Pseudokode 3.5, Donald Shell, 1959: insertion sort pada elemen-elemen yang berjarak h, untuk celah yang makin kecil. Dengan celah 1 ia persis insertion sort, sehingga selalu benar; celah-celah sebelumnya hanya membuat lintasan terakhir murah. Tidak stabil. Buku: Subbab 3.7.1.

Contoh Terkerjakan 1: Celah 3 dan 1



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.

Gambar 3.7: celah 3 mengurutkan pasangan $(22,3)$, $(10,8)$, $(15,2)$*; 22 melompat tiga tempat dalam satu gerakan dan larik menjadi* $[3,8,2,22,10,15]$*, menghapus 9 dari 13 inversi. Celah 1 menghapus 4 sisanya: 11 perbandingan dibandingkan 15. Buku: Subbab 3.7.2.*

Barisan Celah Menentukan Segalanya

Shell: $\lfloor n/2 \rfloor, \lfloor n/4 \rfloor, \ldots, 1$: kasus terburuk tetap $\Theta(n^2)$.

Knuth: $1, 4, 13, 40, 121, \ldots$, Persamaan (3.7), kasus terburuk $O(n^{3/2})$:

$$h_1 = 1, h_{k+1} = 3h_k + 1, \tag{3.7}$$

Ciura: $1, 4, 10, 23, 57, 132, 301, 701$ (2001, ditemukan lewat percobaan): terbaik yang dikenal dalam praktik.

```
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 akan memerlukan sekitar 2,5 miliar perbandingan pada 100.000 bilangan ini; Shell sort memerlukan 2,6 sampai 4,5 juta — seribu kali lebih sedikit kerja dari perubahan delapan baris. Buku: Subbab 3.7.3.

Merge Sort: Bagi dan Kuasai

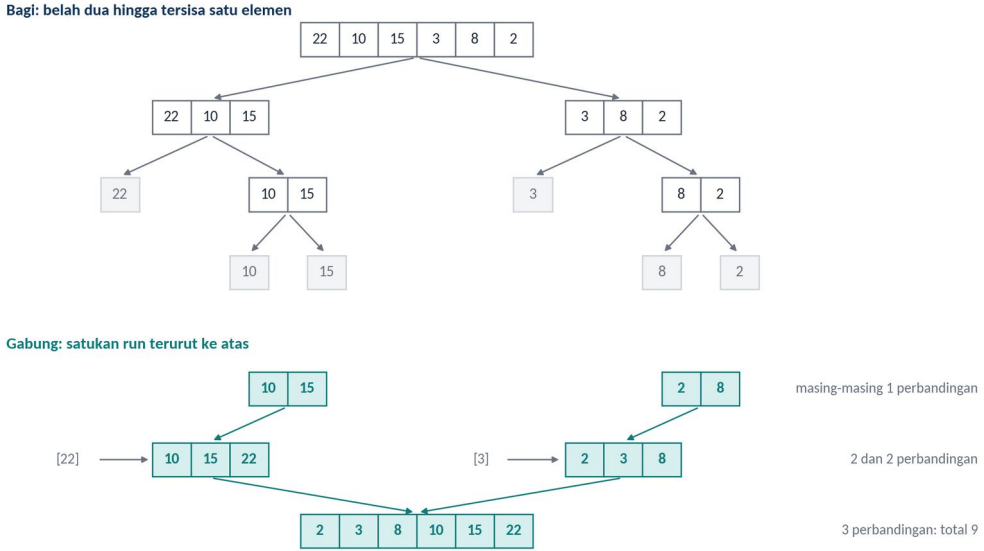
Pseudokode 3.6 `MERGE` dan `MERGE-SORT`. Penyangga T sama panjang dengan A dan dialokasikan sekali saja, sebelum pemanggilan pertama.

```
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)
```

*Pseudokode 3.6: **bagi** dua, **kuasai** setiap separuh secara rekursif, **gabungkan** — ambil kartu teratas yang lebih kecil dari dua tumpukan terurut, ulangi. Satu buffer T, dialokasikan sekali; `lo + (hi − lo)/2` tidak dapat meluap. Buku: Subbab 3.8.1.*

Jejak Merge Sort



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.

Gambar 3.8: separuh atas membagi, separuh bawah menggabungkan; $1+2+1+2+3=9$ *perbandingan. Penggabungan terakhir mengambil 2, 3, dan 8, lalu menyalin 10, 15, 22 tanpa membandingkan. Buku: Subbab 3.8.2.*

`merge` dalam C++

Kode 3.4 Dari `merge_sort.cpp`: menggabungkan melalui satu *buffer* bersama, dan rekursinya.

```cpp
// 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);
```

*Kode 3.4: `<=` mengambil dari deret kiri bila kuncinya sama, sehingga merge sort **stabil**. Program mencetak 9 perbandingan dan 16 pemindahan untuk A. Buku: Subbab 3.8.3.*

Mengapa Merge Sort Selalu $n \lg n$

Dua pemanggilan berukuran separuh dan satu penggabungan linear memberi Persamaan (3.8):

$$T(n) = 2T(n/2) + cn, T(1) = c. \tag{3.8}$$

Setiap aras pohon rekursi berbiaya cn, dan ada $\lg n + 1$ aras, sehingga Persamaan (3.9):

$$T(n) = cn(\lg n + 1) = \Theta(n \lg n). \tag{3.9}$$

Tidak ada yang bergantung pada data: $\Theta(n \lg n)$ pada kasus terbaik, rata-rata, *dan* terburuk.

*Harganya adalah memori: buffer berisi n elemen, sehingga merge sort **tidak di tempat**. Timsort dan Powersort adalah merge sort yang dibuat adaptif dengan menemukan deret-deret yang sudah ada di dalam data. Buku: Subbab 3.8.4.*

Quicksort: Partisi, lalu Rekursi

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)
```

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
```

Pseudokode 3.7, Hoare 1959: kerja nyata untuk membagi, tanpa kerja untuk menggabungkan. Pseudokode 3.8 menjaga $A[lo..i] \le x$ dan $A[i+1..j-1] > x$, lalu menaruh pivot di antara kedua daerah. Buku: Subbab 3.9.1 dan 3.9.2.

Jejak Partisi Lomuto



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.

Gambar 3.9: dengan elemen terakhir sebagai pivot, 2 lalu 22 tidak memisahkan apa pun di satu sisi; baru pivot ketiga, 8, membagi dengan seimbang. Buku: Subbab 3.9.2.

Partisi Hoare

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]
```

i berhenti pada elemen $\geq x$, j pada elemen $\leq x$; keduanya berada di sisi yang salah, maka keduanya ditukar.

Pivot tidak harus berakhir di j: pemanggilan rekursifnya `QUICKSORT(A, lo, j)` dan `QUICKSORT(A, j+1, hi)`.

Pseudokode 3.9 melakukan sekitar sepertiga pertukaran Lomuto dan membagi larik berkunci sama dengan seimbang; quicksort pustaka diturunkan darinya. Buku: Subbab 3.9.3.

Contoh Terkerjakan 2: Hoare pada A



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 $[15,22]$.

Gambar 3.10: pivot 15. Tukar 22 dan 2; tukar 15 dan 8; lalu $i=4\geq j=3$: kedua indeks bersilangan dan partisi mengembalikan 3, membagi A menjadi $[2,10,8,3]$ dan $[15,22]$. Buku: Subbab 3.9.3.

Partisi Lomuto dalam C++

Kode 3.5 Dari `quick_sort.cpp`: partisi Lomuto, dengan menghitung perbandingan dan pertukaran.

```cpp
// 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
}
```

```
  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
```

Kode 3.5 pada 3.000 elemen: pivot terakhir mengubah masukan terurut menjadi kasus terburuk, $3000 \cdot 2999/2$ *perbandingan dan rekursi sedalam 2.998. Java memakai thread ber-stack 64 MB. Buku: Subbab 3.9.4.*

Kasus Terbaik, Terburuk, dan Rata-rata

Terburuk: pivot selalu yang terkecil atau terbesar, Persamaan (3.10):

$$T_{\text{worst}}(n) = T_{\text{worst}}(n-1) + (n-1) = \frac{n(n-1)}{2} = \Theta(n^2). \tag{3.10}$$

Rata-rata, masukan acak atau pivot acak, Persamaan (3.11) — hanya 39% di atas kasus terbaik:

$$\overline{C}_{\text{quick}}(n) = 2(n+1)H_n - 4n \approx 2n \ln n \approx 1.39\, n \lg n, \tag{3.11}$$

Median dari tiga mengubah masukan terurut menjadi kasus terbaik; **pivot acak** membuat setiap masukan sama baiknya.

Rekursilah ke sisi yang lebih kecil agar stack tetap di bawah $\lg n$*; beralihlah ke heap sort bila rekursinya terlalu dalam, dan jadilah* ***introsort****, isi* `std::sort`*. Quicksort tidak stabil. Buku: Subbab 3.9.5.*

Heap Sort: Pohon di Dalam Larik

Bacalah $A[0..n-1]$ sebagai pohon biner dengan aritmetika indeks Persamaan (3.12); tidak ada pointer yang disimpan.

$$left(i)=2i+1,\ right(i)=2i+2,\ parent(i)=\lfloor (i-1)/2 \rfloor. \tag{3.12}$$

DEFINISI. Sebuah larik adalah **max-heap** bila $A[parent(i)] \geq A[i]$ untuk setiap $i \geq 1$. Elemen terbesar lalu berada di akar, $A[0]$.

Sift down: tukar sebuah simpul dengan anaknya yang lebih besar sampai ia paling sedikit sebesar keduanya, paling banyak $2\lfloor \lg n \rfloor$ perbandingan. **Membangun heap**: *sift down* dari indeks $\lfloor n/2 \rfloor - 1$ mundur sampai 0, dalam $O(n)$.

Heap sort mempertahankan rencana selection sort — ambil yang terbesar, taruh di ujung — tetapi membuat yang terbesar gratis untuk ditemukan. Bab 14 membangun heap secara lengkap. Buku: Subbab 3.10.1.

SIFT-DOWN dan HEAP-SORT

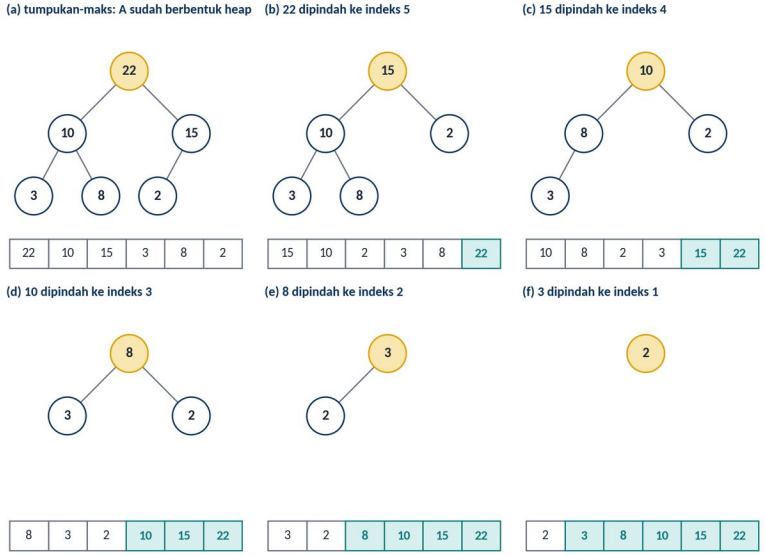
Pseudokode 3.10 SIFT-DOWN dan HEAP-SORT: bangun *max-heap*, lalu berulang kali pindahkan akar ke ujung *heap* yang menyusut dan perbaiki akarnya.

```
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
```

Pseudokode 3.10: $O(n \lg n)$ *pada kasus terburuk dan di tempat — perpaduan yang tidak dimiliki pengurutan lain di bab ini. Lompatannya dari* i *ke* $2i+1$ *tidak ramah terhadap cache, dan ia tidak stabil. Buku: Subbab 3.10.1.*

Jejak Heap Sort



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.

Gambar 3.11: A sudah merupakan max-heap. Langkah (b) menukar 22 dengan 2 lalu menenggelamkan 2 ke bawah 15, menyisakan heap $[15, 10, 2, 3, 8]$; ekor hijau-kebiruan bertambah satu elemen per langkah. Buku: Subbab 3.10.2.

Sift Down dalam C++

Kode 3.6 Dari `heap_sort.cpp`: menenggelamkan sebuah simpul di bawah anaknya yang lebih besar sampai heap pulih.

```cpp
// 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;
    }
```

Kode 3.6: heap-nya adalah `a[0..size-1]`, dan ekor terurut tinggal di larik yang sama di belakangnya. Pada $[3,8,2,22,10,15]$ *fase pembangunan memindahkan 22 ke akar lebih dahulu. Buku: Subbab 3.10.2.*

Batas yang Tidak Dapat Dilintasi Pengurutan Perbandingan

$n!$ urutan yang mungkin harus dibedakan; k perbandingan ya-atau-tidak membedakan paling banyak 2^k di antaranya, sehingga Persamaan (3.13):

$$k \geq \lg(n!) \geq \lg\left(\left(\frac{n}{2}\right)^{n/2}\right) = \frac{n}{2}\lg\frac{n}{2} = \Omega(n \lg n). \tag{3.13}$$

Untuk enam elemen, $\lg 720 \approx 9,49$: suatu masukan memerlukan paling sedikit 10 perbandingan, algoritma apa pun.

Merge sort dan heap sort optimal sampai faktor konstan.

Jalan melewati batas ini adalah berhenti membandingkan, dan memakai nilai kunci sebagai indeks larik. Buku: Subbab 3.11.1.

Counting Sort

Pseudokode 3.11 *Counting sort* untuk kunci bilangan bulat di $0..k-1$. Tidak ada dua kunci yang pernah dibandingkan.

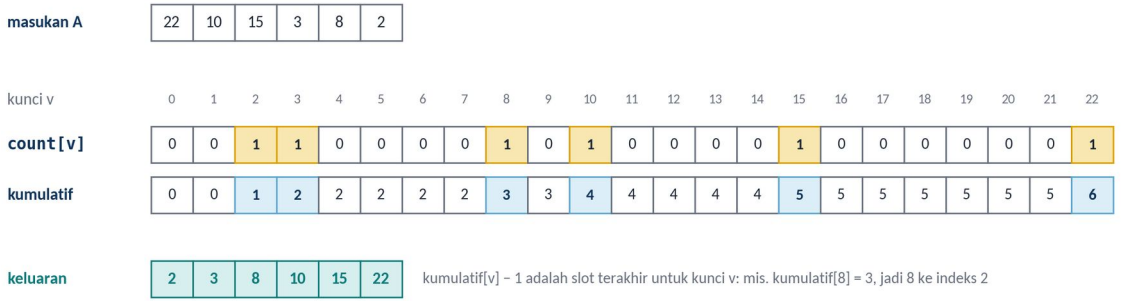
```
COUNTING-SORT(A, n, k)                          // kunci 0..k-1; keluaran B
 1  count[0..k-1] = 0
 2  for i = 0 to n - 1
 3      count[A[i]] = count[A[i]] + 1
 4  for v = 1 to k - 1
 5      count[v] = count[v] + count[v-1]          // kini: banyak kunci ≤ v
 6  for i = n - 1 down to 0                      // kanan ke kiri: stabil
 7      count[A[i]] = count[A[i]] - 1
 8      B[count[A[i]]] = A[i]
 9  return B
```

$$T_{\text{counting}}(n,k)=\Theta(n+k). \tag{3.14}$$

$$T_{\text{radix}}(n)=\Theta(d(n+b)). \tag{3.15}$$

Pseudokode 3.11: hitung, jadikan total berjalan, tempatkan dari ***kanan ke kiri*** *— stabil. Persamaan (3.14) linear bila k paling banyak konstanta kali n; radix sort, Persamaan (3.15), adalah d lintasan counting sort dalam basis b. Buku: Subbab 3.11.2 dan 3.11.3.*

Jejak Counting Sort



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.

Gambar 3.12: total berjalan pada 8 adalah 3 — tiga kunci (2, 3, 8) paling besar 8 — sehingga 8 menempati indeks $3-1=2$. *Buku: Subbab 3.11.2.*

Radix Sort: Digit Paling Tidak Signifikan Dahulu



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.

Gambar 3.13: urutkan menurut digit satuan, lalu secara stabil menurut digit puluhan; 22 tetap di depan 2 setelah lintasan pertama, dan 2, 3, 8 mempertahankan urutan itu pada lintasan kedua. Buku: Subbab 3.11.3.

Counting Sort dalam C++

Kode 3.7 Dari `counting_radix.cpp`: counting sort yang stabil dengan parameter fungsi kunci, sehingga satu fungsi melayani radix sort juga.

```cpp
// 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)
```

```
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]
```

Kode 3.7: radix sort memberikan `x / place % 10` sebagai kuncinya. Keduanya memerlukan larik keluaran (tidak di tempat), keduanya stabil, dan keduanya hanya untuk kunci yang dapat dipotong menjadi digit. Buku: Subbab 3.11.3.

Kestabilan yang Dapat Dilihat

Masukan (urut id)		Stabil (insertion sort)		Tidak stabil (selection sort)	
Fiksi	**B1001**	Anak-anak	B1006	Anak-anak	B1006
Fiksi	**B1002**	Fiksi	**B1001**	Fiksi	**B1002**
Fiksi	**B1003**	Fiksi	**B1002**	Fiksi	**B1003**
Sejarah	B1004	Fiksi	**B1003**	Fiksi	**B1001**
Teknologi	B1005	Fiksi	**B1008**	Fiksi	**B1008**
Anak-anak	B1006	Sains	B1007	Sains	B1007
Sains	B1007	Sejarah	B1004	Sejarah	B1004
Fiksi	**B1008**	Teknologi	B1005	Teknologi	B1005

Keempat buku Fiksi tetap berurutan B1001, B1002, B1003, B1008 hanya pada pengurutan stabil.

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.

Gambar 3.14: diurutkan menurut kategori, insertion sort mempertahankan keempat buku Fiksi dalam urutan id; pertukaran pertama selection sort melempar B1001 ke belakang B1002 dan B1003. Untuk mengurutkan dengan beberapa kunci, urutkan menurut kunci yang paling tidak penting dahulu, lalu secara stabil menurut yang lebih penting. Buku: Subbab 3.12.1.

Tabel Lengkapnya

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

Tabel 3.2: ruang tambahan quicksort adalah stack rekursinya; rata-rata Shell sort adalah taksiran empiris. Buku: Subbab 3.12.2.

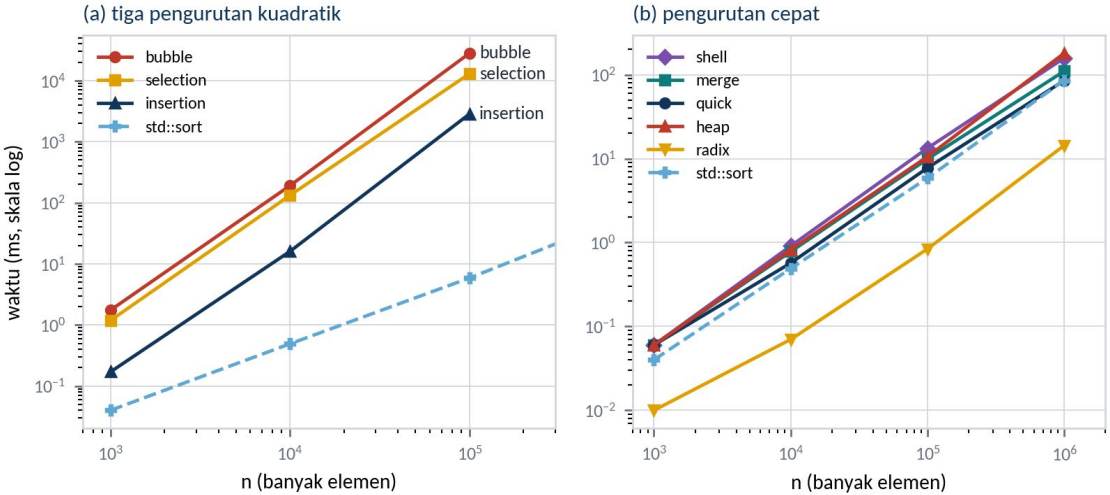
Enam Bilangan yang Sama, Kerja yang Sangat Berbeda

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

Tabel 3.3: angka 24 milik Hoare menghitung setiap uji gelung pemindainya, termasuk yang menghentikannya; pada enam elemen perbedaan seperti ini hanyalah derau. Untuk melihat pengurutan mana yang cepat, kita harus memakai data besar. Buku: Subbab 3.12.2.

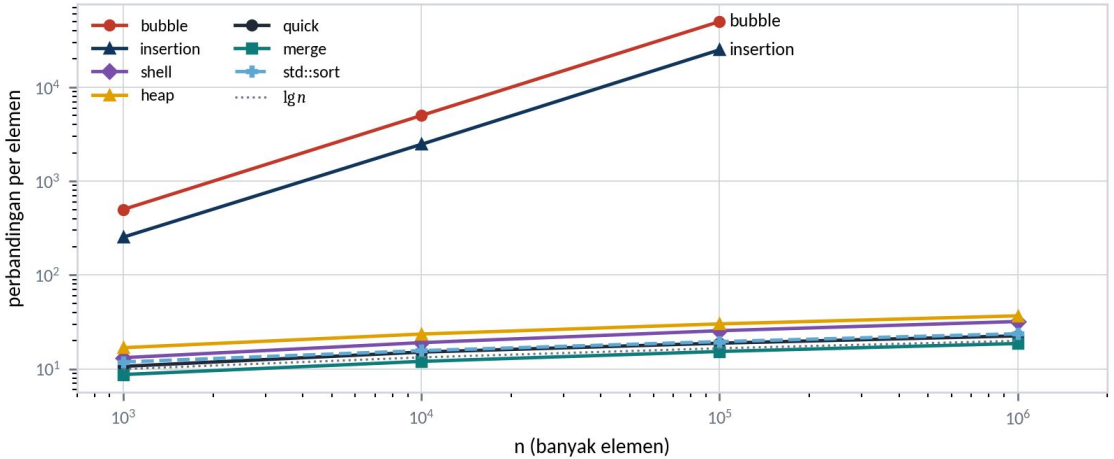
Waktu yang Terukur



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.

Gambar 3.15: pada sumbu log–log, pengurutan kuadratik naik dua dekade waktu untuk setiap dekade n; pada 10^5 elemen bubble sort memerlukan 28 detik dan `std::sort` 6 milidetik. Pada masukan terurut bubble sort dan insertion sort selesai dalam mikrodetik; selection sort memerlukan waktu yang sama apa pun urutannya. Buku: Subbab 3.13.1–3.13.3.

Perbandingan Bukan Seluruh Cerita



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.

Gambar 3.16: merge sort paling sedikit membandingkan, heap sort paling banyak; namun quicksort dan `std::sort` *tercepat pada* 10^6 *(sekitar 85 ms), karena keduanya memindahkan lebih sedikit data dan memindai memori secara berurutan. Radix sort, tanpa perbandingan, memerlukan 14 ms. Buku: Subbab 3.13.2.*

Pengurutan Pustaka Masa Kini

Tabel 3.4 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	$O(n \lg n)$

Tabel 3.4: `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; Python memakai Timsort dengan aturan penggabungan Powersort sejak 3.11. Pakailah, dan curahkan usaha Anda pada pembanding atau kuncinya. Buku: Subbab 3.14.

Profitina dalam Praktik

Daftar sinyal diurutkan menurut dampak, tertinggi dahulu; pengurutan *stabil* menjaga dampak yang sama dalam urutan tetap, sehingga halaman tidak teracak setiap kali dikunjungi.

Tren mingguan datang hampir terurut menurut waktu: Timsort menemukan deret panjangnya dan menggabungkan yang tercecer dalam hampir n perbandingan.

Papan skor hanya memerlukan beberapa teratas: `heapq.nlargest` atau `std::partial_sort`, $O(n \lg k)$.

PostgreSQL memilih quicksort, *top-N heapsort*, atau *external merge* untuk `ORDER BY` — ketiga pengurutan $n \lg n$ hari ini.

Membaca `EXPLAIN ANALYZE` dengan Tabel 3.2 di kepala memperlihatkan kapan sebuah indeks membuat basis data dapat melewatkan pengurutan. Buku: Subbab 3.15.

Profitina Laundry: Terberat Dahulu

Kode 3.8 Dari `laundry_orders.py`: antrean cuci, terberat dahulu, dengan pengurutan stabil.

```
# 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)
```

```
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
```

Kode 3.8: pesanan Maya dan Hadi sama-sama 3,0 kg; pesanan Maya, yang datang lebih dahulu, tetap di depan, karena `sorted` stabil bahkan dengan `reverse=True`. C++ memerlukan `std::stable_sort` untuk jaminan yang sama; `List.sort` pada Java stabil. Buku: Subbab 3.15.

Kesalahan yang Sering Terjadi

Menukar tanpa variabel sementara, atau fungsi tukar yang menerima nilai; gelung dalam yang membaca `a[n]`.

Menguji elemen sebelum indeks; penanda `swapped` yang hanya diatur ulang sekali.

Kehilangan kestabilan karena satu karakter (`>=` alih-alih `>`); *buffer* penggabungan dialokasikan di setiap penggabungan; `(lo + hi) / 2`.

Mencampur pemanggilan rekursif Lomuto dan Hoare; pivot pertama atau terakhir pada data yang tidak Anda kendalikan.

Pembanding dengan `<=`; menghitung dengan `int`; mengira kuadratik itu cukup; menulis pengurutan sendiri di kode produksi.

Lima belas kesalahan, masing-masing dengan perbaikannya, tercantum di buku. Buku: Subbab 3.16.

Ringkasan: Lima Hal yang Perlu Diingat

1. Pengurutan adalah susunan ulang menjadi terurut (Persamaan (3.1)); inversi (Persamaan (3.2)) mengukur seberapa tidak terurut sebuah larik.

2. Pengurutan dengan pertukaran bertetangga memerlukan rata-rata $n(n-1)/4$ pertukaran (Persamaan (3.5)); insertion sort yang terbaik di antaranya: stabil, adaptif, $n+inv(A)$.

3. Merge sort selalu $\Theta(n \lg n)$ (Persamaan (3.9)) tetapi memerlukan *buffer*; quicksort tercepat dalam praktik tetapi $\Theta(n^2)$ dengan pivot yang buruk (Persamaan (3.10)).

4. Heap sort $O(n \lg n)$ pada kasus terburuk dan di tempat; tidak ada pengurutan perbandingan yang mengalahkan $\Omega(n \lg n)$ (Persamaan (3.13)); counting dan radix sort melampauinya pada kunci bilangan bulat.

5. Kestabilan memungkinkan pengurutan multikunci; pengurutan pustaka adalah hibrida — pakailah.

Algoritma yang lebih baik mengalahkan perangkat keras yang lebih baik: seperempat abad perangkat keras memberi faktor 10 sampai 60, sedangkan $n \lg n$ lawan n^2 pada sejuta elemen adalah faktor 25.000. Buku: Subbab 3.17.

Pertemuan Berikutnya

Bab 4 — Evaluasi dan Pembahasan I. Soal-soal terkerjakan tentang Bab 1 sampai 3, beberapa di antaranya meminta kita mengurutkan *string* dan rekaman dengan pembanding sendiri.

Setiap soal berjalan dari pernyataan ke rancangan, ke kode dalam tiga bahasa, ke pemeriksaan.

Sebelum itu: kerjakan Soal Latihan 3.1, 3.3, 3.9, 3.12, dan 3.17 dengan tangan, dan 3.4 dengan kode.

Soal Latihan 3.17 adalah aturan multikunci dari kestabilan yang diterapkan pada Profitina Laundry. Buku: Subbab 3.18 dan 3.19.

Kuliah 4 — Evaluasi dan Pembahasan I: Array, Pencarian, dan Pengurutan, Diselesaikan Langkah demi Langkah

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 4

Dosen: Irfan Subakti — yifana@gmail.com

Tidak ada struktur data baru hari ini: kita mengerjakan sebelas soal tentang Bab 1 sampai 3, masing-masing dari halaman kosong, dalam C++, Java, dan Python, dan masing-masing diperiksa terhadap jejak tangan. Buku: Bab 4, Subbab 4.1–4.17.

Cakupan Pertemuan Ini

1. **Sebuah metode** — delapan langkah dari pernyataan soal menuju program yang teruji, dan anggaran untuk memperkirakan biaya sebelum menulis kode.
2. **Empat soal klasik** — mengurutkan kata, karakter pertama yang berulang, uang menurut nama, mengurutkan menurut kepentingan.
3. **Tujuh soal baru** — alamat larik, menghitung kunci, pencarian interpolasi, menelusuri pengurutan dasar, penggabungan, penghapusan di tempat, dua harga yang cocok dengan voucher.
4. **Menoleh ke belakang** — soal-soal berdampingan, Profitina, kesalahan yang sering terjadi.

Setiap soal menempuh jalur yang sama: pernyataan, dipahami dengan tangan, dirancang di atas kertas, diselesaikan dalam C++ (serta Java dan Python), sebuah penyelesaian keliru yang menggoda, dan varian uji diri. Buku: Subbab 4.2.3.

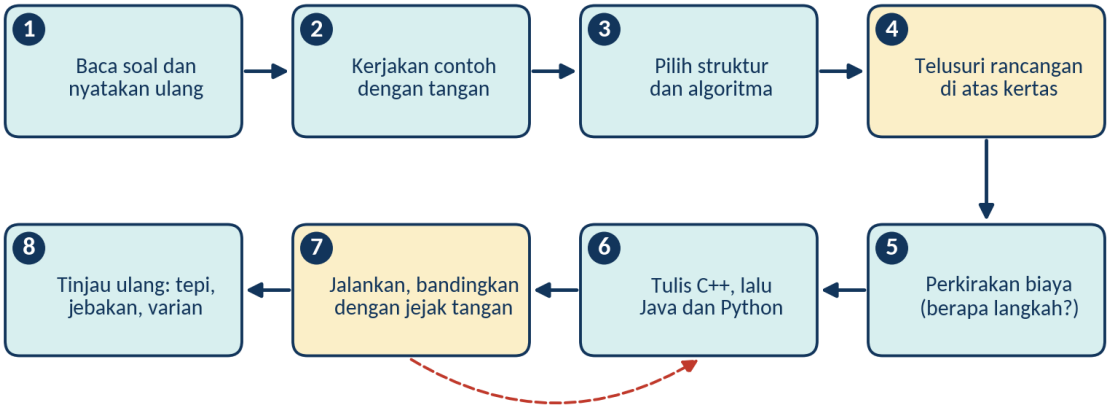
Posisi Kita: Kotak Perkakas

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

***Tabel 4.1**: Bab 1 memberi larik di memori, Bab 2 pencarian sekuensial, sentinel, biner, dan interpolasi, Bab 3 pengurutan dan kata stabil. Belum yakin dengan pencarian biner? Mulailah dari Soal 4.3 dan 4.6. Buku: Subbab 4.1.*

Cara Menggarap Soal



Gambar 4.1 Delapan langkah yang dipakai untuk setiap soal dalam bab ini; panah putus-putus adalah putaran antara menjalankan program dan memperbaikinya.

***Gambar 4.1**: empat gerakan Pólya — pahami, rencanakan, laksanakan, tinjau kembali — direntangkan menjadi delapan langkah kecil; panah putus-putus adalah putaran antara menjalankan program dan memperbaikinya. Buku: Subbab 4.2.*

Delapan Langkah

1. **Nyatakan ulang** apa yang masuk dan apa yang keluar. **2. Kerjakan contoh dengan tangan**, termasuk tepi buatan sendiri: kosong, satu butir, kunci yang hilang, posisi pertama atau terakhir, nilai kembar.
3. **Pilih struktur dan algoritmanya** — kebanyakan soal adalah algoritma yang sudah dikenal dalam penyamaran. **4. Jejaki rancangan di atas kertas**, satu baris per langkah.
5. **Perkirakan biayanya. 6. Tulis kodenya**, C++ lebih dahulu, dalam potongan kecil yang teruji, lalu Java dan Python.
7. **Jalankan dan bandingkan dengan jejak tangan**: temukan nilai *pertama* yang berbeda. **8. Tinjau kembali**: masukan yang belum dicoba, penyelesaian keliru yang menggoda, sebuah varian.

> **MENGAPA JEJAK DIDAHULUKAN DARIPADA KODE.** Program hanya dapat sebenar pemahaman Anda tentang soalnya. Bila Anda tidak dapat menggerakkan indeks melalui sembilan kotak di atas kertas, Anda tidak dapat memeriksa program yang mengaku melakukannya.

Setiap keluaran yang diharapkan hari ini lebih dahulu dihitung dengan tangan. Buku: Subbab 4.2.1.

Anggaran Sebelum Kode

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}

$$(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} \tag{4.1}$$

$$\lfloor \log_2 n \rfloor + 1 \tag{4.2}$$

***Tabel 4.2**: sekitar 10^8 sampai 10^9 langkah sederhana per detik. Dua gelung bersarang atas pasangan memerlukan Persamaan (4.1) — 21 untuk $n = 7$, 28 untuk $n = 8$; pencarian biner memerlukan paling banyak Persamaan (4.2) pemeriksaan — 4 untuk delapan kode, 20 untuk sejuta. Buku: Subbab 4.2.2.*

Soal 4.1: Mengurutkan Kata Menurut Abjad

SOAL 4.1. Create a program that sorts the words in alphabetical order.

Pernyataan satu kalimat: *kitalah* yang memutuskan masukannya (n, lalu n kata), keluarannya, dan urutannya — urutan kamus, dengan kata berhuruf kapital lebih dahulu bila kembar.

Masukan `mango Banana apple Cherry durian banana Apple grape`; dengan tangan: `Apple apple Banana banana Cherry durian grape mango`.

Karakter dibandingkan menurut **kodenya**: `A`–`Z` bernilai 65–90 dan `a`–`z` 97–122, sehingga `<` biasa menyatakan `"Cherry" < "apple"`.

Pernyataan pendek adalah yang paling berbahaya: setiap keputusan yang tidak diambil pernyataan harus Anda ambil sendiri, dan Anda tuliskan. Buku: Subbab 4.3.1 dan 4.3.2.

Soal 4.1: Pengurutannya Tetap, Pembandingnya Berganti

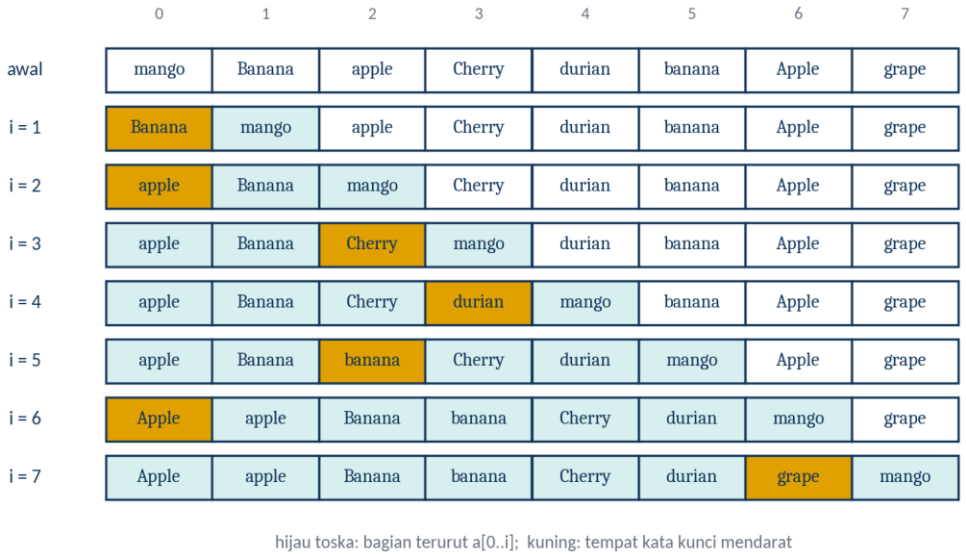
Pseudokode 4.1 Insertion sort untuk kata; perbandingan COMES-BEFORE mewujudkan urutan kamus.

```
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
```

`COMES-BEFORE(a, b)`: bandingkan salinan berhuruf kecil; bila berbeda, itulah penentunya; bila kembar, `a < b` menurut kode menaruh `Apple` sebelum `apple`.

Pseudokode 4.1 *adalah insertion sort Bab 3 dengan satu nama baru di baris 4. Insertion sort pendek, stabil (Soal 4.4 memerlukannya), dan cepat pada daftar yang hampir terurut; biayanya $O(n^2 L)$ untuk kata sepanjang L huruf. Buku: Subbab 4.3.3.*

Soal 4.1: Jejaknya



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.

***Gambar 4.2**: pada baris $i = 3$ kunci `Cherry` melewati `mango` dan berhenti di `Banana`; pada baris 6 kunci `Apple` bergerak sampai indeks 0, melewati `apple`. Buku: Subbab 4.3.3.*

Soal 4.1: Penyelesaiannya

Kode 4.1 Dari `sort_words.cpp`: pembanding urutan kamus.

```cpp
// 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
}
```

```
Sorted:
Apple
apple
Banana
banana
Cherry
durian
grape
mango
```

***Kode 4.1**: gelung `j >= 0 && comesBefore(key, words[j])` mengandalkan evaluasi hubung-pendek. Java memakai `compareTo` dan `equals`; Python `a.lower()`; jalan pustakanya `sorted(words, key=lambda w: (w.lower(), w))`. Buku: Subbab 4.3.4 dan 4.3.5.*

Soal 4.1: Penyelesaian Keliru yang Menggoda

Pertahankan pengurutannya, bandingkan dengan `<` biasa: keluarannya terurut — menurut **kode karakter**.

```
Apple
Banana
Cherry
apple
banana
durian
grape
mango
```

Cherry berdiri sebelum *apple*. Pada masukan yang seluruhnya berhuruf kecil, kesalahan ini tidak tampak.

> **UJI DIRI.** Urutan kamus terbalik: hanya `comesBefore` yang berubah (`la > lb`, dan `a > b` bila kembar); daftarnya dimulai dengan `mango` dan diakhiri `apple`, `Apple`.

`wrong_sort_words.cpp` adalah alasan langkah 2 meminta contoh buatan sendiri, dan alasan contoh kita sengaja mencampur huruf kapital. Buku: Subbab 4.3.6.

Soal 4.2: Karakter Pertama yang Berulang

> **SOAL 4.2.** Find the first duplicate character in a string. Example 1: `abbcbbca` gives `a`. Example 2: `abbcc` gives `b`. The optimal solution is $O(n)$.

Tafsiran A: berhenti pada karakter pertama yang sudah pernah terlihat — `b` untuk Contoh 1. **Tafsiran B**: karakter paling kiri yang muncul lebih dari sekali — `a`.

Contoh 1 berkata `a`: Tafsiran B. Contoh 2 saja tidak dapat membedakan keduanya. Periksalah *setiap* contoh.

Contoh sendiri: `abcdef` (tidak ada), `Profitina` (`i`), `hello world` (`l`), `xyzzyx` (`x` menurut B, `z` menurut A).

Dinyatakan ulang: hitung setiap karakter; jawabannya karakter paling kiri yang hitungannya paling sedikit 2. Buku: Subbab 4.4.1 dan 4.4.2.

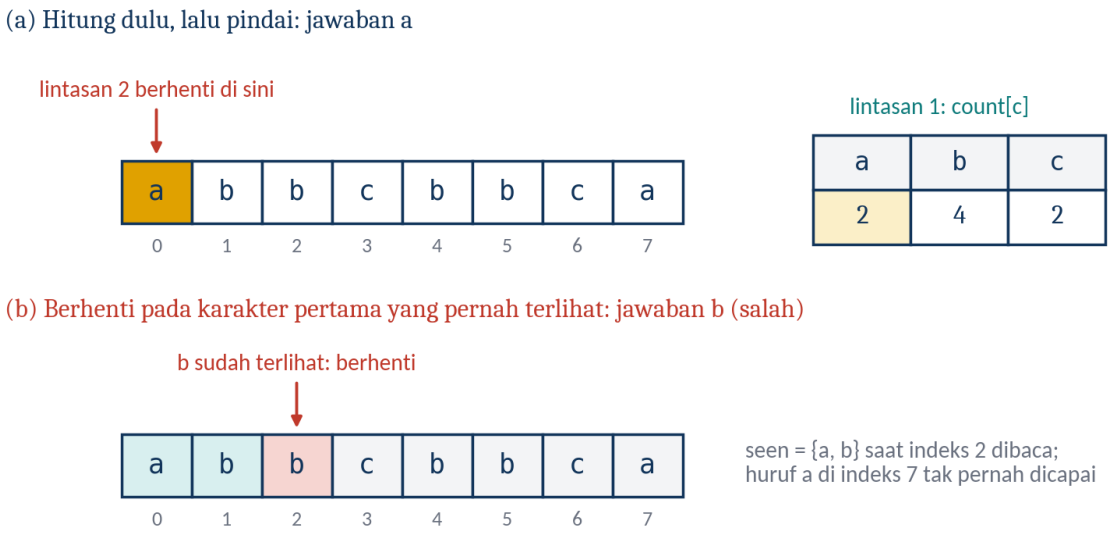
Soal 4.2: Nilainya Adalah Indeksnya

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
```

***Pseudokode 4.2**: semua pasangan akan memakan Persamaan (4.1), sekitar $5 \cdot 10^{11}$ untuk sejuta karakter. Sebagai gantinya, 256 penghitung, satu per byte yang mungkin, diindeks oleh karakternya sendiri: `count['a']` adalah `count[97]`. Dua lintasan, $2n + 256$ langkah: waktu $O(n)$, ruang tambahan $O(1)$. Inilah gagasan tabel hash Bab 15. Buku: Subbab 4.4.3.*

Soal 4.2: Jejaknya



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.

***Gambar 4.3**: (a) lintasan 1 menghasilkan `count['a'] = 2`, `count['b'] = 4`, `count['c'] = 2`, dan lintasan 2 langsung berhenti pada `a`; (b) Tafsiran A berhenti pada `b` kedua, jawaban keliru yang menggoda. Buku: Subbab 4.4.3.*

Soal 4.2: Penyelesaiannya

Kode 4.2 Dari `first_repeat.cpp`: dua lintasan dengan 256 penghitung.

```cpp
// 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;
}
```

```
abbcbbca -> a
abbcc -> b
xyzzyx -> x
```

***Kode 4.2**: `= {0}` menolkan penghitung; `unsigned char` menjaga setiap indeks di 0..255. Tafsiran A yang satu lintasan menjawab `b` untuk `abbcbbca` dan `z` untuk `xyzzyx`. Buku: Subbab 4.4.4 dan 4.4.6.*

Soal 4.3: Mencari Uang Menurut Nama

> **SOAL 4.3.** Make a program in C++ that can search the amounts of money owned by persons. Data: Puspita 50, Bejo 35, Sarita 20, Pujo 45, Lestari 60. `Pujo` gives 45; `Komeng` gives Not found.

Kueri sendiri: `Lestari` (baris terakhir, kasus terburuk, 5 perbandingan), `Puspita` (baris pertama, 1), `pujo` (cocok persis: Not found).

Struktur: satu `struct Person { string name; int money; }` per baris, sehingga nama tidak pernah terpisah dari jumlahnya.

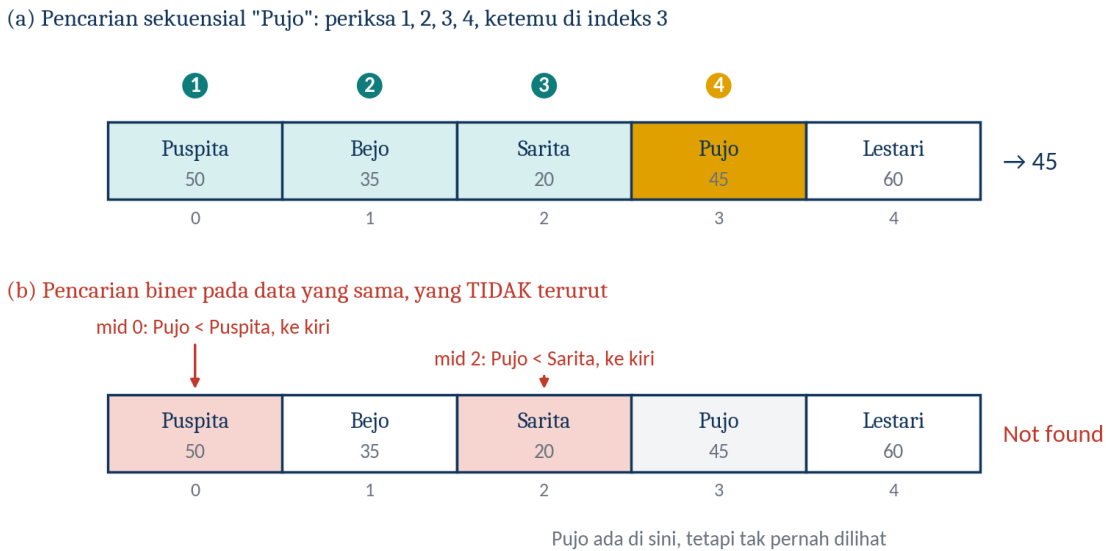
Algoritma: nama-namanya tidak terurut, maka pencarian sekuensial, **Pseudokode 4.3**.

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
```

$O(n)$ per kueri; untuk sejuta nasabah dan sejuta kueri sehari, urutkan sekali lalu cari biner. Buku: Subbab 4.5.1–4.5.3.

Soal 4.3: Dua Pencarian, Satu Larik Tak Terurut



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.

***Gambar 4.4**: (a) pencarian sekuensial menemukan Pujo pada pemeriksaan keempat; (b) pencarian biner memeriksa Sarita, menyimpulkan bahwa Pujo berada di kiri, lalu membuang kotak yang justru memuatnya. Buku: Subbab 4.5.3 dan 4.5.6.*

Soal 4.3: Penyelesaiannya

Kode 4.3 Dari `search_money.cpp`: pencarian sekuensial atas larik rekaman `Person`.

```
// 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
}
```

```
45
Not found
60
Not found
50
```

> ***STRING* JAVA.** Di Java, `people[i].name == name` membandingkan *referensi*; nama yang dibaca dari papan ketik adalah objek baru. Pakailah `equals`.

Kode 4.3 *untuk `Pujo`, `Komeng`, `Lestari`, `pujo`, `Puspita`. Versi pencarian biner mencetak Not found empat kali: prasyarat yang tidak terpenuhi tidak memberi pesan galat, hanya jawaban salah. Buku: Subbab 4.5.4–4.5.6.*

Soal 4.4: Mengurutkan Menurut Kepentingan

> **SOAL 4.4.** Sort strings from most to least important: more `'!'` is more important; with equal `'!'`, the longer string is less important.

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

***Tabel 4.3**: tiga kelompok menurut kunci pertama — 4 tanda, lalu 1, lalu 0 — dan menurut panjang di dalam setiap kelompok. Dua string yang kembar pada kedua kunci mempertahankan urutan masukannya: pengurutannya harus **stabil**. Buku: Subbab 4.6.1 dan 4.6.2.*

Soal 4.4: Dua Kunci dalam Satu Aturan

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
```

Kode 4.4 Dari `importance_sort.cpp`: pembanding dua kunci.

```cpp
// 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
}
```

Pseudokode 4.4 *dan* ***Kode 4.4****: uji panjangnya tegas, sehingga kembar pada kedua kunci menjawab "tidak" ke dua arah dan insertion sort membiarkan pasangan itu di tempatnya. Buku: Subbab 4.6.3 dan 4.6.4.*

Soal 4.4: Hasilnya

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.

***Gambar 4.5**: `Most Important!!!!`, `!important`, `important!`, `meh`, `how about this?`, `not important at all` — persis keluaran pernyataan soal. Dalam Python: `sorted(lines, key=lambda s: (-s.count("!"), len(s)))`. Buku: Subbab 4.6.3–4.6.5.*

Soal 4.4: Dua Pengurutan Stabil, Urutan Keliru

Kode 4.5 Dari `wrong_importance_sort.py`: dua pengurutan stabil, diterapkan dalam urutan yang keliru.

```
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
```

Pengurutan stabil mempertahankan urutan sebelumnya hanya *di antara yang kembar pada kunci saat ini*: pengurutan **terakhir** menentukan urutan utama.

Aturannya: **urutkan menurut kunci yang paling tidak penting dahulu, yang paling penting terakhir**.

> **UJI DIRI.** Tambahkan `Hi!` dan `Yo!`: keduanya bertanda satu dan panjangnya 3, sehingga keluaran dimulai `Most Important!!!!`, `Hi!`, `Yo!`, `!mportant` — `Hi!` lebih dahulu, karena kestabilan.

Kode 4.5 *mengurutkan menurut panjang paling akhir, sehingga panjanglah yang menang. Buku: Subbab 4.6.6.*

Soal 4.5: Alamat Larik dengan Tangan

> **SOAL 4.5.** `int A[10]` dan `int B[3][5]` sama-sama dimulai di `0x1000`. Carilah alamat `A[7]` dan `B[2][3]` untuk `int` 4 byte dan 2 byte, lalu periksalah terhadap memori nyata.

`A[7]`: $7 \cdot 4 = 28$ byte setelah alamat dasar, `0x101C`; dengan 2 byte, `0x100E`.

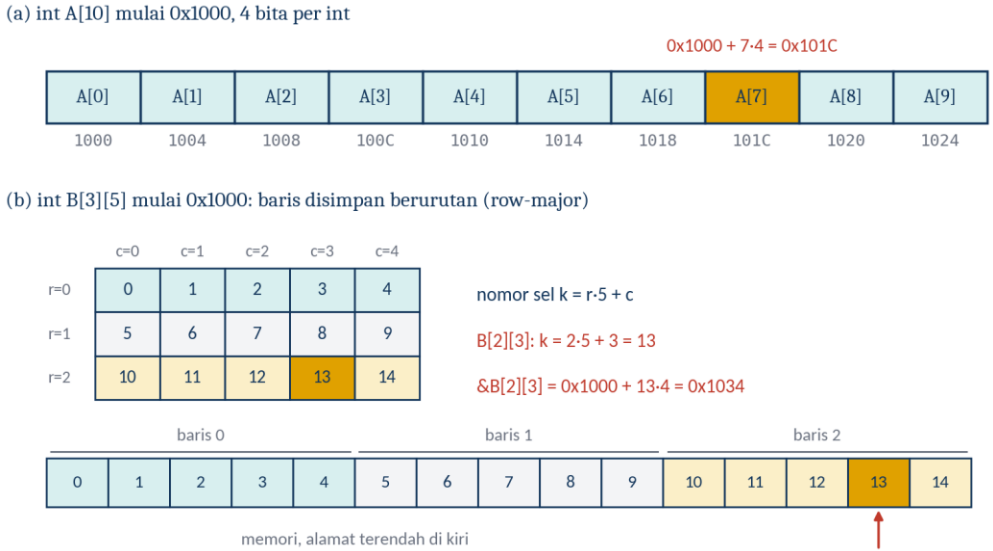
`B[2][3]`: baris-mayor, dua baris penuh berisi 5 lalu 3: 13 sel, 52 byte, `0x1034`; dengan 2 byte, `0x101A`.

$$\text{addr}(A[i]) = b + i \cdot s \tag{4.3}$$

$$\text{addr}(B[r][c]) = b + (r \cdot C + c) \cdot s \tag{4.4}$$

Persamaan (4.3) dan Persamaan (4.4): banyaknya baris tidak pernah masuk ke alamat. Buku: Subbab 4.7.1–4.7.3.

Soal 4.5: Memori, Digambarkan



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.

***Gambar 4.6**: (b) menomori sel-sel **B** menurut urutan memori, sehingga $k = r \cdot 5 + c$ dapat dibaca langsung. Buku: Subbab 4.7.3.*

Soal 4.5: Rumusnya, Diperiksa

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
```

```
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
```

Pseudokode 4.5 *berwaktu konstan — sifat utama larik, dan sifat yang dilepaskan linked list.* `array_address.cpp` *mencetak kedua jawaban untuk kedua ukuran, lalu mengukur jarak sungguhan di memori: 28 dan 52 byte. Buku: Subbab 4.7.3 dan 4.7.4.*

Soal 4.5: Dua Kekeliruan

Kode 4.6 Dari `wrong_array_address.py`: melupakan ukuran elemen, dan mengalikan dengan banyaknya baris.

```
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
```

***Kode 4.6**: `0x1007` bukan kelipatan 4 dari alamat dasar; `0x1024` kurang dari dua baris penuh (40 byte) di atasnya. Uji diri: `double M[4][6]` di `0x2000` menaruh `M[3][5]` di `0x20B8`, dan byte pertama sesudah larik di `0x20C0`. Buku: Subbab 4.7.5 dan 4.7.6.*

Soal 4.6: Menghitung Kunci dalam Larik Terurut

> **SOAL 4.6.** Tahun terbit di satu rak, terurut: 1998, 2001, 2001, 2001, 2005, 2010, 2010, 2019, 2023. Berapa buku dari tahun 2001, 2010, 2002, 2023, 1990, dan 2030, dalam waktu $O(\log n)$?

Nilai yang sama berdiri berdampingan dalam satu blok: 2001 menempati indeks 1 sampai 3, jadi hitungannya $4 - 1 = 3$.

Tahun yang tidak ada memiliki blok kosong: 2002 “mulai” dan berakhir di 4. Blok 2030 terletak lewat ujung, di indeks 9.

$$\text{count}(key) = \text{upper}(key) - \text{lower}(key) \tag{4.5}$$

Persamaan (4.5): lower bound adalah indeks pertama dengan $a[i] \geq key$, upper bound yang pertama dengan $a[i] > key$; keduanya boleh bernilai n. Buku: Subbab 4.8.1–4.8.3.

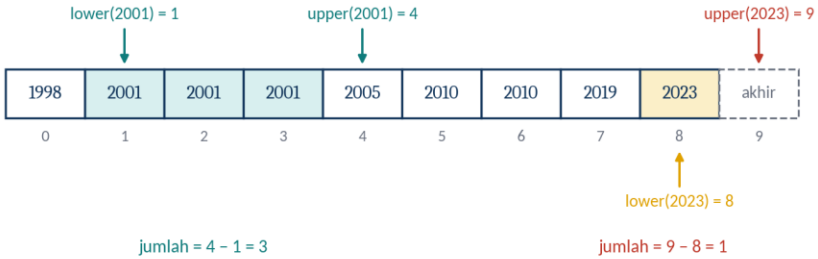
Soal 4.6: Pencarian Biner untuk Sebuah Batas

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
```

***Pseudokode 4.6**: rentangnya dimulai sebagai `[0, n]`, termasuk posisi lewat ujung, karena jawabannya boleh n. `hi = mid` mempertahankan calon; `lo < hi` berhenti bila tinggal satu. Buku: Subbab 4.8.3.*

Soal 4.6: Jejaknya



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.

***Gambar 4.7**: `LOWER-BOUND(2001)` menyusutkan `[0, 9]` menjadi `[0, 4]`, `[0, 2]`, `[0, 1]`, `[1, 1]`: empat pemeriksaan, seperti diizinkan Persamaan (4.2). Buku: Subbab 4.8.3.*

Soal 4.6: Penyelesaiannya, dan Meleset Satu

Kode 4.7 Dari `count_range.cpp`: *lower bound* pada rentang `[0, n]`.

```cpp
// 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;
}
```

```
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
```

***Kode 4.7**: C++ menyediakan `std::lower_bound`, `std::upper_bound`, dan `std::equal_range`; Python `bisect_left` dan `bisect_right`. Mulai dari `[0, n-1]` tidak pernah dapat mengembalikan 9, sehingga 2023 mendapat hitungan 0. Uji diri: tahun 2000–2010 adalah* $\text{lower}(2011) - \text{lower}(2000) = 7 - 1 = 6$. *Buku: Subbab 4.8.4–4.8.6.*

Soal 4.7: Pencarian Interpolasi pada Kode Buku

SOAL 4.7. Kode 025, 034, 041, 056, 063, 072, 088, 096. Carilah 88, 60, 41, 25, 96, dan 100 dengan pencarian interpolasi, cetak setiap pemeriksaan, dan bandingkan dengan pencarian biner.

$$pos = low + \left\lfloor \frac{(key - a[low]) \cdot (high - low)}{a[high] - a[low]} \right\rfloor \tag{4.6}$$

88: $\lfloor 63 \cdot 7/71 \rfloor = \lfloor 6{,}21 \rfloor = 6$, dan $a[6] = 88$: satu pemeriksaan.

60: $\lfloor 245/71 \rfloor = 3$, $a[3] = 56 < 60$, `low = 4`; kini $a[4] = 63 > 60$: tidak ada, setelah satu pemeriksaan.

41: periksa 1 ($a[1] = 34$), lalu $2 + 0 = 2$: ditemukan setelah dua pemeriksaan.

Persamaan (4.6): periksa pecahan rentang indeks yang sama dengan pecahan rentang nilai tempat kunci berada. Buku: Subbab 4.9.1 dan 4.9.2.

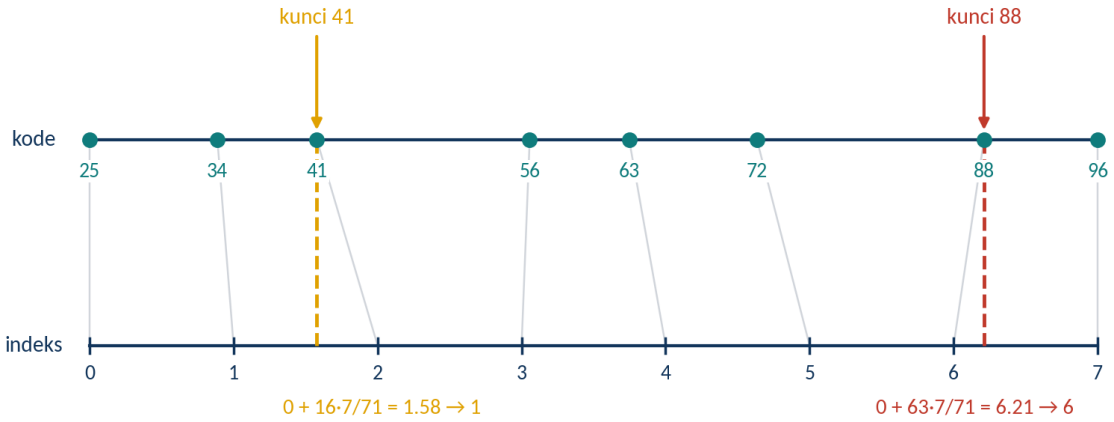
Soal 4.7: Dua Penjaga

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
```

Pseudokode 4.7*: uji rentang pada baris 2 menjaga pecahannya di antara 0 dan 1, sehingga setiap pemeriksaan berada di dalam larik; baris 3 mencegah pembagian dengan nol. Sekitar $log_2 log_2 n$ pemeriksaan pada kunci yang tersebar merata, tetapi $O(n)$ pada kunci yang miring. Buku: Subbab 4.9.3.*

Soal 4.7: Memproyeksikan Nilai ke Indeks



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.

***Gambar 4.8**: 88 jatuh di 6,21 dan ditemukan di indeks 6; 41 jatuh di 1,58 dan memerlukan pemeriksaan kedua. Buku: Subbab 4.9.3.*

Soal 4.7: Penyelesaiannya

Kode 4.8 Dari `interpolation_search.cpp`: gelung berpenjaga; perkaliannya dibentuk sebelum pembagian.

```
    // 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]);
        }
        if (codes[pos] == key) {
            return pos;
        } else if (codes[pos] < key) {
            low = pos + 1;
        } else {
            high = pos - 1;
        }
    }
```

```
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)
```

***Kode 4.8**: membagi lebih dahulu hampir selalu menghasilkan 0. Tanpa penjaga rentang, kunci 200 memeriksa indeks 17: `IndexError` di Python, sampah di C++. Buku: Subbab 4.9.4 dan 4.9.6.*

Soal 4.8: Menelusuri Pengurutan Dasar

SOAL 4.8. Urutkan 5, 6, 8, 1, 3, 25, 10 dengan bubble sort, selection sort, dan insertion sort; tuliskan larik setelah setiap lintasan dan hitung perbandingan serta pertukaran atau pergeserannya.

Hitung inversinya lebih dahulu: 5, 6, dan 8 masing-masing mengalahkan 1 dan 3; 25 mengalahkan 10: $2 + 2 + 2 + 1 = 7$.

Bubble sort: 21 perbandingan (Persamaan (4.1)), satu pertukaran per inversi: **7**. Selection sort: 21 perbandingan, paling banyak 6 pertukaran. Insertion sort: 7 pergeseran, dan lebih sedikit perbandingan.

Ketiganya $O(n^2)$ pada larik terbalik; pada larik terurut insertion sort hanya memerlukan $n - 1$ perbandingan. Buku: Subbab 4.10.1–4.10.3.

Soal 4.8: Ketiga Pengurutan

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
```

***Pseudokode 4.8**: ketiga pengurutan persis seperti didefinisikan Bab 3. Buku: Subbab 4.10.3.*

Soal 4.8: Lintasan Pertama Bubble Sort

Bubble sort, lintasan 1: bandingkan tetangga j dan j+1, tukar bila terbalik

awal	5	6	8	1	3	25	10	
j = 0	5	6	8	1	3	25	10	5 ≤ 6: tetap
j = 1	5	6	8	1	3	25	10	6 ≤ 8: tetap
j = 2	5	6	1	8	3	25	10	8 > 1: tukar
j = 3	5	6	1	3	8	25	10	8 > 3: tukar
j = 4	5	6	1	3	8	25	10	8 ≤ 25: tetap
j = 5	5	6	1	3	8	10	25	25 > 10: tukar

setelah lintasan 1 nilai terbesar, 25, berada di tempat akhirnya (indeks 6)

Gambar 4.9 Enam perbandingan pada lintasan pertama bubble sort; pasangan kuning ditukar, pasangan hijau toska sudah berurutan.

***Gambar 4.9**: enam perbandingan, tiga pertukaran (8 dengan 1, 8 dengan 3, 25 dengan 10); 25 terbawa seperti gelembung ke ujung kanan, dan tinggal di sana. Buku: Subbab 4.10.3.*

Soal 4.8: Hitungannya

```
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
```

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

***Tabel 4.4**: setiap ramalan terbukti. Bubble sort sudah menyelesaikan kerja nyatanya setelah lintasan 3; penanda `swapped` akan menghentikannya setelah lintasan 4. Uji diri: pada larik terbalik insertion sort melakukan 21 perbandingan dan 21 pergeseran; pada yang terurut, 6 dan 0. Buku: Subbab 4.10.4.*

Soal 4.8: Satu Langkah Terlalu Jauh

Kode 4.9 Dari `WrongBubble.java`: gelung dalam berjalan satu langkah terlalu jauh.

```
        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;
                }
            }
        }
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 7 out of bounds for length 7
    at WrongBubble.main(WrongBubble.java:9)
```

***Kode 4.9**: pada $i = 0$ nilai `j` terakhir adalah 6, dan `a[j + 1]` membaca `a[7]`. Java menyebut indeks yang salah, panjangnya, dan barisnya — sebuah hadiah. Bug yang sama di C++ membaca, bahkan mungkin menulis, memori yang bukan miliknya, diam-diam; karena itulah `-fsanitize=address`. Buku: Subbab 4.10.6.*

Soal 4.9: Menggabungkan Dua Daftar Terurut

SOAL 4.9. Cabang A: 3, 9, 12, 15, 31. Cabang B: 5, 9, 17, 23, 35, 40, 41. Hasilkan satu katalog terurut dalam satu lintasan, dan hitung perbandingannya.

Dua tumpukan kartu terbuka: ambil kartu teratas yang lebih kecil. Pada kembar 9–9, ambil milik A lebih dahulu (stabil).

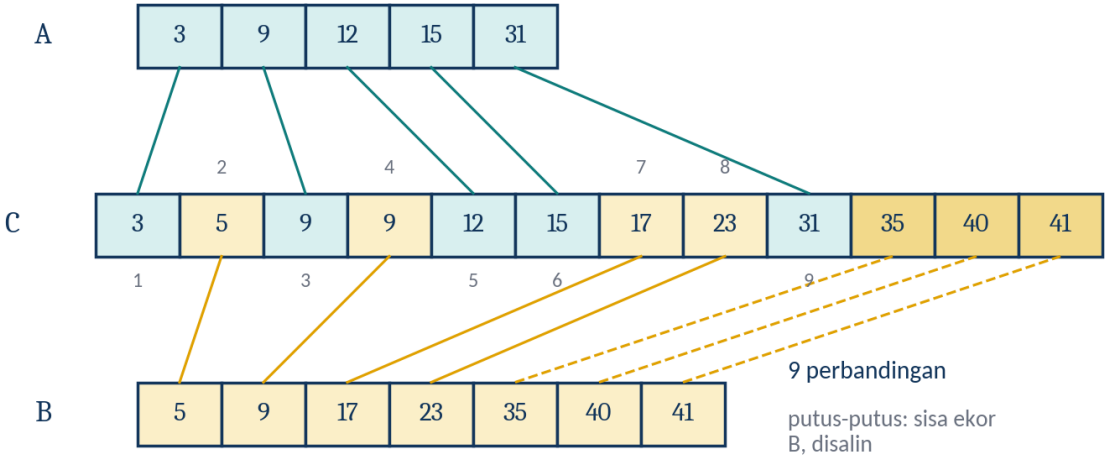
Setelah 31, A kosong: sembilan perbandingan, dan 35, 40, 41 disalin tanpa perbandingan sama sekali.

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
```

***Pseudokode 4.9**: setiap elemen yang diambil adalah yang terkecil di antara yang belum diambil, sehingga C tumbuh terurut. Paling banyak $n + m - 1$ perbandingan: $O(n + m)$. Buku: Subbab 4.11.1–4.11.3.*

Soal 4.9: Asal Setiap Elemen



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.

***Gambar 4.10**: garis hijau-kebiruan dari A, jingga dari B, putus-putus untuk ekor B yang disalin tanpa perbandingan. Mengurutkan dua belas kode dari awal dapat memakan 66 perbandingan. Buku: Subbab 4.11.3.*

Soal 4.9: Penyelesaiannya

Kode 4.10 Dari `merge_sorted.cpp`: penggabungan, dengan menghitung perbandingan.

```
    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++];
    }
```

```
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
```

***Kode 4.10**: `c[k++] = a[i++]` menyalin, lalu memajukan keduanya. Tinggalkan kedua gelung penyalin dan hanya 9 dari 12 kode yang bertahan — periksalah bahwa keluarannya berisi $n + m$ elemen. Pustaka: `std::merge`, `heapq.merge`. Buku: Subbab 4.11.4–4.11.6.*

Soal 4.10: Menghapus Nilai di Tempat

SOAL 4.10. Hapus setiap kemunculan x dari 7, 3, 3, 5, 3, 9 di tempat, dengan mempertahankan urutan, lalu laporkan ukuran barunya. Uji $x = 3$,7,9,4, dan 3, 3, 3 dengan $x = 3$.

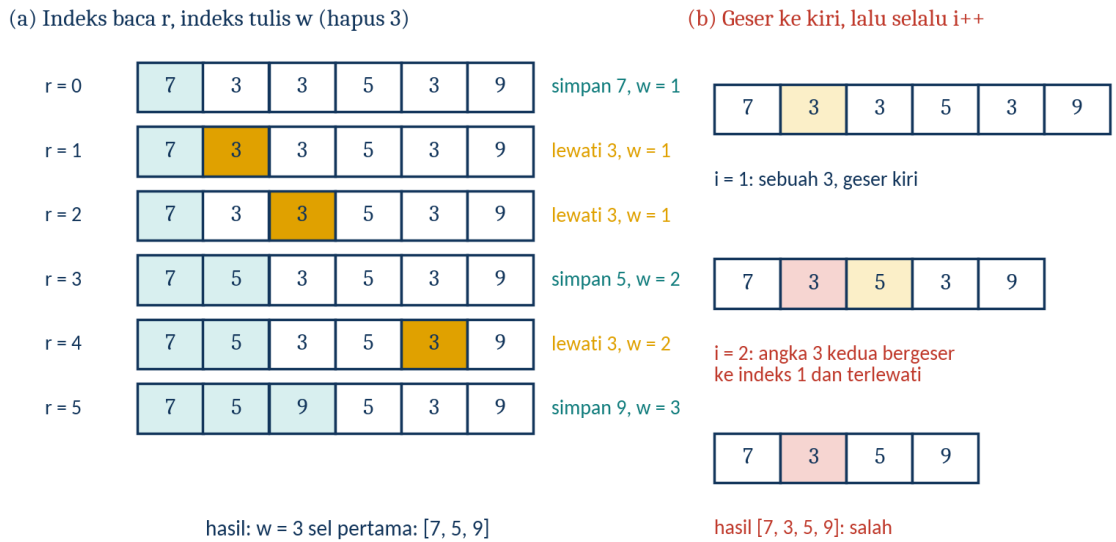
Larik tidak dapat memendek: pindahkan bilangan yang dipertahankan ke depan dan ingat berapa yang terhitung — **ukuran** lawan **kapasitas**.

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
```

***Pseudokode 4.10**: invarian — `a[0..w-1]` memuat, berurutan, tepat elemen-elemen yang dipertahankan dari `a[0..r-1]`; karena `w <= r`, menulis tidak pernah merusak elemen yang belum dibaca. Waktu $O(n)$, ruang $O(1)$, dibandingkan $O(n^2)$ untuk penggeseran berulang. Buku: Subbab 4.12.1–4.12.3.*

Soal 4.10: Indeks Baca dan Indeks Tulis



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.

***Gambar 4.11**: (a) `r` mengunjungi setiap kotak sekali sementara `w` memadatkan bilangan yang dipertahankan ke depan; (b) menggeser ke kiri lalu selalu maju melewatkan angka 3 kedua dari dua 3 yang berdampingan. Buku: Subbab 4.12.3.*

Soal 4.10: Penyelesaian dan Jebakannya

Kode 4.11 Dari `delete_value.cpp`: satu lintasan dengan indeks baca dan indeks tulis.

```cpp
// 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;
}
```

```
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)
[7, 3, 5, 9]
```

Kode 4.11 *adalah* `std::remove`*, yang memadatkan tetapi tidak mengubah ukuran. Versi geser-kiri mencetak* `[7, 3, 5, 9]`*: satu angka 3 lolos. Uji diri: membuang duplikat dari larik terurut mempertahankan* `a[r]` *bila* `a[r] != a[w - 1]` *— itulah* `std::unique`*. Buku: Subbab 4.12.4–4.12.6.*

Soal 4.11: Dua Harga yang Cocok dengan Voucher

SOAL 4.11. Harga Profitina Laundry: 18000, 7000, 25000, 12000, 30000, 5000, 20000, 15000. Untuk voucher 37000, 12000, dan 11000, daftarkan setiap pasangan dua layanan berbeda yang cocok, dengan langkah lebih sedikit daripada mencoba semua pasangan.

Semua pasangan: $8 \cdot 7/2 = 28$. Dengan tangan: $7000 + 30000$ dan $12000 + 25000$; $5000 + 7000$; tidak ada untuk 11000.

Urutkan, lalu **berjalan ke dalam**: terlalu kecil, buang yang termurah; terlalu besar, buang yang termahal.

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
```

Pseudokode 4.11*: harga yang dibuang tidak mungkin menjadi anggota pasangan dengan harga-harga yang masih di antara kedua indeks, sehingga tidak ada pasangan yang terlewat. $O(n \log n)$ untuk pengurutan, $O(n)$ untuk jalannya. Buku: Subbab 4.13.1–4.13.3.*

Soal 4.11: Jalan untuk 37000



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.

***Gambar 4.12**: lima langkah alih-alih 28 pasangan, dalam ribuan rupiah: setiap langkah menemukan pasangan atau membuang harga termurah atau termahal yang tersisa. Buku: Subbab 4.13.3.*

Soal 4.11: Penyelesaiannya

Kode 4.12 Dari `pair_sum.cpp`: jalan dua indeks pada harga yang terurut.

```
    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
        }
    }
```

```
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)
```

***Kode 4.12**: pada harga yang tidak terurut jalan ini mencetak "no pair" tiga kali — prasyaratnya tidak memberi pesan galat. Alternatif Python: himpunan harga yang sudah terlihat, rata-rata $O(n)$. Buku: Subbab 4.13.4–4.13.6.*

Soal-Soal Berdampingan

Tabel 4.5 Kesebelas soal: gagasan, biaya, dan jebakan penyelesaian keliru yang menggoda.

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

Tabel 4.5 (Lanjutan) *Kesebelas soal: gagasan, biaya, dan jebakan penyelesaian keliru yang menggoda.*

Soal	Gagasan kunci	Biaya	Jebakan penyelesaian keliru yang menggoda
4.11 Pasangan voucher	urutkan, lalu berjalan ke dalam	$O(n \log n)$	berjalan pada data tak terurut

Tabel 4.5*: prasyarat, batas, dan contoh tepi. Buku: Subbab 4.14.*

Profitina dalam Praktik

Papan skor dengan pemecah kembar adalah Soal 4.4 dalam kostum lain: kunci utama dan urutan kedua yang stabil, sehingga daftar tidak berkedip.

“Semua pemindaian antara 1 Juli dan 30 September” adalah uji diri Soal 4.6: dua pencarian batas, $O(\log n)$.

Kokpit dasbor menghimpun banyak jenis notifikasi, masing-masing terurut menurut waktu: penggabungan Soal 4.9, diperumum.

Penghitung “N baru” menghitung per kanal dalam satu lintasan: gagasan menghitung dari Soal 4.2.

Laundry itu kembali dengan antrean pesanan di Bab 5, daftar pelanggan di Bab 7, dan tabel harga di Bab 15. Buku: Subbab 4.15.

Kesalahan yang Sering Terjadi

Pencarian biner, interpolasi, penggabungan, atau jalan dua indeks pada data tidak terurut: tanpa galat, hanya jawaban salah.

Kode karakter untuk urutan kamus; salah membaca pernyataan pendek; hanya memeriksa contoh pertama.

Gelung satu langkah terlalu jauh; pencarian batas pada $[0, n-1]$; `(lo + hi) / 2` pada larik raksasa.

Kunci pengurutan multilintasan dalam urutan keliru; ukuran elemen terlupa dalam alamat; membagi sebelum mengalikan.

Ekor penggabungan terlupa; maju setelah menghapus dengan menggeser; *string* Java dibandingkan dengan `==`.

Setiap penyelesaian keliru yang menggoda hari ini lolos paling sedikit satu contoh pernyataan soalnya. Buku: Subbab 4.16.

Ringkasan: Lima Hal yang Perlu Diingat

1. Delapan langkah: nyatakan ulang, contoh dengan tangan, struktur dan algoritma, jejak, biaya, kode, bandingkan, tinjau kembali.

2. Perkirakan sebelum menulis kode: semua pasangan memakan Persamaan (4.1), pencarian biner paling banyak Persamaan (4.2).

3. Pengurutan hanya memerlukan aturan “mana yang lebih dahulu”; ganti aturannya, bukan pengurutannya (Soal 4.1, 4.4).

4. Menghitung (4.2), batas (4.6), dua indeks (4.10, 4.11), dan penggabungan (4.9) mengubah gagasan kuadratik menjadi linear atau logaritmik.

5. Prasyarat dan tepi: ketahui apa yang diandaikan algoritma Anda, dan ujilah yang pertama, yang terakhir, yang hilang, dan yang kembar.

Bab 5 membangun stack dan queue tepat di atas gagasan larik-dengan-ukuran dari Soal 4.10. Buku: Subbab 4.17.

Pertemuan Berikutnya

Bab 5 — Stack dan Queue. Terakhir masuk pertama keluar dan pertama masuk pertama keluar, di atas larik: *push* dan *pop*, *enqueue* dan *dequeue*.

Menara Hanoi, pencocokan kurung, *undo* dan *redo*, infiks ke pascafiks, queue sirkular, dan deque.

Sebelum itu: kerjakan Soal Latihan 4.1, 4.3, 4.6, dan 4.10 dari Bab 4, dengan tangan dahulu, lalu dengan kode.

Setiap soal latihan memiliki penyelesaian lengkap di buku; cobalah sebelum melihatnya. Buku: Subbab 4.18 dan 4.19.

Kuliah 5 — Stack dan Queue

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 5

Dosen: Irfan Subakti — yifana@gmail.com

Hari ini kita sengaja mencabut kebebasan larik: stack hanya menyentuh puncaknya, queue hanya kedua ujungnya, dan pembatasan itulah yang membuat setiap operasi berwaktu konstan. Buku: Bab 5, Subbab 5.1–5.20.

Cakupan Pertemuan Ini

1. **Stack** — Menara Hanoi, ADT-nya, stack berbasis larik dalam tiga bahasa, stack yang tumbuh dan stack bertaut.
2. **Empat tugas sebuah stack** — mencocokkan tanda kurung, *undo* dan *redo*, infiks ke postfiks beserta evaluasinya, dan *call stack* fungsi.
3. **Queue** — ADT-nya, queue larik yang menggeser dan yang bergeser, serta queue sirkular yang menyembuhkan keduanya.
4. **Penuh atau kosong?** — pencacah, atau satu slot dibiarkan kosong; deque; sekilas *priority queue*.
5. **Pustaka standar**, ringkasan biaya, Profitina, dan kesalahan yang sering terjadi.

Setiap struktur digambar keadaan demi keadaan sebelum kodenya diperlihatkan. Buku: Subbab 5.1.

Posisi Kita: dari Akses Bebas ke Disiplin

Bab 1 sampai 4 membiarkan algoritma membaca dan menulis elemen **mana pun** dalam urutan apa pun: larik sebagai ruangan berpintu terbuka.

Troli pengembalian buku Pustaka Ceria: buku ditaruh di atas dan diambil dari atas. **Antrean buku baru**-nya: yang datang pertama, dikatalogkan pertama.

Hanya menyentuh satu posisi yang diketahui memberi **kesederhanaan** (tanpa pencarian, tanpa penggeseran: $O(1)$), **kebenaran karena konstruksi**, dan **kecocokan dengan masalahnya**.

Membatalkan suntingan, mencocokkan kurung, melayani pelanggan, dan mencetak dokumen pada dasarnya bersifat terakhir-masuk-pertama-keluar atau pertama-masuk-pertama-keluar. Buku: Subbab 5.1 dan 5.1.1.

Tipe Data Abstrak

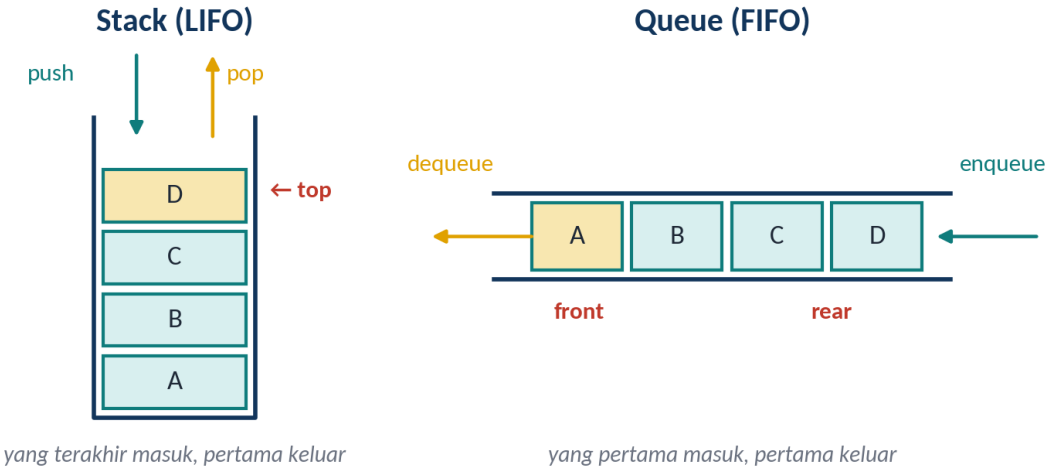
> **DEFINISI.** *Tipe data abstrak* (ADT) adalah sekumpulan nilai **beserta operasi yang diizinkan atasnya dan apa yang dilakukan operasi-operasi itu**, tanpa keterikatan pada cara nilai-nilai itu disimpan. *Struktur data* adalah satu cara konkret menyimpannya agar operasi-operasi itu dapat dijalankan.

"Stack" adalah ADT: `push`, `pop`, `peek`, `isEmpty`, dan `pop` mengembalikan nilai yang paling akhir di-*push* dan masih ada.

Larik dengan indeks `top`, rangkaian simpul bertaut, dan `list` Python adalah tiga struktur data yang mengimplementasikannya.

Kode yang memakai stack hanya melalui operasinya bekerja tanpa perubahan dengan implementasi mana pun. Buku: Subbab 5.1.2.

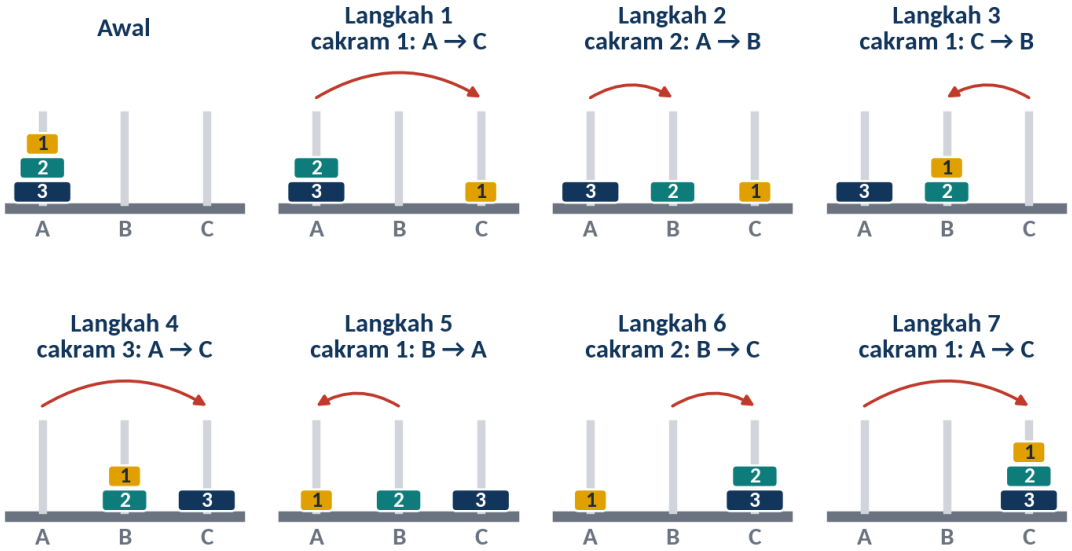
Dua Disiplin



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).

***Gambar 5.1**: stack masuk dan keluar melalui satu bukaan, terakhir masuk pertama keluar (**LIFO**); queue masuk di belakang dan keluar dari depan, pertama masuk pertama keluar (**FIFO**). Buku: Subbab 5.1.2.*

Sebuah Kisah: Menara Hanoi



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.

***Gambar 5.2**: pindahkan satu cakram setiap kali, hanya cakram teratas, dan jangan pernah di atas cakram yang lebih kecil. Setiap tiang adalah stack. Édouard Lucas menjual teka-teki ini pada 1883 dengan legenda 64 cakram emas. Buku: Subbab 5.2 dan 5.2.1.*

Contoh Terkerjakan 1: Tiga Cakram, dan Hitungannya

Langkah 1–3 memarkir cakram 1 dan 2 di B; **langkah 4** satu-satunya gerakan cakram 3; **langkah 5–7** membawa menara dua cakram ke atasnya.

Menyelesaikan n cakram berarti menyelesaikan $n-1$ dua kali dengan satu langkah di antaranya, Persamaan (5.1):

$$M(n) = 2M(n-1) + 1, \qquad M(1) = 1. \tag{5.1}$$

$M(1) = 1, M(2) = 3, M(3) = 7, M(4) = 15, M(5) = 31$: satu kurangnya dari pangkat dua, Persamaan (5.2):

$$M(n) = 2^n - 1. \tag{5.2}$$

Periksa: $2(2^{n-1} - 1) + 1 = 2^n - 1$. *Bab 10 membuktikan hasil seperti ini dengan induksi. Buku: Subbab 5.2.1 dan 5.2.2.*

Dunia Masih Aman untuk Sementara

64 cakram memerlukan $2^{64} - 1 = 18.446.744.073.709.551.615$ langkah. Dengan satu langkah per detik, Persamaan (5.3):

$$\frac{2^{64} - 1}{60 \cdot 60 \cdot 24 \cdot 365.25} \approx 5.85 \times 10^{11} \text{ tahun}, \tag{5.3}$$

Sekitar 585 miliar tahun: kira-kira empat puluh kali umur alam semesta sekarang.

> **PELAJARANNYA.** Algoritma yang biayanya berlipat dua untuk setiap tambahan satu satuan masukan menjadi tanpa harapan jauh sebelum masukannya tampak besar.

Buku: Subbab 5.2.2.

Hanoi dengan Tiga Stack tanpa Rekursi

Kode 5.1 Dari `01_hanoi_stacks.cpp`: satu langkah adalah *push* ke tiang tujuan dan *pop* dari tiang asal, ditolak bila melanggar aturan ukuran.

```cpp
// 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;
}
```

```
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]
```

Kode 5.1*: pada langkah ganjil cakram 1 melangkah menurut siklus tetap; pada langkah genap dibuat satu-satunya langkah sah yang tidak melibatkan cakram 1. Tidak ada yang mengindeks tengah sebuah tiang; gelungnya melakukan tepat* $2^n - 1$ *langkah untuk* $n = 1..10$*. Buku: Subbab 5.2.3.*

ADT Stack

DEFINISI. *Stack* adalah kumpulan tempat penyisipan dan penghapusan sama-sama terjadi di satu ujung, yaitu *puncak* (*top*); elemen yang dihapus selalu yang paling akhir disisipkan di antara yang masih ada: **terakhir masuk, pertama keluar**.

Tumpukan piring, tumpukan buku, tabung kok bulu tangkis: tambah di atas, ambil dari atas.

Dalam larik $S[0..N-1]$ dengan indeks top, konvensi $top = -1$ berarti kosong, sehingga Persamaan (5.4):

$$\text{size} = top + 1. \tag{5.4}$$

Menginisialisasi $top = 0$ lalu menguji $top = -1$ mencampur dua konvensi: stack yang tidak pernah kosong.

Sebagian buku membiarkan top menghitung elemen; keduanya benar, mencampurnya tidak. Buku: Subbab 5.3 dan 5.3.2.

Operasi dan Biayanya

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

***Tabel 5.1**: push ke stack terbatas yang penuh adalah **overflow**; pop atau peek pada stack kosong adalah **underflow**. Tolak, lempar eksepsi, atau tumbuh — tetapi jangan pernah membaca `data[-1]`. Buku: Subbab 5.3.1.*

Operasi dalam Pseudokode

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

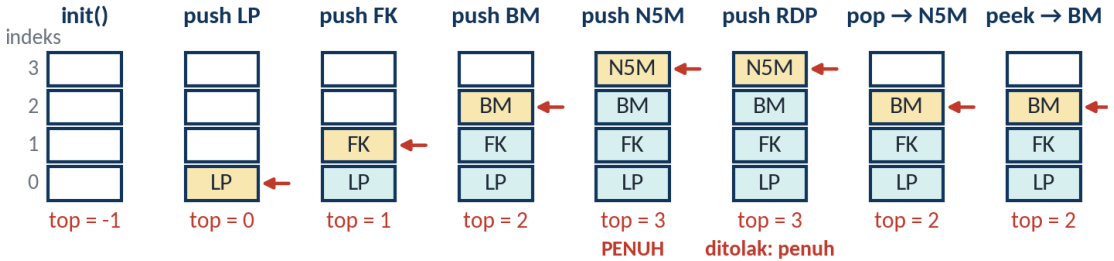
STACK-FULL(S)
 2.  return S.top = N - 1

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
```

***Pseudokode 5.1**: `PUSH` naik dahulu, lalu menulis; `POP` membaca dahulu, lalu turun: `data[++top] = x`, `return data[top--]`. `PEEK` membaca tanpa bergerak. Buku: Subbab 5.3.2.*

Mengamati Stack Berbasis Larik



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.

***Gambar 5.3**: empat push mengisi troli berkapasitas 4; yang kelima ditolak; `pop` mengembalikan N5M dan menurunkan penanda — nilainya tetap di slot 3 sampai ditimpa. Menghapus tidak berarti menghapus isi. Buku: Subbab 5.4.1.*

Stack dalam C++

Kode 5.2 Dari `02_array_stack.cpp`: `push` melaporkan *overflow* dengan `false`; `pop` dan `peek` melempar eksepsi pada *underflow*; `clear` cukup satu penugasan.

```
    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
```

```
push Negeri 5 Menara -> ok, size 4, top index 3
push Ronggeng Dukuh Paruk -> REJECTED, the cart is full
pop  -> Negeri 5 Menara, size 3, top index 2
```

***Kode 5.2**: setiap operasi $O(1)$. Penyalinan dihapus, sehingga satu larik tidak pernah dibebaskan dua kali. Java mengosongkan slot yang di-pop; Python melempar `IndexError`. Buku: Subbab 5.4.2–5.4.4.*

Stack yang Tumbuh

Kode 5.3 Dari `03_growing_stack.cpp`: bila larik penuh, alokasikan larik dua kali lebih besar lalu salin.

```
    void push(int value) {
        if (top_ == capacity_ - 1) grow();              // full: double first
        data_[++top_] = value;
    }
    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;
    }
```

```
pushing 1..17 onto a stack that starts with capacity 1
  grow: capacity 16 -> 32, copied 16 element(s)
size 17, capacity 32, total copies 31 (less than 2n = 34)
```

***Kode 5.3**: push yang mendapati 16 elemen dalam 16 slot menyalin keenam belasnya lebih dahulu. Jadi, apakah `push` masih $O(1)$? Tidak untuk setiap push — tetapi secara rata-rata, ya. Buku: Subbab 5.5.1.*

$O(1)$ Teramortisasi

n kali *push* dari kapasitas 1 menggandakan larik pada ukuran $1,2,4,\ldots,2^k < n$, dan salinannya berjumlah Persamaan (5.5):

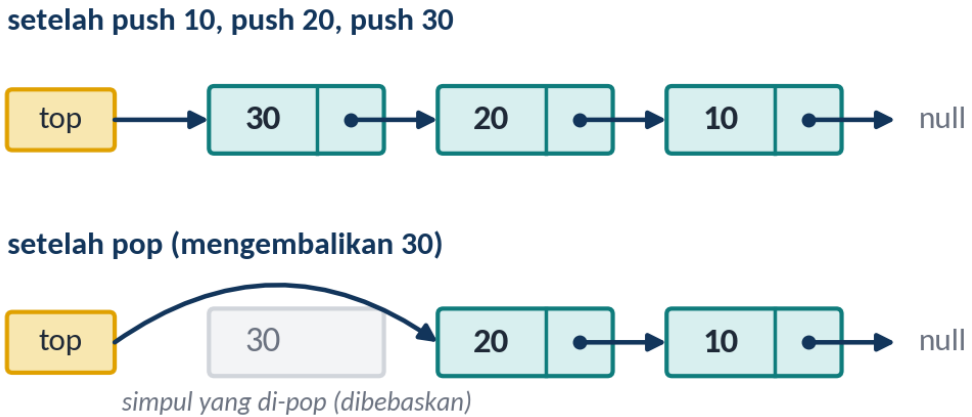
$$1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 < 2n. \tag{5.5}$$

n kali menulis ditambah kurang dari $2n$ salinan: rata-rata kurang dari 3 operasi per *push*.

Tumbuh tetap 10 slot setiap kali akan menyalin sekitar $n^2/20$: $O(n)$ per *push*.

Push mahal yang sesekali terjadi dibayar oleh banyak push murah sebelumnya. `std::vector`, `ArrayList`, `ArrayDeque`, *dan* `list` *Python semuanya tumbuh dengan faktor konstan. Buku: Subbab 5.5.1.*

Sekilas: Stack Bertaut



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.

***Gambar 5.4**: push adalah `top_ = new Node{v, top_};`; pop melepas taut dengan `top_ = old->next;` sebelum `delete old;`. Tidak pernah penuh, $O(1)$ pada kasus terburuk, dengan harga satu alokasi dan satu pointer per elemen. Bab 6 dan 7 menjelaskan pointernya. Buku: Subbab 5.5.2.*

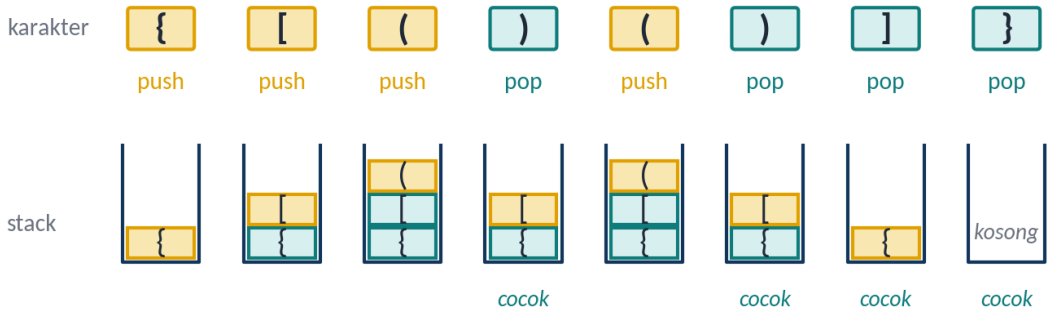
Mencocokkan Tanda Kurung

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"
```

***Pseudokode 5.2**: menghitung saja tidak cukup — `([)]` memiliki dua dari tiap jenis namun salah. Setiap penutup harus cocok dengan pembuka terbaru yang belum dicocokkan: puncak sebuah stack. Tiga cara gagal: penutup tanpa pembuka (baris 8), tidak cocok (baris 10), pembuka yang tidak pernah ditutup (baris 14). Buku: Subbab 5.6 dan 5.6.1.*

Jejak Tanda Kurung



Gambar 5.5 Memeriksa `{[()()]}`: setiap pembuka di-*push*, setiap penutup mengambil pasangannya dari puncak, dan stack kosong di akhir, sehingga teksnya seimbang.

***Gambar 5.5**: `{[()()]}` — setiap pembuka di-push, setiap penutup mem-pop pasangannya, dan stack kosong di akhir: seimbang. Buku: Subbab 5.6.1.*

Tanda Kurung dalam C++

Kode 5.4 Dari `05_brackets.cpp`: kasus-kasus kurung penutup dan pemeriksaan akhir, dengan posisi kesalahan pertama.

```
        } 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";
```

```
"{[()()]}"                  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
```

***Kode 5.4**, dengan `std::stack<char>`: setiap ragam kegagalan muncul sekali. Satu lintasan, kerja stack $O(1)$ per karakter: waktu $O(n)$ dan, untuk n pembuka, ruang $O(n)$. Buku: Subbab 5.6.2.*

Undo dan Redo: Dua Stack

Sunting: *push* keadaan kini ke *undo*, lakukan suntingan, **kosongkan *redo***.

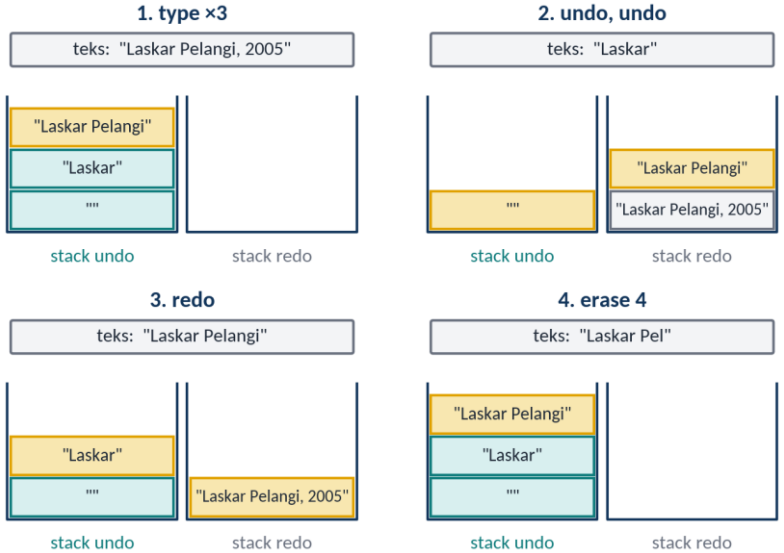
Undo: *push* keadaan kini ke *redo*, *pop* dari *undo*. **Redo:** *push* ke *undo*, *pop* dari *redo*.

Kode 5.5 Dari `06_undo_redo.cpp`: *undo* dan *redo* memindahkan satu keadaan tersimpan di antara kedua stack.

```
    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");
    }
```

Kode 5.5 *menyimpan salinan utuh teksnya (memento); penyunting sungguhan menyimpan operasinya, dengan disiplin dua stack yang sama. Tombol Back dan Forward peramban adalah dua stack yang sama. Buku: Subbab 5.7.*

Undo dan Redo, Digambarkan



Gambar 5.6 Stack undo dan redo pada empat saat selama program berjalan. Suntingan baru (panel 4) menambah stack undo dan mengosongkan stack redo.

***Gambar 5.6**: setelah `erase 4`, sebuah suntingan baru, stack redo dikosongkan — sehingga `redo` berikutnya melaporkan "nothing to redo". Mempertahankannya akan menghidupkan kembali keadaan dari riwayat yang sudah ditinggalkan. Buku: Subbab 5.7.*

Tiga Cara Menulis Jumlah yang Sama

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

***Tabel 5.2**: dalam postfiks operator muncul persis dalam urutan penerapannya, sehingga tidak diperlukan tanda kurung — itulah sebabnya mesin dengan stack dapat mengevaluasinya dalam satu lintasan (Hamblin, Bauer, Dijkstra, akhir 1950-an). Buku: Subbab 5.8.1.*

Presedensi dan Asosiativitas

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 di antara operator yang berbeda: `*` mengalahkan `+`.

Asosiativitas memutuskan di antara yang setara: `8-3-2` adalah $(8 - 3) - 2 = 3$, bukan 7; `2^3^2` adalah $2^9 = 512$, bukan $8^2 = 64$.

Dalam *shunting yard*, operan langsung lewat; `(` adalah sekat; `)` mem-*pop* sampai sekatnya; operator lebih dahulu mem-*pop* apa yang harus diterapkan sebelumnya, lalu menunggu di stack.

Tabel 5.3 *memuat aturan aritmetika sekolah biasa, dengan* `^` *untuk pangkat. Buku: Subbab 5.8.2 dan 5.8.3.*

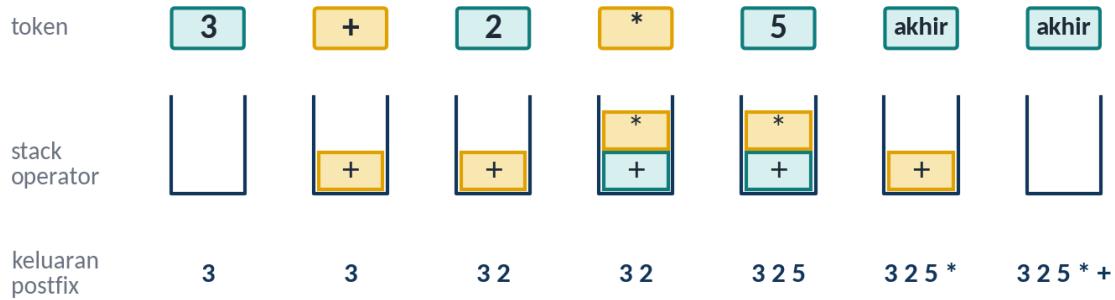
Algoritma Shunting-Yard Dijkstra

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
```

***Pseudokode 5.3**, Dijkstra 1961: halaman langsir kereta api. Buku: Subbab 5.8.3.*

Contoh Terkerjakan 2: `3+2*5`



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 * +`.

***Gambar 5.7**: ketika `*` bertemu `+`, prec(+) < prec(∗), sehingga tidak ada yang di-pop dan `*` menunggu di atas. Satu perbandingan itulah yang membuat perkalian dikerjakan lebih dahulu. Hasilnya: `3 2 5 * +`. Buku: Subbab 5.8.4.*

Contoh Terkerjakan 3: `a+b*c-d`

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 -`

***Tabel 5.4**: ketika `-` tiba, `*` di-pop (lebih tinggi) dan `+` juga (setara, asosiatif kiri). `a b c * + d -` persis $(a + (b \times c)) - d$. Untuk `^` yang asosiatif kanan, kasus setara **tidak** mem-pop. Buku: Subbab 5.8.5.*

Mengevaluasi Postfiks

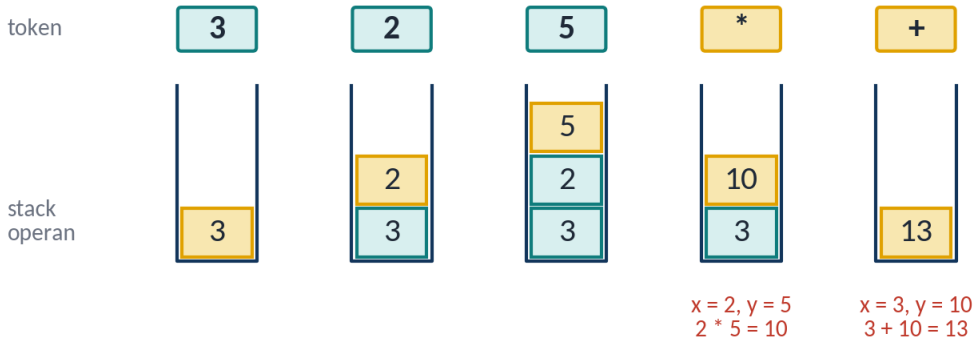
Pseudokode 5.4 Mengevaluasi ekspresi postfiks dengan satu stack operan.

```
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)
```

JEBAKAN OPERAN. Untuk `8 3 -`, *pop* pertama mengembalikan 3, yang kedua 8: hitung $x - y = 8 - 3 = 5$. Penjumlahan dan perkalian menyembunyikan *bug* ini; pengurangan dan pembagian membukanya.

Pseudokode 5.4*: push bilangan; operator mem-pop kedua operannya, menerapkan dirinya, lalu mem-push hasilnya. Tepat satu bilangan harus tersisa. Buku: Subbab 5.9.*

Postfiks, Dievaluasi



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.

***Gambar 5.8**: `3 2 5 * +` — `*` mem-pop 5 dan 2 lalu mem-push 10; `+` mem-pop 10 dan 3 lalu mem-push 13. Buku: Subbab 5.9.*

Pengevaluasi Kalkulator

Kode 5.6 Dari `07_calculator.cpp`: stack operan; *pop* pertama adalah operan kanan.

```
    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 5.6 *setelah* `toPostfix` *— konversi dan evaluasi masing-masing $O(n)$ untuk n token. Python memerlukan satu baris tambahan:* `//` *miliknya membulatkan ke arah minus tak hingga, sedangkan C++ dan Java memotong ke arah nol. Buku: Subbab 5.9.1.*

Kalkulator Bekerja

```
(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
```

Tanda kurung dan bilangan multidigit: $10 \cdot 6/5 = 12$. Asosiativitas kanan: 512 lawan 64.

Asosiativitas kiri dengan operan dalam urutan yang benar: `100/10/5` bernilai 2, `8-3-2` bernilai 3.

Kurung yang tidak berpasangan tertangkap saat konversi; operan yang kurang dan pembagian dengan nol saat evaluasi.

Jejak untuk `3+2*5` dan `a+b*c-d` cocok dengan gambarnya token demi token: 13 dan 9. Buku: Subbab 5.9.1.

Call Stack Fungsi

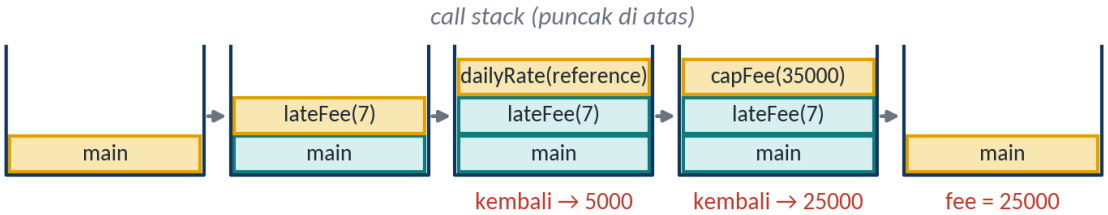
Setiap pemanggilan mem-*push* sebuah **stack frame**: alamat kembali, parameter, variabel lokal, register yang disimpan. Setiap *return* mem-*pop*-nya.

Variabel lokal lenyap ketika fungsinya kembali: jangan pernah mengembalikan pointer atau referensi kepadanya.

```
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
```

*`08_call_stack.cpp` mem-push nama saat masuk dan mem-pop-nya dalam destruktor pada setiap return. Rekursi adalah frame dari fungsi yang sama yang bertumpuk; rekursi yang lepas kendali adalah **stack overflow** (bawaan 1 MB di Windows, umumnya 8 MB di Linux). Buku: Subbab 5.10.*

Call Stack, Digambarkan



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.

***Gambar 5.9**: `dailyRate` dipanggil di tengah perhitungan biaya keterlambatan, mengembalikan 5000, dan di-pop sebelum `capFee` sempat dipanggil; biaya $7 \times 5000 = 35.000$ dibatasi Rp25.000. Buku: Subbab 5.10.2.*

ADT Queue

DEFINISI. *Queue* menyisipkan di *belakang* (*rear*) dan menghapus di *depan* (*front*); elemen yang dihapus selalu yang paling lama menunggu: **pertama masuk, pertama keluar**.

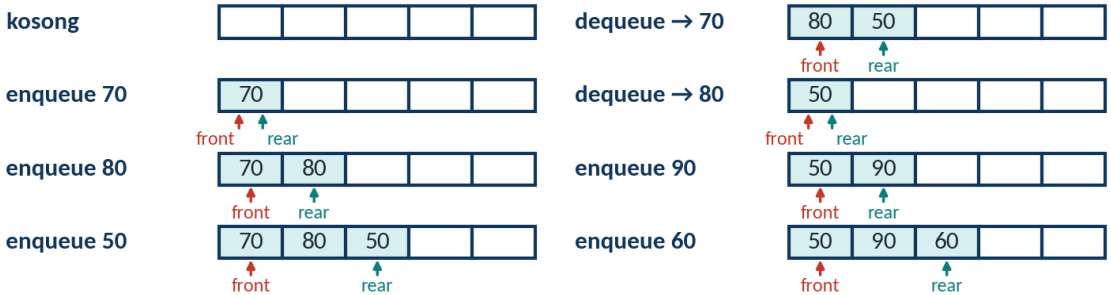
Antrean cetak, penjadwalan proses *round-robin*, *buffer* router, *buffer* papan ketik, penelusuran melebar (Bab 13).

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	—	$O(1)$

***Tabel 5.5**: `enqueue` juga disebut add, offer, push, atau put; `dequeue` juga remove, poll, pop, atau get. Buku: Subbab 5.11 dan 5.11.1.*

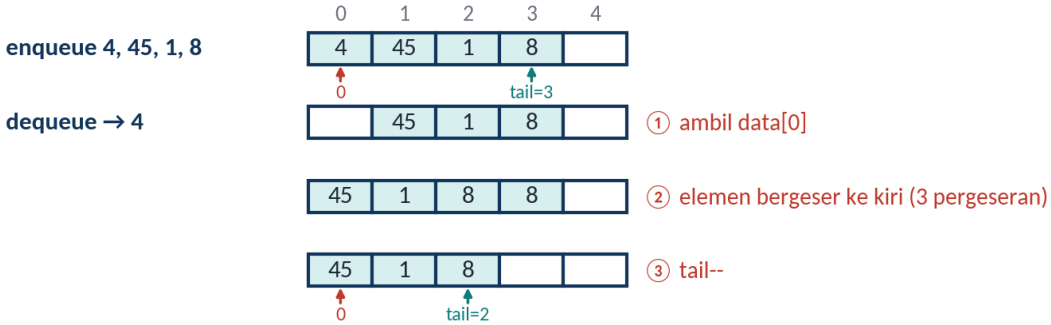
Queue Melalui Tujuh Operasi



Gambar 5.10 Sebuah queue melalui tujuh operasi, digambar dengan bagian depan di kiri. Elemen keluar persis dalam urutan kedatangannya: 70, lalu 80.

***Gambar 5.10**: 50 adalah kedatangan terakhir saat dequeue pertama, namun 70 yang keluar, lalu 80. Gambarnya menyembunyikan penyimpanan; queue berubah di **kedua** ujung, sehingga larik menjadi lebih sulit. Buku: Subbab 5.11.1.*

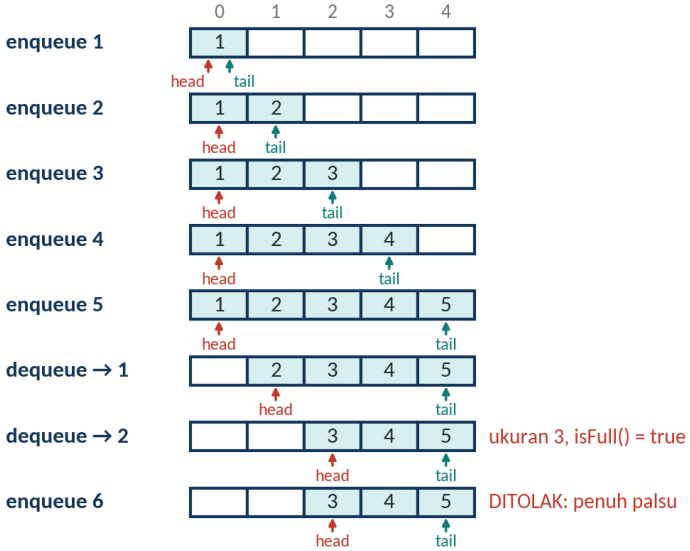
Percobaan Pertama: Geser pada Setiap *Dequeue*



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*.

***Gambar 5.11**: benar dan tidak pernah menyia-nyiakan slot, tetapi setiap dequeue memindahkan $n - 1$ elemen: $O(n)$. Lupa `tail--` menggandakan elemen terakhir (`45 1 8 8`). Buku: Subbab 5.12.1.*

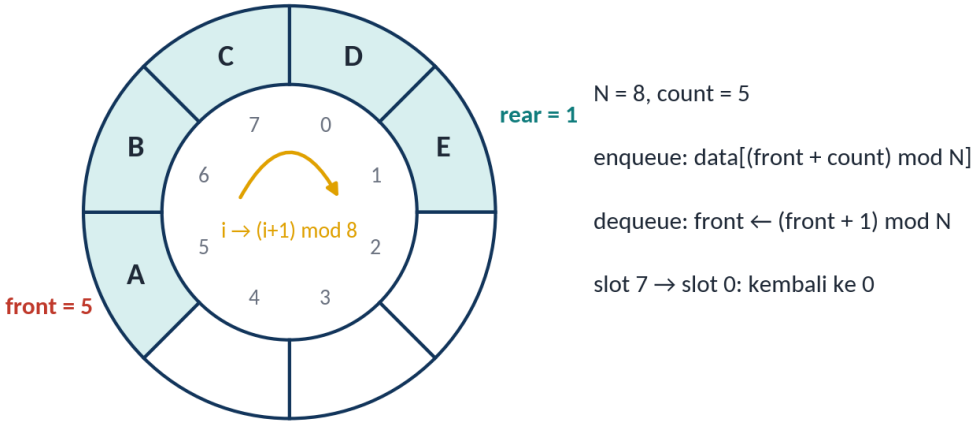
Percobaan Kedua: Queue yang Bergeser



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*.

***Gambar 5.12**: kedua indeks hanya bergerak ke kanan, sehingga setiap slot di kiri `head` hilang. `09_linear_queue.cpp` berakhir dengan "size 3 of 5, but isFull() says true": **penuh palsu**. Buku: Subbab 5.12.2.*

Queue Sirkular



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.

$$i \leftarrow (i + 1) \bmod N. \tag{5.6}$$

***Gambar 5.13**: tekuk larik menjadi cincin. Persamaan (5.6) membawa $7 \to (7 + 1) \bmod 8 = 0$: indeks tidak pernah keluar dari larik, dan tidak diperlukan `if`. Buku: Subbab 5.13 dan 5.13.1.*

Front Ditambah Pencacah

$$\text{free} = (\text{front} + \text{count}) \bmod N, \tag{5.7}$$

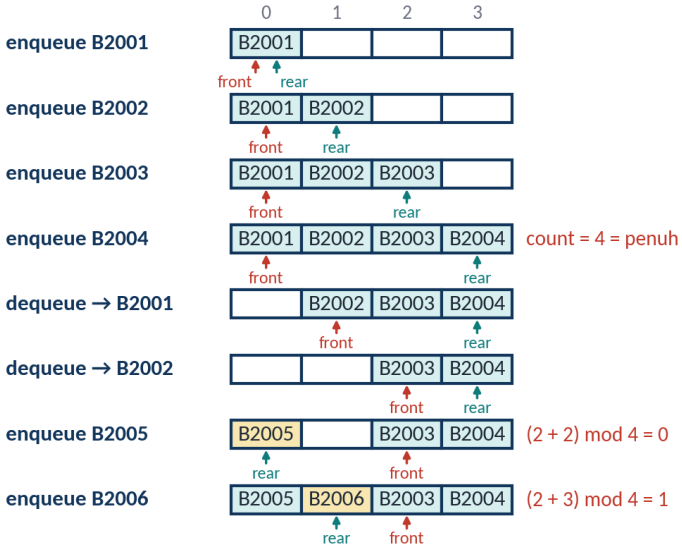
Pseudokode 5.5 `ENQUEUE` dan `DEQUEUE` pada larik sirkular berisi N slot dengan indeks `front` dan pencacah `count`.

```
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
```

*Persamaan (5.7): slot kosong pertama dihitung, bukan disimpan; elemen terakhir ada di $(front + count - 1)$ mod N. **Pseudokode 5.5**: kosong bila $count = 0$, penuh bila $count = N$. Buku: Subbab 5.13.2.*

Contoh Terkerjakan 4: Antrean Buku Baru



Gambar 5.14 Queue sirkular antrean buku baru dengan kapasitas 4. Dua enqueue terakhir melingkar ke slot 0 dan 1, sehingga belakang kini berada di kiri depan, dan queue tetap dalam urutan FIFO yang sempurna.

***Gambar 5.14**: setelah dua dequeue, `front` bernilai 2; B2005 masuk ke $(2 + 2)\ mod\ 4 = 0$ dan B2006 ke $(2 + 3)\ mod\ 4 = 1$. Secara fisik B2005 B2006 B2003 B2004; sebagai queue B2003 B2004 B2005 B2006. Buku: Subbab 5.13.2.*

Queue Sirkular dalam C++

Kode 5.7 Dari `10_circular_queue.cpp`: *enqueue* pada slot sesudah belakang yang dilingkarkan; *dequeue* memajukan `front` lalu melingkar.

```
    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;
    }
```

SISA BAGI NEGATIF. Di C++ dan Java, `-1 % 4` bernilai `-1`, bukan 3: `rear()` menambahkan `capacity_` sebelum mengambil `%`. `%` pada Python selalu tak negatif.

***Kode 5.7**: jalannya program cocok dengan gambar baris demi baris, dan mencetak "cataloging order: B2003 B2004 B2005 B2006". Buku: Subbab 5.13.3.*

Queue Sirkular dalam Python

Kode 5.8 Dari `10_circular_queue.py`: dua operasi yang sama; slot yang di-*dequeue* diisi `None`.

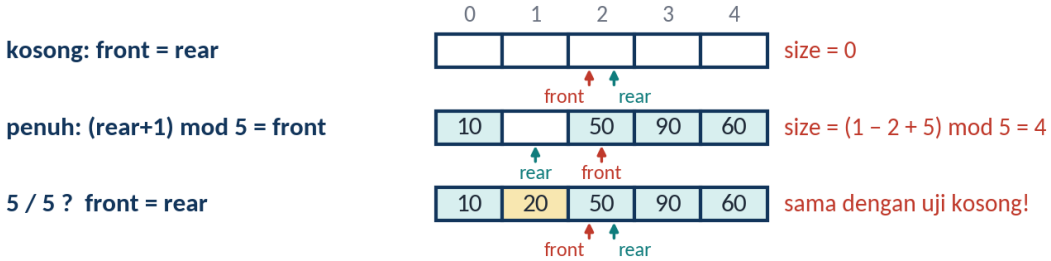
```
    def enqueue(self, value):
        if self.is_full():
            return False
        self._data[(self._front + self._count) % len(self._data)] = value   # just after the rear, wrapped
        self._count += 1
        return True

    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
```

```
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
```

***Kode 5.8**: C++, Java, dan Python mencetak empat belas baris yang sama. Buku: Subbab 5.13.3.*

Rancangan Kedua: Sisakan Satu Slot 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`.

***Gambar 5.15**: dengan hanya `front` dan `rear` (slot kosong berikutnya), queue kosong dan queue penuh sama-sama memiliki `front = rear`. Jangan pernah mengisi slot terakhir, maka kedua keadaan itu berbeda. Buku: Subbab 5.13.4.*

Pencacah atau Slot Kosong?

$$\text{size} = (\text{rear} - \text{front} + N) \bmod N. \tag{5.8}$$

$$\text{full} \Leftrightarrow (\text{rear} + 1) \bmod N = \text{front}, \qquad \text{empty} \Leftrightarrow \text{rear} = \text{front}. \tag{5.9}$$

Ukurannya adalah jarak dari `front` ke `rear` mengelilingi cincin, Persamaan (5.8); `+ N` melindungi dari operan kiri yang negatif.

Penuh bila satu *enqueue* lagi akan membuat `rear` menyusul `front`, Persamaan (5.9): N slot menampung paling banyak $N - 1$.

Harganya satu slot. Setelah `enqueue 60`, `rear` melingkar ke 0 dan ukurannya $(0 - 2 + 5) \bmod 5 = 3$. *Buku: Subbab 5.13.4.*

Dua Rancangan Berdampingan

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`

***Tabel 5.6**: dua thread masing-masing hanya menulis indeksnya sendiri — ring buffer di driver audio dan kartu jaringan. Buku: Subbab 5.13.4. Buku: Subbab 5.13.4.*

Deque

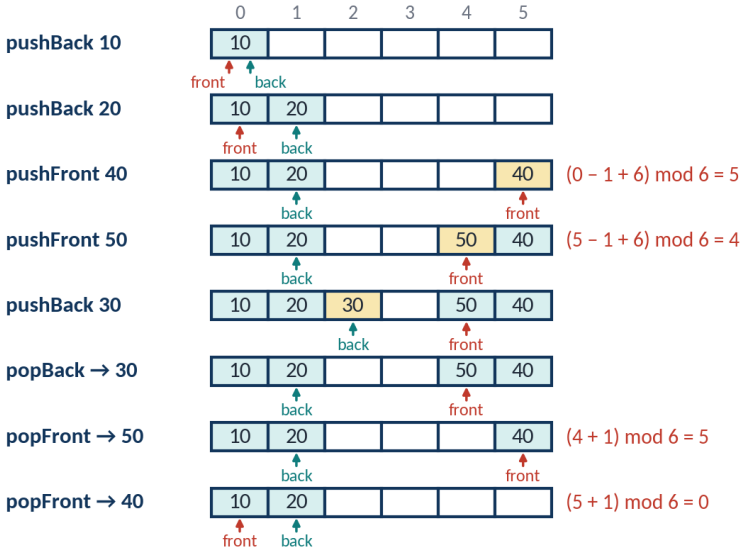
> **DEFINISI.** *Deque* (dibaca “dek”) menyediakan `pushFront`, `pushBack`, `popFront`, `popBack`, `peekFront`, dan `peekBack`. Dengan hanya ujung belakang ia menjadi stack; masuk di belakang dan keluar di depan, ia menjadi queue.

`pushFront` melangkahkan `front` **mundur**, dengan menambahkan N lebih dahulu agar sisa baginya tak negatif, Persamaan (5.10):

$$\text{front} \leftarrow (\text{front} - 1 + N) \bmod N. \tag{5.10}$$

`popBack` hanya mengurangi `count`, karena posisi belakang dihitung darinya. Buku: Subbab 5.14.

Deque di Atas Larik Sirkular



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.

***Gambar 5.16**: `pushFront` pertama memberi $(0 - 1 + 6) \bmod 6 = 5$, sehingga 40 mendarat di slot terakhir, yang secara logis berada di depan 10. Buku: Subbab 5.14.*

Deque dalam C++, dan Palindrom

Kode 5.9 Dari `12_array_deque.cpp`: `pushFront` mundur dengan pelingkaran; `popBack` membaca slot belakang yang lama.

```
    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_;
    }
    T popBack() {
        if (isEmpty()) throw underflow_error("deque is empty");
        --count_;
        return data_[(front_ + count_) % cap_];         // the old back slot
    }
```

```
"kasur rusak"                    palindrome
"Taat"                           palindrome
"racecar"                        palindrome
"stack"                          not a palindrome
"A man, a plan, a canal: Panama"  palindrome
```

***Kode 5.9**: uji palindrom mem-pop kedua ujung lalu membandingkannya, $O(n)$ — "kasur rusak" lolos. Deque juga menggerakkan penjadwal work-stealing dan maksimum jendela geser $O(n)$ (Soal Latihan 5.14). Buku: Subbab 5.14.*

Priority Queue: Sekilas

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

***Tabel 5.7**: unit gawat darurat melayani prioritas tertinggi lebih dahulu, kapan pun kedatangannya. Larik tak terurut atau terurut membuat salah satu operasi linear; heap biner Bab 14 membuat keduanya $O(lgn)$ — sekitar 20 langkah untuk sejuta elemen. Buku: Subbab 5.15.*

Stack dan Queue dalam Pustaka Standar

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`

***Tabel 5.8**: di C++ `pop()` mengembalikan `void`; `std::priority_queue` adalah max-heap. Di Java pakailah `ArrayDeque`; di Python jangan memakai `list.pop(0)` sebagai queue. Buku: Subbab 5.16.*

Tur Python

Kode 5.10 Dari `13_library.py`: `list` sebagai stack dan `collections.deque` sebagai queue.

```
    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)))
```

```
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
```

***Kode 5.10**: `append` dan `pop()` di ujung list bernilai $O(1)$ teramortisasi; `deque.popleft()` bernilai $O(1)$. `heapq` adalah min-heap, sehingga baris terbesar-dahulu mem-push —v. Buku: Subbab 5.16.3.*

Ringkasan Biaya

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 5.9**: yang terbaru dahulu — stack; yang terlama dahulu — queue, sebagai buffer sirkular, jangan yang bergeser; kedua ujung — deque; yang terpenting dahulu — priority queue. Dalam produksi, pakailah versi pustaka. Buku: Subbab 5.17.*

Profitina dalam Praktik

Data terstruktur harus seimbang: satu `}` yang hilang dalam blok JSON-LD menyembunyikannya dari mesin jawab — pemeriksaan **Pseudokode 5.2**.

Perayapan adalah queue: mengambil halaman yang ditemukan aras demi aras adalah penelusuran melebar.

Pekerjaan lambat masuk ke queue pekerjaan yang terbatas: FIFO itu adil, dan queue yang penuh mengatakannya alih-alih menerima pekerjaan yang tidak dapat diselesaikannya.

Pemindaian ulang terjadwal adalah *priority queue* berkunci waktu jatuh tempo; **umpan langsung memerlukan *ring buffer* terbatas**; **unggahan optimistis adalah *undo* satu tingkat**.

Jalur umpan balik — kirim, kurasi, dataset, latih dan ukur, rilis — adalah queue dari queue. Buku: Subbab 5.18.

Kesalahan yang Sering Terjadi

`data[top++] = x` dengan $top = -1$, atau mencampur dua konvensi `top`; *pop* tanpa memeriksa.

Menukar operan dalam evaluasi postfiks; memperlakukan `^` sebagai asosiatif kiri; melupakan sekat kurung.

Lupa `tail--`; mengirim queue yang bergeser; sisa bagi negatif di C++ dan Java.

`front == rear` **untuk penuh sekaligus kosong**; `list.pop(0)` atau `java.util.Stack`; mempertahankan stack *redo* setelah suntingan baru.

Mengembalikan alamat variabel lokal; menyalin kelas yang memiliki larik.

Empat belas kesalahan, masing-masing dengan perbaikannya, tercantum di buku. Buku: Subbab 5.19.

Ringkasan: Lima Hal yang Perlu Diingat

1. Stack bersifat LIFO: `push`, `pop`, `peek` dalam $O(1)$ dengan $top = -1$ berarti kosong (Persamaan (5.4)); larik yang berlipat dua bernilai $O(1)$ teramortisasi (Persamaan (5.5)).

2. Hanoi memerlukan $2^n - 1$ langkah (Persamaan (5.2)) dan dapat diselesaikan dengan tiga stack dan satu gelung.

3. Stack mencocokkan kurung, menjalankan *undo* dan *redo*, mengubah infiks ke postfiks lalu mengevaluasinya, dan menjalankan setiap pemanggilan fungsi.

4. Queue bersifat FIFO; queue sirkular melingkar dengan Persamaan (5.6) dan membedakan penuh dari kosong dengan pencacah atau slot kosong (Persamaan (5.9)).

5. Deque bekerja di kedua ujung; *priority queue* melayani yang terpenting dahulu; pustaka menyediakan semuanya.

Buku: Subbab 5.20.

Pertemuan Berikutnya

Bab 6 — Pointer, Referensi, dan Fungsi. Kisah pointer yang dimulai di Bab 1, dituntaskan: `new`, `delete`, referensi, dan cara fungsi menerima argumennya.

Stack bertaut hari ini adalah struktur pertama yang dibangun dari pointer; Bab 7 membangun seluruh list darinya.

Sebelum itu: kerjakan Soal Latihan 5.2, 5.5, 5.9, dan 5.14.

Soal Latihan 5.15 merancang loket pesanan Profitina Laundry dengan struktur-struktur hari ini. Buku: Subbab 5.21 dan 5.22.

Kuliah 6 — Pointer, Referensi, dan Fungsi

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 6

Dosen: Irfan Subakti — yifana@gmail.com

Hari ini kita menuntaskan kisah pointer yang dimulai Bab 1: alamat, memori dinamis, bug yang menyertainya, tiga cara fungsi menerima argumen, dan apa yang dilakukan Java serta Python sebagai gantinya. Buku: Bab 6, Subbab 6.1–6.18.

Cakupan Pertemuan Ini

1. **Apa itu pointer** — `&`, `*`, gambar kotak dan panah, penugasan, perbandingan, dan aritmetika.
2. **Pointer dan larik** — peluruhan, `a[i] ≡ *(a + i)`, *string* dan terminatornya, larik sebagai parameter, operator panah, dan rantai simpul pertama.
3. **Memori dinamis** — `new` dan `delete`, larik yang berlipat dua, dan lima *bug* klasik yang ditangkap AddressSanitizer dan Valgrind.
4. **`nullptr`, referensi, tiga cara mengirim argumen, `const`, mengembalikan dengan aman.**
5. ***Smart pointer*, referensi Java dan nama Python**, Profitina Laundry.

Segala sesuatu sejak Bab 7 bertumpu pada tiga gagasan: alamat, variabel yang menyimpannya, dan memori yang ada karena program memintanya ketika sedang berjalan. Buku: Subbab 6.1.

Dari Mana Kita Melanjutkan

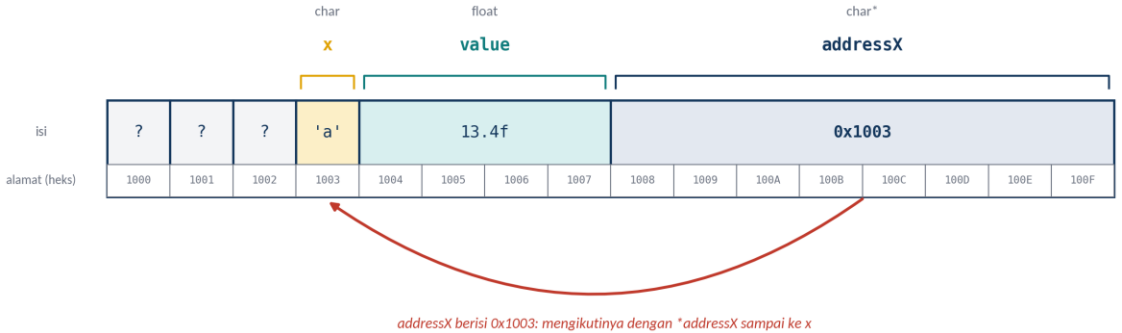
Bab 1 menggambar memori sebagai byte bernomor; Bab 2 dan 3 mencari dan mengurutkan larik; Bab 5 membangun stack dan queue di atas larik lalu menemukan titik lemahnya: **ukuran yang tetap**.

Kita sudah menulis `&catalog[i]`, `const Book& b`, dan mengirim larik yang dapat diubah fungsi. Hari ini semuanya dijelaskan.

Linked list, tree, dan graf menyimpan bagian-bagiannya **tersebar** di memori dan menyambungkannya dengan **alamat**.

Contoh berjalan: rekaman `Book` milik Pustaka Ceria (id, judul, pengarang, dan tahun), dengan Laskar Pelangi, Filosofi Kopi, Bumi Manusia, Negeri 5 Menara, dan Cantik Itu Luka. Buku: Subbab 6.1.

Memori sebagai Jalan dengan Rumah Bernomor



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`.

***Gambar 6.1**: setiap byte punya alamat, ditulis dalam heksadesimal; `char x` menyimpan `'a'` di `0x1003`, `float value` mengisi `0x1004`–`0x1007`, dan `addressX` menyimpan bilangan `0x1003`: alamat `x`. Buku: Subbab 6.2.1.*

Apa Itu Pointer

DEFINISI. *Pointer* adalah variabel yang nilainya alamat suatu lokasi di memori; ia *menunjuk* apa pun yang tinggal di sana. Pointer yang tidak menunjuk apa pun berisi *null*, ditulis `nullptr`.

Nomor rumah teman di secarik kertas: kertas itu bukan rumahnya, melainkan cara mencapainya — dan memfotokopinya tidak membangun rumah baru.

`&x` adalah operator **alamat**: "alamat `x`".

`*p` dalam ekspresi melakukan **dereferensi**: "benda yang tinggal di alamat dalam `p`". Membaca di kanan, menulis di kiri.

`*&x` adalah `x`, dan `&*p` adalah `p`: kedua operator saling membalik. Buku: Subbab 6.2.1–6.2.3.

Pointer Pertama

Kode 6.1 Dari `01_first_pointer.cpp`: pointer ke `float`, diikuti untuk membaca dan menulis.

```
    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';
```

```
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
```

***Kode 6.1**: `*address = 27.5f` mengubah `value` itu sendiri — pointer adalah jalan kedua menuju `float` yang sama. Setiap pointer berukuran 8 byte pada mesin 64-bit. Alamat berubah dari satu jalan ke jalan berikutnya (address space layout randomization); selisih antaralamatnya tidak. Buku: Subbab 6.2.2.*

Dua Simbol, Masing-masing Tiga Makna

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 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

***Tabel 6.1**: konteks yang menentukan. **Tabel 6.2**: baris terakhirnya benih setiap bug pointer — umur yang terpisah. Buku: Subbab 6.2.3 dan 6.2.4.*

Menyalin Alamat versus Menyalin Nilai

Kode 6.2 Dari `02_pointer_assign.cpp`: `*q = *p` menyalin nilai antara dua kotak; `q = p` menyalin alamat.

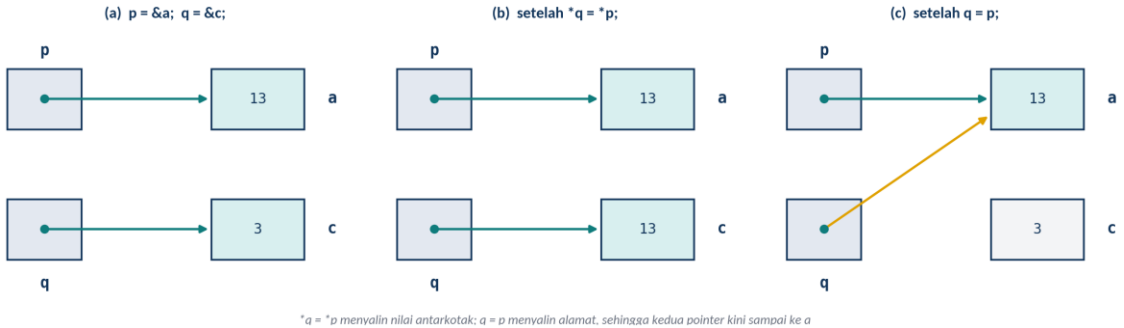
```
    // 3. Dua pointer, dua kotak: *q = *p menyalin nilai antarkotak.
    int c = 3;
    int* q = &c;
    std::cout << "3. before *q = *p: *p = " << *p << ", *q = " << *q
              << ", p == q: " << (p == q) << '\n';
    *q = *p;
    std::cout << "   after  *q = *p: *p = " << *p << ", *q = " << *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';
```

```
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
```

***Kode 6.2**: setelah `*q = *p`, `c` bernilai 13 dan panahnya tidak bergerak; setelah `q = p`, kedua panah mencapai `a`, dan menulis `*q = 99` mengubah `a`. Buku: Subbab 6.3.1.*

Kotak dan Panah

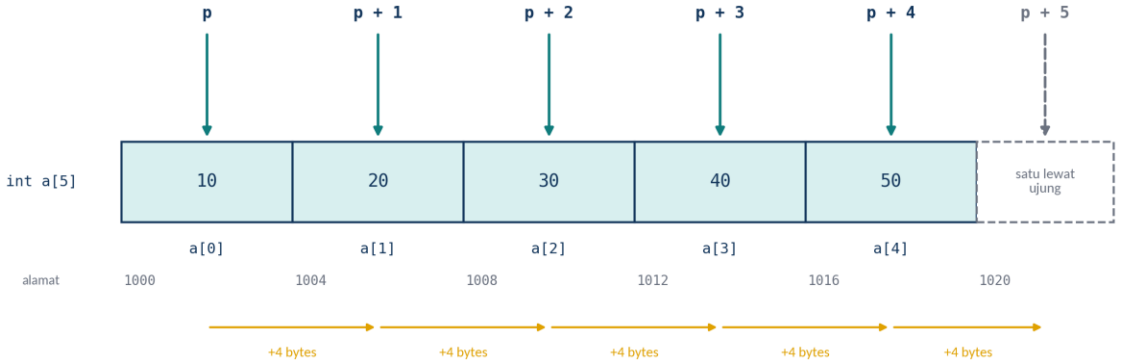


Gambar 6.2 Dua cara menugasi dengan pointer. `*q = *p` memindahkan nilai antarkotak; `q = p` memindahkan panah.

> **WAWASAN KUNCI.** Tanyakan pada setiap penugasan yang melibatkan pointer: *adakah `*` di sebelah kiri?* Dengan bintang, objek di ujung panah berubah. Tanpa bintang, sebuah panah berpindah dan tidak ada objek yang berubah.

***Gambar 6.2**: gambarlah setiap variabel sebagai kotak, gambar ulang hanya yang diubah sebuah pernyataan, dan gambarlah panah baru ketika pointer disalin. `p == q` bertanya "objek yang sama?", bukan "nilai yang sama?". Buku: Subbab 6.2.5, 6.3.1, dan 6.3.2.*

Aritmetika Pointer



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.

$$\text{alamat}(p + i) = \text{alamat}(p) + i \cdot \text{sizeof}(T) \tag{6.1}$$

*Persamaan (6.1): menambah 1 memindahkan satu elemen, bukan satu byte — rumus alamat Bab 1 dengan pointer sebagai alamat dasar. `p + 5` menunjuk satu lewat ujung: boleh dihitung dan dibandingkan, tidak pernah didereferensi (**Gambar 6.3**). Buku: Subbab 6.3.3.*

Contoh Terkerjakan 1: Langkah dan Jarak

$$q - p = \frac{\text{alamat}(q) - \text{alamat}(p)}{\text{sizeof}(T)} \tag{6.2}$$

```
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
```

Persamaan (6.2): mengurangkan dua pointer menghitung elemen: `last - first` bernilai 4, bukan 16 byte. Satu langkah `Book` adalah 104 byte: tiga `std::string` 32 byte, satu `int` 4 byte, diisi hingga kelipatan 8 — dan kompilator mengalikan dengan `sizeof(Book)` sehingga Anda tidak perlu tahu. Buku: Subbab 6.3.3.

Yang Boleh dan yang Tidak

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

PERILAKU TAK TERDEFINISI. Tidak ada ketentuan sama sekali tentang apa yang terjadi: *crash*, jawaban salah yang tampak masuk akal, atau perilaku sempurna sampai pemutakhiran kompilator berikutnya. Melanggar syarat mana pun dalam tabel adalah perilaku tak terdefinisi.

***Tabel 6.3**: operasi di luar tabel ditolak atau tak terdefinisi. Buku: Subbab 6.3.4.*

Nama Larik Menjadi Pointer

Dalam hampir setiap ekspresi, `a` meluruh menjadi `&a[0]`; pengecualiannya `sizeof(a)` dan `&a`. Standar mendefinisikan kurung siku dengan Persamaan (6.3):

$$a[i] \equiv *(a + i) \tag{6.3}$$

Kode 6.3 Dari `04_array_walk.cpp`: gelung satu-lewat-ujung, dan `2[a]`.

```
    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";
```

```
The value of a = 0 10 20 30 40
2[a] = 20 (the same as a[2] = 20)
```

***Kode 6.3**: $*(a+i) = *(i+a)$, sehingga `i[a]` sah — melihatnya sekali menghilangkan kesan gaib dari subskrip. Pasangan (pertama, satu lewat terakhir) adalah **rentang setengah terbuka** $[first, last)$: `v.begin()` dan `v.end()`. Buku: Subbab 6.4.1 dan 6.4.2.*

String adalah Larik `char`

Kode 6.4 `05_danaya.cpp`: menyusuri string C maju dan mundur dengan pointer; `overrun` mengulang kesalahan klasik.

```
    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;
    }
```

```
sizeof(s) = 7, strlen(s) = 6
Forward:  Danaya
p now points at s[6], the terminator
Backward: ayanaD
```

***Kode 6.4**: `"Danaya"` terdiri atas tujuh karakter, enam huruf dan `'\0'`. Berjalan lagi tanpa mengatur ulang `p` membaca melewati larik; dibangun dengan `-fsanitize=address` dan dijalankan sebagai `./danaya overrun`, ia melaporkan stack-buffer-overflow di baris 32. Tanpa sanitizer ia "berjalan". Buku: Subbab 6.4.3.*

Larik sebagai Parameter Fungsi

Larik yang dikirim ke fungsi tiba sebagai alamat elemen pertamanya: fungsi dapat mengubah larik pemanggil, dan ia **tidak tahu panjangnya**.

`int data[]` dalam daftar parameter ditulis ulang menjadi `int* data`; GCC memperingatkan bila Anda menanyakan `sizeof`-nya.

```
Fill the data...
  in main: sizeof(data) = 40 (10 ints)
  inside fill: sizeof(data) = 8 (the size of a pointer)
sum of all ten = 55
sum of data[2..4] = 12
std::array sum = 26, std::vector sum = 10
```

Tiga cara memberi panjang: pointer dan cacah, rentang setengah terbuka dua pointer (`sum(data + 2, data + 5)` bernilai $3 + 4 + 5 = 12$), atau kontainer yang dikirim sebagai referensi `const`. Pada larik 2-D `int g[][4]`, panjang baris harus ada dalam tipe. Buku: Subbab 6.4.4 dan 6.4.5.

Operator Panah, dan Rantai Pertama

> **DEFINISI.** Untuk pointer `p` ke sebuah struktur, `p->m` berarti persis `(*p).m`. (`*bp.title` akan berarti `*(bp.title)`: titik mengikat lebih kuat.)

Kode 6.5 `07_book_pointer.cpp`: operator panah pada larik `Book`, dan rantai simpul yang pertama.

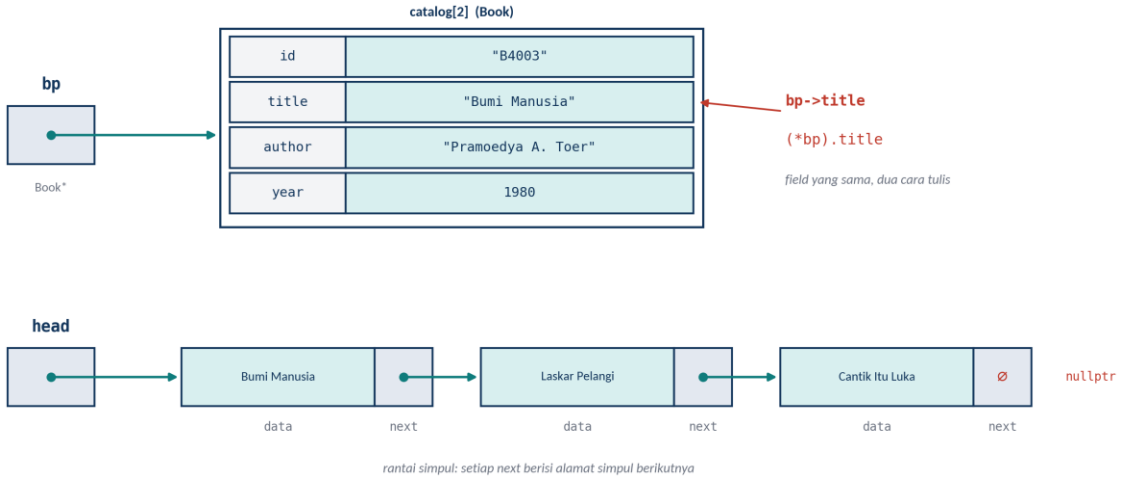
```
    // Tiga simpul, masing-masing menunjuk simpul berikutnya: rantai, bukan
    // array.
    Node third  {catalog[4], nullptr};
    Node second {catalog[0], &third};
    Node first  {catalog[2], &second};
    Node* head = &first;

    std::cout << "\nFollowing the chain from head:\n";
    for (Node* cur = head; cur != nullptr; cur = cur->next) {
        std::cout << "  " << cur->data.title << '\n';
```

```
Following the chain from head:
  Bumi Manusia
  Laskar Pelangi
  Cantik Itu Luka
head->next->next->data.title = Cantik Itu Luka
```

***Kode 6.5**: tiga variabel lokal terpisah, dideklarasikan terbalik, membentuk daftar terurut — urutannya tinggal sepenuhnya di pointer `next`. Mulai dari head, berhenti di null, maju dengan `next`. Buku: Subbab 6.5.*

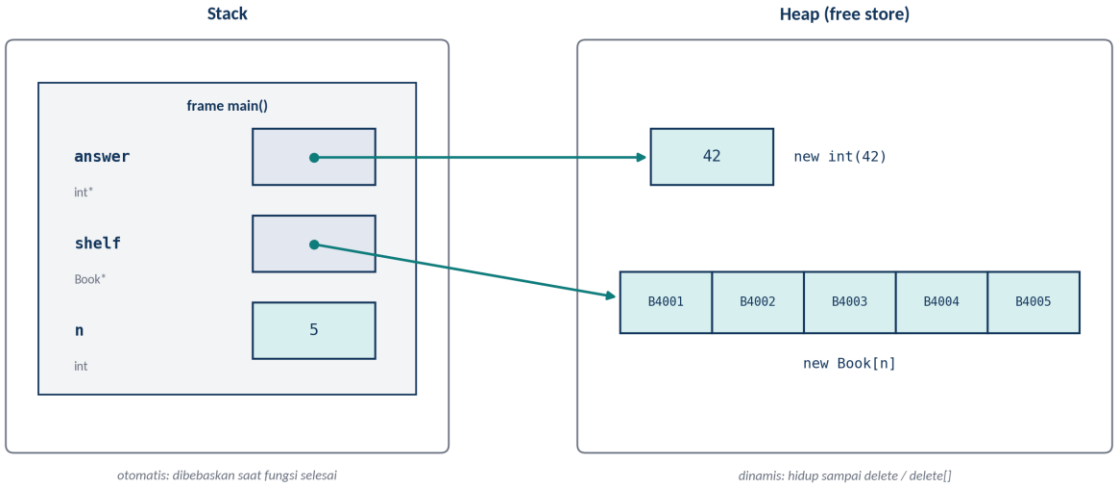
Simpul yang Ditautkan Pointer



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.

***Gambar 6.4**: bacalah `head->next->next->data.title` satu panah demi satu panah: Bumi Manusia, Laskar Pelangi, lalu Cantik Itu Luka. Ubah satu pointer dan urutannya berubah tanpa memindahkan satu `Book` pun: seluruh gagasan Bab 7. Buku: Subbab 6.5.*

Dua Tempat Tinggal: *Stack* dan *Heap*



Gambar 6.5 Variabel otomatis tinggal di *frame stack* fungsinya; objek yang dibuat dengan `new` tinggal di *heap* dan hanya dapat dijangkau melalui pointer.

> **DEFINISI.** Objek yang *dialokasikan secara dinamis* dibuat oleh `new T` (atau `new T[n]`) ketika program berjalan dan hidup sampai `delete` (atau `delete[]`), fungsi mana pun yang membuatnya. Pointer yang dikembalikan `new` adalah satu-satunya jalan untuk mencapainya.

***Gambar 6.5**: variabel otomatis tinggal di frame fungsinya; objek heap hidup lebih lama. "Heap" ini tidak berhubungan dengan heap Bab 14. Buku: Subbab 6.6.1.*

`new`, `delete`, dan Larik yang Berlipat Dua

Kode 6.6 Dari `08_dynamic.cpp`: bila blok *heap* penuh, alokasikan dua kali kapasitasnya, salin, lalu kembalikan blok lama.

```
    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;
    }
```

```
  grow: capacity 32, copied 16
size 17, capacity 32, last 289, total copies 31 (< 2 * 17 = 34)
```

***Kode 6.6**: setiap `new` berpasangan dengan satu `delete`, setiap `new[]` dengan satu `delete[]`; `~IntArray()` membebaskan blok terakhir — benih smart pointer. Setelah `delete answer;` program menulis `answer = nullptr;`. Bila `new` gagal ia melempar `std::bad_alloc`. Buku: Subbab 6.6.2 dan 6.6.4.*

Aturan Penggandaan

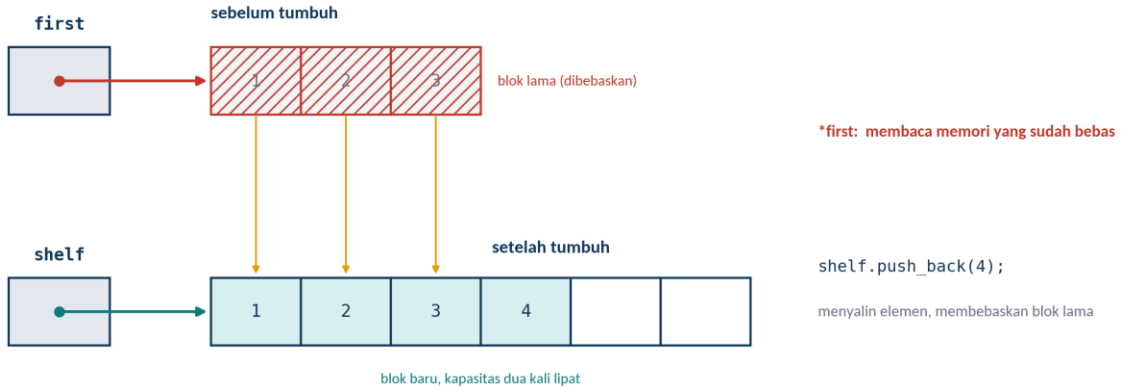
Pseudokode 6.1 `DYNAMIC-ARRAY-PUSH`: menambahkan `x`, setelah lebih dahulu menggandakan kapasitas bila larik penuh.

```
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
```

$$C(n) = 1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 < 2n \tag{6.4}$$

Pseudokode 6.1 *dan Persamaan (6.4): 17 kali push memakan 31 salinan, di bawah $2 \cdot 17 = 34$: kurang dari dua per push, $O(1)$ teramortisasi. Tumbuh tetap $+10$ akan memakan $\Theta(n^2)$. Buku: Subbab 6.6.3.*

Pertumbuhan Memindahkan Elemen



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.

***Gambar 6.6**: setelah satu langkah pertumbuhan blok lama dibebaskan, sehingga pointer yang diambil sebelum pertumbuhan kini menunjuk ke memori yang sudah bebas — bug `stale` yang segera kita tangkap. Buku: Subbab 6.6.3.*

Lima *Bug*, Satu Akar

Pointer menggantung: objeknya sudah tiada, alamatnya tersisa — *use after free*.

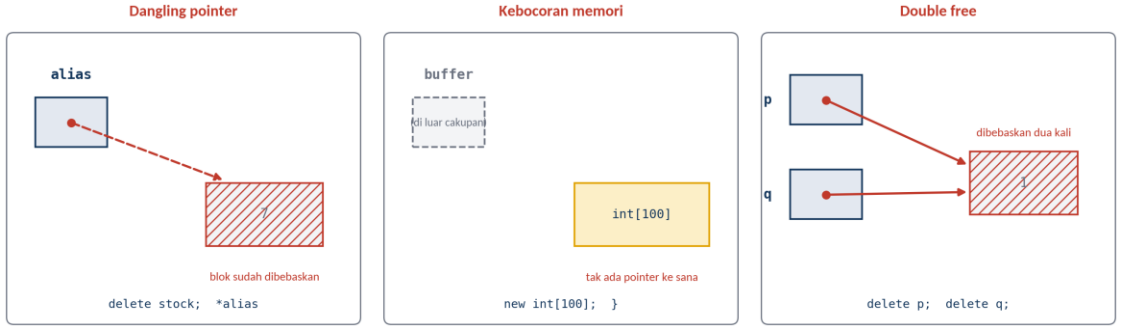
Kebocoran memori: objeknya tersisa, pointernya tidak. **Pembebasan ganda:** blok yang sama di-`delete` dua kali.

`delete` **yang tidak cocok:** `new[]` dibebaskan dengan `delete`. **Dereferensi null:** hasil "tidak ditemukan" yang dipercaya begitu saja.

Ditambah **akses di luar batas** dari `"Danaya"`.

Masing-masing adalah ketidaksesuaian antara umur pointer dan umur objeknya. Buku: Subbab 6.7.1.

Tiga *Bug* Umur, Digambarkan



Gambar 6.7 Tiga bug umur objek yang digambar sebagai kotak dan panah. Blok yang diarsir sudah dibebaskan.

***Gambar 6.7**: panah ke blok berarsir (sudah bebas), blok tanpa panah yang menuju kepadanya, dan dua panah ke blok yang dibebaskan dua kali. Gambarnya membuat setiap bug langsung tampak. Buku: Subbab 6.7.1.*

Bukan Sekadar Akademis: CWE Top 25 Tahun 2025

Tabel 6.4 Kelemahan keamanan memori dalam 2025 CWE Top 25 Most Dangerous Software Weaknesses, yang diterbitkan MITRE pada Desember 2025.

Peringkat 2025	Kelemahan (pengenal CWE)	Dalam bab ini
5	Out-of-bounds Write (CWE-787)	`overflow` di **Kode 6.7**
7	Use After Free (CWE-416)	`dangling`, `stale` di **Kode 6.7**
8	Out-of-bounds Read (CWE-125)	`overrun` di **Kode 6.4**
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 6.7**
16	Heap-based Buffer Overflow (CWE-122)	`overflow` di **Kode 6.7**

***Tabel 6.4**: enam dari 25 kelemahan paling berbahaya tahun 2025 persis bug-bug bab ini. Buku: Subbab 6.7.1.*

Bug-Bug Itu, dengan Sengaja

Kode 6.7 Dari `09_mem_bugs.cpp`: pemakaian setelah `delete`, dan pointer yang basi ketika vector tumbuh.

```
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';
}
```

```
usage: mem_bugs dangling|stale|leak|double|mismatch|overflow|null
no bug chosen; the correct versions:
stock is 7
```

***Kode 6.7**: setiap bug dipilih dengan sebuah kata di baris perintah. Dikompilasi biasa, sebagian besar tampak berjalan: `./mem_bugs dangling` mencetak `stock is 7`. Programnya salah, keluarannya benar. Buku: Subbab 6.7.2.*

AddressSanitizer Membaca Seperti Detektif

```
==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
```

Apa: *heap use after free*. **Akses mana:** pembacaan 4 byte. **Di mana:** baris 14. **Siapa yang membebaskannya:** `delete` di baris 13.

Laporan `stale`: tidak ada yang memanggil `delete` — bloknya dibebaskan *di dalam* `push_back`. Pointer, referensi, dan iterator ke dalam `std::vector` mati ketika ia tumbuh.

Bangunlah dengan `g++ -std=c++17 -g -fsanitize=address,undefined`: sekitar dua kali lebih lambat, dan setiap bug dilaporkan tepat di barisnya. Valgrind menjalankan program tanpa perubahan 20–50 kali lebih lambat dan memberi laporan kebocoran terbaik ("definitely lost"). Buku: Subbab 6.7.3 dan 6.7.4.

Alat yang Mana, di Komputer yang Mana

Tabel 6.5 Perbandingan AddressSanitizer dan Valgrind, per September 2026.

	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

WAWASAN KUNCI. Alat menemukan *bug* yang sudah *terjadi* dalam satu jalan; alat tidak dapat membuktikan program benar. Beri setiap objek *heap* satu pemilik yang jelas, utamakan kontainer dan *smart pointer*, dan biarkan destruktor yang membebaskan.

***Tabel 6.5**: C++26, rampung Maret 2026, menjadikan pembacaan variabel lokal yang belum diinisialisasi erroneous, bukan tak terdefinisi, dan menambah pustaka yang diperkeras; GCC 16.1 mengimplementasikan sebagian besar. Semua itu tidak memeriksa pointer mentah. Buku: Subbab 6.7.5.*

`nullptr`: Pointer ke Ketiadaan

Kode 6.8 Dari `10_nullptr.cpp`: pencarian yang mengembalikan alamat yang ditemukannya, atau `nullptr`.

```
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
}
```

```
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
```

***Kode 6.8**: `0` adalah `int`, sehingga `describe(0)` memilih overload `int`; `nullptr` hanya dapat dikonversi ke pointer. **Periksa sebelum mendereferensi**: `if (hit != nullptr)`. Parameter pointer berkata "mungkin null"; referensi berkata "tidak pernah null". Buku: Subbab 6.8.*

Referensi: Nama Kedua

DEFINISI. `T& r = x;` menjadikan `r` alias dari `x`: setiap pemakaian `r` adalah pemakaian `x`. Referensi harus diinisialisasi, tidak pernah dapat dipindahkan, dan tidak pernah null.

Kode 6.9 Dari `11_references.cpp`: menugasi referensi berarti menugasi objeknya; `r = b` tidak memindahkannya.

```
    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';
```

```
after r = 20: a = 20, &r == &a: true
after r = b; b = 7: a = 99, b = 7, r = 99
```

***Kode 6.9**: setelah `r = b; b = 7;`, `a` dan `r` tetap 99 — tidak ada panah untuk dipindah, hanya label kedua yang terpaku pada kotak yang sama. `for (int& x : stock)` mengubah kontainernya di tempat. Buku: Subbab 6.9.1.*

Referensi versus Pointer

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

***Tabel 6.6**: pakai referensi bila harus selalu ada objek dan objek itu tidak berganti; pakai pointer bila "tidak ada objek" mungkin terjadi atau pointernya kelak harus menunjuk ke tempat lain. Struktur bertaut memerlukan pointer untuk kedua alasan itu. Buku: Subbab 6.9.2.*

Tiga Cara Mengirim Argumen

Berdasarkan nilai `T x`: parameter diinisialisasi dengan **salinan** argumen.

Berdasarkan pointer `T* x`: parameter menyimpan salinan sebuah **alamat**, yang diberikan dengan `&`.

Berdasarkan referensi `T& x`: tidak ada objek baru — parameter adalah **nama lain bagi argumennya**.

```
a in passByValue() is 26
a in main() after passByValue() is 13
*a in passByPointer() is 26
a in main() after passByPointer() is 26
```

Deklarasi (prototipe) menjanjikan sebuah fungsi; parameter formal diinisialisasi dari parameter aktual; variabel lokal hanya hidup selama bloknya berjalan, sedangkan lokal `static` *mempertahankan nilainya antarpemanggilan. Buku: Subbab 6.10.1 dan 6.10.2.*

Contoh Terkerjakan 2: Kisah *Swap*

Kode 6.10 `13_swap.cpp`: menukar berdasarkan nilai (gagal), melalui pointer dan melalui referensi (keduanya berhasil), serta `std::swap`.

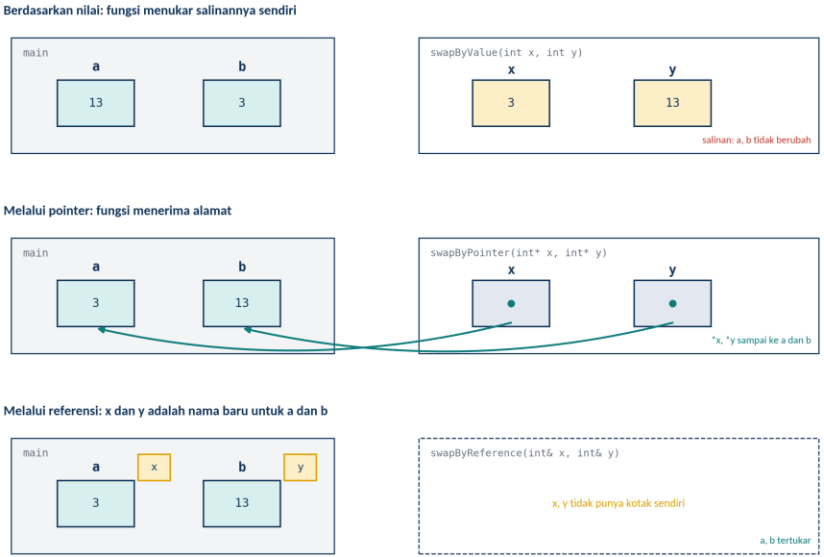
```
void swapByValue(int x, int y) {          // x, y adalah salinan
    int t = x;
    x = y;
    y = t;
    std::cout << "  inside swapByValue:     x = " << x << ", y = " << y << '\n';
}

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;
}
```

***Kode 6.10**: percobaan terbaik dalam seluruh topik ini. Versi pointer memeriksa `nullptr`; versi referensi di tempat pemanggilan tampak persis seperti versi berdasarkan nilai. Buku: Subbab 6.10.3.*

Kisah *Swap*, Dijejaki



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.

***Gambar 6.8**: berdasarkan nilai, salinannya ditukar lalu lenyap; berdasarkan pointer, $a = 3, b = 13$; berdasarkan referensi keduanya ditukar lagi,* 13 *dan* 3*;* `std::swap` *menjadikannya* 3 *dan* 13*. Fungsi yang dipanggil berdasarkan nilai tidak dapat merusak variabel pemanggilnya. Buku: Subbab 6.10.3.*

Biaya Pengiriman Berdasarkan Nilai, dan Sebuah Pilihan

$$\text{byte yang disalin} = k \cdot s \qquad \text{berdasarkan nilai,} \qquad 0 \qquad \text{melalui referensi atau pointer} \tag{6.5}$$

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)

*Persamaan (6.5): k pemanggilan menyalin k · s byte berdasarkan nilai, ditambah memori heap milik objek itu. **Tabel 6.7**: tipe bawaan kecil berdasarkan nilai, objek besar dengan `const T&`, perubahan dengan `T&`, objek opsional dengan `T*`. Buku: Subbab 6.10.4 dan 6.10.5.*

Ketepatan `const`

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

```
1000 calls by value:     1000 copies
1000 calls by const ref: 0 copies
```

***Tabel 6.8**: bacalah dari kanan ke kiri mulai dari namanya. Fungsi anggota const, `int age(int now) const`, boleh dipanggil pada `const Book`. Penghitung salinan membenarkan Persamaan (6.5): 1.000 salinan berdasarkan nilai, nol dengan `const&`. Langgar janjinya dan kompilator menghentikan Anda. Buku: Subbab 6.11.*

Mengembalikan Pointer dan Referensi dengan Aman

Kode 6.11 Dari `15_find_book.cpp`: mengembalikan pointer ke dalam larik pemanggil, atau `nullptr`.

```
// 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;
}
```

Aman: pointer atau referensi ke penyimpanan milik pemanggil; nilai baru (*return value optimization*); `std::unique_ptr` untuk objek *heap* baru.

Jangan pernah: alamat, atau referensi ke, variabel lokal — GCC 13 memperingatkan dan bahkan mengembalikan null.

Kode 6.11 *adalah pola setiap pencarian pada struktur bertaut: hasilnya pointer ke sebuah simpul, atau null. Buku: Subbab 6.12.*

Smart Pointer: Kepemilikan dalam Tipe

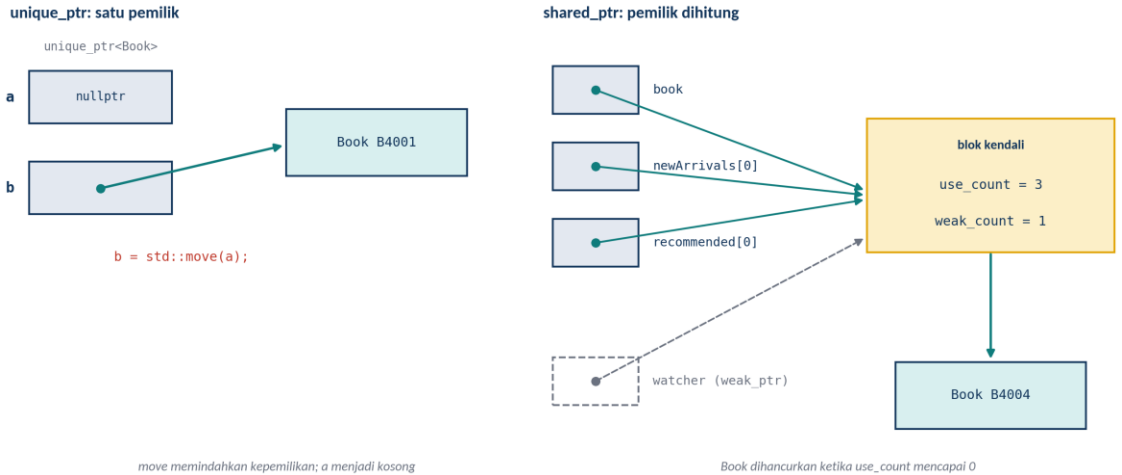
RAII: ikatkan setiap sumber daya pada objek yang destruktornya melepaskannya, baik pada keluar normal, `return` dini, maupun eksepsi.

`std::unique_ptr<T>`: tepat satu pemilik; tidak dapat disalin, hanya **dipindahkan** dengan `std::move`; tanpa biaya tambahan dibanding pointer mentah.

```
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
```

`withdraw(std::move(b))` mengambil kepemilikan, sehingga `[destroy B4001]` tercetak di dalam langkah 3. Vector berisi `unique_ptr` memiliki satu rak; `get()` memberi pengamat yang bukan pemilik; `make_unique<int[]>` memakai `delete[]`. Tidak ada `delete` di mana pun. Buku: Subbab 6.13.1 dan 6.13.2.

Pemilik Bersama, dan Sebuah Siklus



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.

***Gambar 6.9**: `shared_ptr` menghitung pemiliknya di blok kendali — 1, 3, 2, 1, lalu 0 dan `[destroy B4004]`; `weak_ptr` mengamati tanpa dihitung. Dua anggota yang saling memegang dengan `shared_ptr` bocor; menautkannya dengan `weak_ptr` memutus siklusnya. Pemilik menunjuk ke bawah, taut-balik menunjuk ke atas. Buku: Subbab 6.13.3.*

Pointer yang Mana, Kapan

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

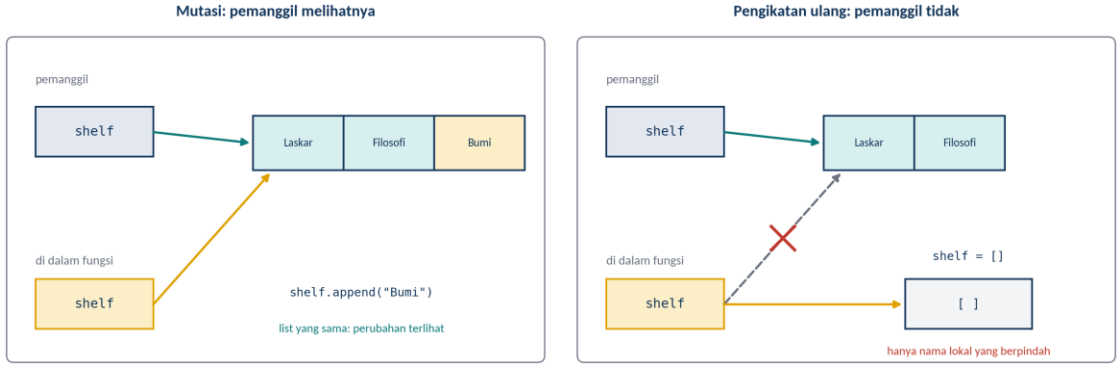
WAWASAN KUNCI. Bab 7 sampai 13 masih menulis `new` dan `delete` mentah, karena studi pertama struktur data adalah tempat Anda belajar apa yang dikerjakan *smart pointer* untuk Anda. Dalam produksi, simpul list akan memegang penerusnya dalam `unique_ptr`.

***Tabel 6.9**: pointer mentah tetap baik sebagai pengamat yang bukan pemilik; yang dihindari kode modern adalah pointer mentah yang memiliki. Buku: Subbab 6.13.3.*

Referensi Java dan Nama Python

Java: tipe primitif menyimpan nilai; variabel kelas dan larik menyimpan **referensi**; setiap argumen **dikirim berdasarkan nilai** — untuk objek, nilainya adalah referensi.

Python: setiap variabel adalah **nama yang terikat pada objek**; pemanggilan mengikat parameter pada objek pemanggil — *call by sharing*.



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.

***Gambar 6.10**: pihak yang dipanggil dapat **memutasi** objek bersama, dan pemanggil melihatnya; ia dapat **mengikat ulang** namanya sendiri, dan pemanggil tidak melihatnya. Buku: Subbab 6.14.1 dan 6.14.2.*

Mutasi versus Pengikatan Ulang dalam Python

Kode 6.12 Dari `refs_demo.py`: mutasi, pengikatan ulang, `int` yang imutabel, dan argumen bawaan yang mutabel.

```
# 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)
```

```
shelf after add_book:      ['Laskar Pelangi', 'Filosofi Kopi', 'Bumi Manusia']
shelf after replace_shelf: ['Laskar Pelangi', 'Filosofi Kopi', 'Bumi Manusia']
a is b: False, a == b: True
add_tag('novel')  -> ['novel']
```

***Kode 6.12**: `year += 1` membuat `int` baru dan mengikat ulang nama lokal. Nilai bawaan `tags=[]` dibuat sekali, sehingga pemanggilan kedua mengembalikan `['novel', 'sejarah']`; pakailah `tags=None`. Java berperilaku sama: `==` pada objek membandingkan referensi, `equals` membandingkan isi. Buku: Subbab 6.14.2 dan 6.14.3.*

Kisah *Swap* dalam Python

Kode 6.13 Dari `swap_demo.py`: mengikat ulang nama lokal tidak menukar apa pun; memutasi dua list bersama menukarnya.

```python
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
```

```
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
```

Kode 6.13*: baik Java maupun Python tidak dapat menukar dua variabel pemanggil. Idiomnya adalah* `a, b = b, a` *di lingkup pemanggil sendiri; Java mengurutkan dengan* `swapInArray(a, i, j)`*. CPython membebaskan memori dengan penghitungan referensi ditambah pengumpul siklus; Java dengan garbage collector penelusur. Buku: Subbab 6.14.4 dan 6.14.5.*

Tiga Bahasa Berdampingan

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`

Tabel 6.10 (Lanjutan) *Pointer, referensi, dan pengiriman parameter dalam C++, Java, dan Python.*

Pertanyaan	C++	Java	Python
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

Tabel 6.10*: garbage collection menghapus pointer menggantung dan pembebasan ganda, itulah sebabnya Java dan Python aman memori; ia tidak menghapus kebocoran melalui referensi yang tersimpan tanpa sengaja. Buku: Subbab 6.14.6.*

Profitina dalam Praktik

Pemindaian adalah *pipeline* tahap-tahap yang semuanya menerima objek hasil *yang sama*: berbagi itu gratis, menyalin teks halaman di setiap tahap akan memakan Persamaan (6.5).

Berbagi membawa kewajiban: tahap yang hanya membaca tidak boleh memutasi; gerbang rilis berisi jauh lebih dari seribu uji memeriksa persis hal itu.

Satu objek mahal, banyak pemakai: model bahasa lokal dimuat sekali dan dibagi melalui referensi.

"Tidak ada model" ditangani secara eksplisit, tidak pernah didereferensi begitu saja: disiplin `nullptr` pada skala sebuah layanan.

Argumen bawaan yang mutabel adalah sumber klasik bug semacam ini pada layanan web Python: satu list dibagi oleh setiap permintaan. Buku: Subbab 6.16.

Profitina Laundry: Memindahkan Kepemilikan

Kode 6.14 Dari `18_laundry.cpp`: promosi memindahkan pointer pemilik antar-queue; pesanannya sendiri tidak pernah berpindah.

```
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;
}
```

```
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]
```

***Kode 6.14**: pengamat mentah `watched` masih melihat #102 setelah perpindahan. `quoteWithDiscount` bekerja pada salinan (Rp34650, harga tersimpan tetap Rp38500); `applyDiscount` menerima referensi dan mengubah harga sebenarnya. Tanpa `delete`, dan Valgrind melaporkan semua blok bebas. Buku: Subbab 6.16.*

Kesalahan yang Sering Terjadi

`q = p` **untuk** `*q = *p`; `int* p, q;` hanya mendeklarasikan satu pointer; pointer yang belum diinisialisasi; tanpa pemeriksaan null.

Berjalan melewati ujung; `sizeof` parameter larik; mengembalikan alamat variabel lokal.

Lupa `delete`, `delete` **dua kali,** `delete` **untuk** `new[]`; pointer ke dalam vector setelah ia tumbuh.

Mengira `r = b` **memindahkan referensi**; mengirim objek besar berdasarkan nilai; siklus `shared_ptr`.

`==` **Java pada *string***; mengira metode dapat mengikat ulang variabel pemanggil; nilai bawaan mutabel di Python.

Lima belas kesalahan, masing-masing dengan perbaikannya, tercantum di buku. Buku: Subbab 6.17.

Ringkasan: Lima Hal yang Perlu Diingat

1. Pointer menyimpan alamat: `&` mengambilnya, `*` mengikutinya; `p + i` bergerak i elemen (Persamaan (6.1)), dan `a[i]` adalah `*(a + i)` (Persamaan (6.3)).
2. `new` dan `delete` berpasangan tepat; larik yang berlipat dua tumbuh dalam $O(1)$ teramortisasi (Persamaan (6.4)); ASan dan Valgrind menangkap lima *bug* umur.
3. `nullptr` berarti "tidak ada objek"; referensi adalah alias yang tidak pernah null dan tidak pernah dipindahkan.
4. Berdasarkan nilai menyalin, berdasarkan pointer menyalin alamat, berdasarkan referensi membuat alias; argumen besar yang hanya dibaca adalah `const T&` (Persamaan (6.5)).
5. `unique_ptr` memiliki sendirian, `shared_ptr` menghitung, `weak_ptr` mengamati; Java dan Python dapat memutasi objek bersama tetapi tidak pernah mengikat ulang variabel pemanggil.

Mencapai elemen ke-i larik bernilai $\Theta(1)$; mencapai simpul ke-i sebuah rantai bernilai $\Theta(i)$ — dan menyisipkan ke dalam rantai cukup dua perubahan pointer. Buku: Subbab 6.15 dan 6.18.

Pertemuan Berikutnya

Bab 7 — Single Linked List. Simpul yang dibuat ketika program berjalan, disambung pointer `next`: penyisipan tanpa penggeseran.

Sisip dan hapus di depan, di belakang, dan di tengah; telusuri, cari, balikkan.

Sebelum itu: kerjakan Soal Latihan 6.3, 6.6, 6.10, dan 6.12, lalu jalankan satu *bug* dari `09_mem_bugs.cpp` di bawah AddressSanitizer.

Gelung `for (Node* cur = head; cur != nullptr; cur = cur->next)` hari ini adalah gelung yang paling sering Anda tulis pekan depan. Buku: Subbab 6.19 dan 6.20.

Kuliah 7 — Single Linked List

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 7

Dosen: Irfan Subakti — yifana@gmail.com

Hari ini kita membangun struktur data pertama yang seluruhnya terbuat dari pointer: simpul yang dibuat ketika program berjalan, disambung lewat medan `next`, dan dicapai dari satu pointer bernama `head`. Buku: Bab 7, Subbab 7.1–7.16.

Cakupan Pertemuan Ini

1. **Gagasannya** — perburuan harta karun, sejarahnya (IPL, COMIT, LISP), dan kosakatanya: simpul, `next`, `head`, NULL.
2. **Larik versus list** — akses acak melawan akses berurutan, biaya penggeseran, memori per elemen.
3. **List ber-*head*** — sisip dan hapus di kedua ujung, telusuri, kosongkan; cari, sisip di tengah, hapus berdasarkan kunci, balikkan.
4. **Varian** — simpul kepala *dummy*, pointer ke tautan, *head* dan *tail*, list melingkar, *stack* dan *queue* berantai.
5. **Membuktikan aman dan memilih dengan tepat** — Valgrind, AddressSanitizer, pustaka baku, pengukuran, Profitina.

Setiap operasi hari ini digambar sebelum dan sesudahnya, lalu dikodekan; urutan dua penugasan pointer akan berulang kali menentukan. Buku: Subbab 7.1.

Dari Mana Kita Melanjutkan

Bab 5 membangun *stack* dan *queue* di atas **larik**: cepat, tetapi dengan ukuran maksimum yang ditebak sebelum elemen pertama tiba.

Bab 6 memberi alat yang menghapus tebakan itu: `new` mengembalikan alamat memori baru, `delete` mengembalikannya, `->` mencapai medan, dan parameter referensi membuat fungsi dapat mengubah pointer milik pemanggil.

Hari ini alat-alat itu bersatu: setiap elemen tinggal dalam **simpul**-nya sendiri, dan setiap simpul mengingat di mana simpul berikutnya.

Harganya: list tidak dapat melompat ke elemen kelimanya, setiap simpul membayar sebuah pointer, dan satu baris yang salah tempat dapat menghilangkan seluruh rantai. Setiap masalah kita temui dengan sengaja, kita gambar, ukur, dan perbaiki. Buku: Subbab 7.1.

Perburuan Harta Karun

Petunjuk pertama di gerbang: “Lihat di bawah pohon mangga besar.” Di sana ada hadiah dan petunjuk berikutnya, dan seterusnya sampai “Perburuan selesai.”

Setiap perhentian memegang dua hal: hadiah (data) dan arah ke perhentian berikutnya (pointer).

Perhentian dapat berada di mana saja — tidak bersebelahan, tidak berurutan.

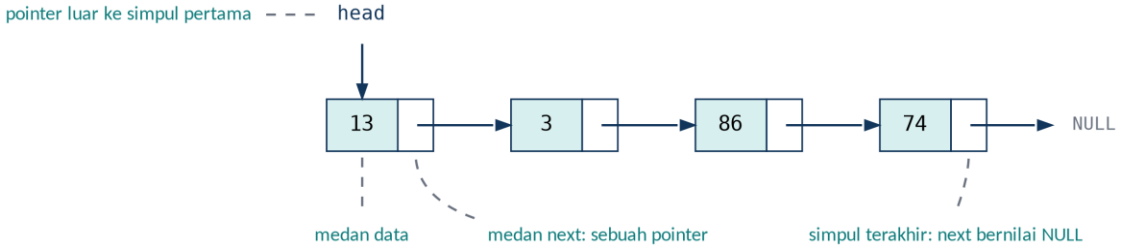
Rantai hanya dapat diikuti dari awal: hadiah kelima memerlukan empat petunjuk pertama.

Menambah atau membuang perhentian mengubah satu petunjuk; tidak ada hadiah yang dipindahkan.

Kartu yang dibagikan di gerbang adalah variabel pointer, `head`. Gagasannya berasal dari 1955–1956: IPL karya Newell, Shaw, dan Simon di RAND, lalu COMIT karya Yngve dan LISP karya McCarthy. Buku: Subbab 7.2.1 dan 7.2.2.

Kosakatanya

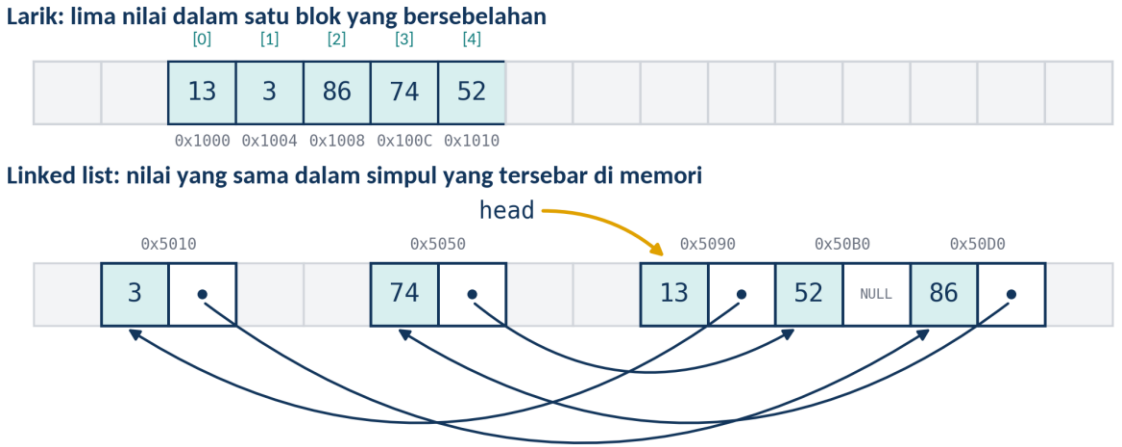
DEFINISI. Sebuah *linked list* adalah barisan *simpul*; setiap simpul menyimpan satu elemen (*data*-nya) dan sebuah pointer, *tautan* atau medan `next`, ke simpul berikutnya. List dicapai lewat pointer eksternal ke simpul pertamanya, `head`. Medan `next` simpul terakhir bernilai nol (`nullptr`, `null`, `None`); list kosong memiliki `head` yang nol.



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.

***Gambar 7.1**: data di kiri, `next` di kanan. Bertaut tunggal: satu tautan maju, tanpa jalan kembali (Bab 9 menambahkannya). Tidak melingkar: rantai berakhir di NULL, tanda berhenti setiap gelung hari ini. Buku: Subbab 7.2.3.*

Dua Cara Menyimpan Lima Bilangan



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.

***Gambar 7.2**: larik adalah satu blok dengan alamat berurutan; simpul-simpul list berada di mana pun `new` menaruhnya — 13 di `0x5090`, 3 di `0x5010`, 86 di `0x50D0` — dan hanya disambung oleh `next`. Buku: Subbab 7.3.1.*

Akses Acak Melawan Akses Berurutan

$$\text{alamat}(i) = b + i \cdot s \tag{7.1}$$

$$T_{\text{capai}}(k) = k \text{ langkah pointer}, \qquad 0 \le k \le n-1 \tag{7.2}$$

$$\frac{1}{n}\sum_{k=0}^{n-1} k = \frac{1}{n} \cdot \frac{(n-1)n}{2} = \frac{n-1}{2} \tag{7.3}$$

$$0 + 1 + 2 + \cdots + (n-1) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \tag{7.4}$$

Persamaan (7.1): `a[3]` *dalam* $O(1)$. *Persamaan (7.2), Persamaan (7.3): simpul ke-k memerlukan k langkah, rata-rata* $(n-1)/2$: $\Theta(n)$. *Persamaan (7.4): membangun larik dengan penyisipan depan menggeser* $\Theta(n^2)$ *elemen; list hanya perlu dua perubahan pointer per penyisipan. Buku: Subbab 7.3.1.*

Perbandingannya, Ciri demi Ciri

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

***Tabel 7.1**: ukuran list hanya dibatasi memori yang dapat diperoleh program. Melepaskan list justru lebih sulit: setiap simpul harus dihapus, satu per satu. Buku: Subbab 7.3.2.*

Memori yang Dimakan Setiap Elemen

$$M_{\text{larik}} = n\,d, \qquad M_{\text{list}} = n\,(d + p + a) \tag{7.5}$$

Persamaan (7.5): d bita data, $p = 8$ bita pointer, a bita alokator per simpul.

Simpul `int` diisi *padding* menjadi 16 bita; glibc membulatkannya ke *chunk* 32 bita — sekitar 8 kali `int`-nya.

Untuk catatan buku 200 bita, pointer 8 bita hanyalah pajak kecil.

> **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.

Cache line berukuran 64 bita: larik memakai setiap bita yang dimuatnya, list yang tersebar memuat line baru untuk hampir setiap simpul. Buku: Subbab 7.3.2 dan 7.3.3.

Tiga Simpul dengan Tangan

Kode 7.1 Dari `node_basics.cpp`: tiga simpul dibuat dengan `new`, ditautkan dengan tangan, dan ditelusuri dengan `aux`.

```
    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";
```

```
Three nodes linked by hand: 13 -> 3 -> 86 -> NULL
head->data             = 13
head->next->data       = 3
head->next->next->data = 86
```

***Kode 7.1**: `struct Node` bersifat self-referential: ia memegang alamat sebuah simpul, bukan simpulnya. `head` adalah pointer, bukan simpul. Gelung `for` atas `aux` adalah detak jantung setiap program list; `head->next->next->data` dibaca satu panah demi satu panah. Buku: Subbab 7.4.1–7.4.3.*

`init`, `isEmpty`, dan `frontInsertion`

```
void init(TNode *&head) { head = nullptr; }
bool isEmpty(const TNode *head) { return head == nullptr; }
```

Pseudokode 7.1 `FRONT-INSERTION`: sisipkan simpul pertama yang baru; dua penugasan pointer, dengan urutan ini.

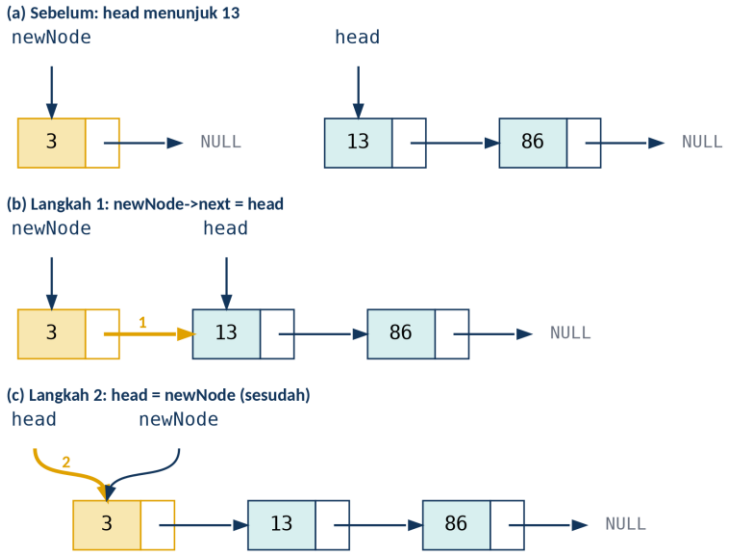
```
FRONT-INSERTION(L, x):
 1.  z ← a new node
 2.  z.data ← x
 3.  z.next ← L.head          // langkah 1: tautkan simpul baru ke simpul pertama lama
 4.  L.head ← z               // langkah 2: simpul baru menjadi simpul pertama
```

`head` yang tidak diinisialisasi tidak berarti "kosong", melainkan "entah di mana": selalu panggil `init`.

Pseudokode 7.1: tautkan simpul baru ke simpul pertama yang lama, *baru kemudian* pindahkan `head`. Untuk list kosong, dua baris yang sama tetap benar.

Biaya $O(1)$; n penyisipan depan berbiaya $\Theta(n)$, melawan $\Theta(n^2)$ untuk larik. Menyisipkan 13 lalu 3 menghasilkan 3, 13: penyisipan depan membalik urutan, persis cara kerja stack. Buku: Subbab 7.5.1 dan 7.5.2.

Penyisipan Depan, Langkah demi Langkah



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.

***Gambar 7.3**: pada panel (b) dua pointer menuju 13. Bila baris 4 dijalankan sebelum baris 3, simpul baru akan menunjuk dirinya sendiri dan 13 serta 86 hilang selamanya. Buku: Subbab 7.5.2.*

`backInsertion`: Simpul Terakhir yang Baru

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
```

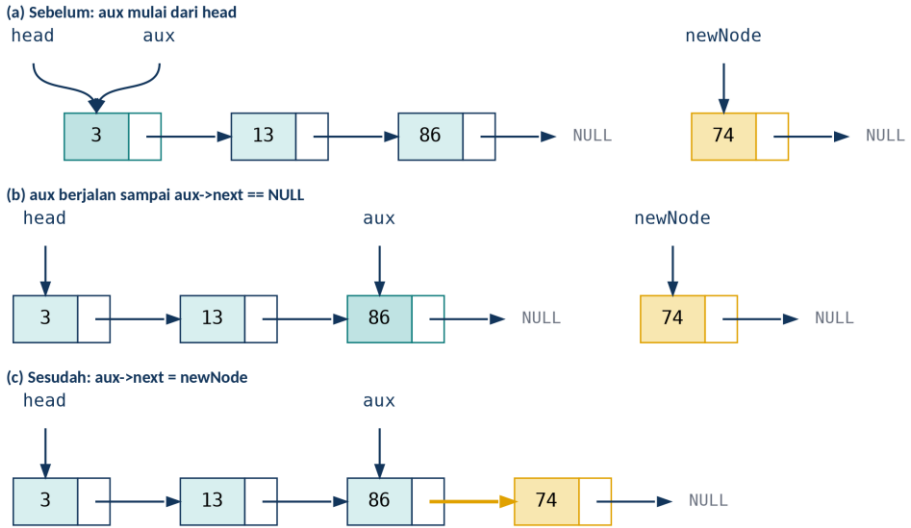
List ber-*head* tidak tahu di mana simpul terakhirnya, jadi `aux` berjalan ke sana.

Gelung menguji `aux.next ≠ NIL`: ia harus berakhir *berdiri pada* simpul terakhir.

List kosong perlu cabang sendiri: `aux->next` pada `aux` yang nol adalah perilaku tak terdefinisi.

Pseudokode 7.2*: $n - 1$ langkah, $\Theta(n)$; membangun n nilai dengan penyisipan belakang berbiaya $\Theta(n^2)$. Untuk mengubah tautan menuju sebuah posisi, Anda harus berdiri pada simpul sebelum-nya. Buku: Subbab 7.5.3.*

Penyisipan Belakang, Langkah demi Langkah



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.

***Gambar 7.4**: uji gelung melihat satu simpul ke depan. Setelah dua langkah `aux` berada di 86, yang `next`-nya NULL, jadi ia berhenti di sana, bukan satu langkah sesudahnya; baris 10 menempelkan 74. Buku: Subbab 7.5.3.*

`show`: Menyusuri List

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
```

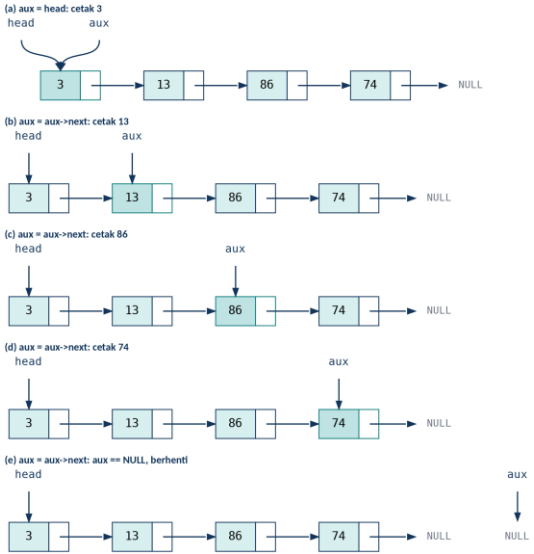
Di sini ujinya `aux ≠ NIL`: simpul terakhir pun harus dikunjungi.

Cara memilih: tanyakan pada simpul mana gelung harus *berakhir*. Sudah mengunjungi setiap simpul → uji `aux`. Berdiri pada simpul terakhir → uji `aux.next`.

Salah memilih adalah kesalahan selisih satu yang paling sering dalam kode list.

***Pseudokode 7.3**: $\Theta(n)$, dan itu optimal — mencetak n nilai memerlukan n langkah. Buku: Subbab 7.5.4.*

Penelusuran, Langkah demi Langkah



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.

> **ATURAN HEAD.** Jangan pernah memindahkan `head` untuk menyusuri list: setiap simpul yang dilewatinya hilang. Berjalanlah dengan pointer pembantu (`aux`, `cur`).

***Gambar 7.5**: `aux` mengunjungi setiap simpul dan berakhir di NULL; `head` tidak pernah bergerak. Buku: Subbab 7.5 dan 7.5.4.*

`frontDeletion`: Simpan, Pindahkan, Hapus

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
```

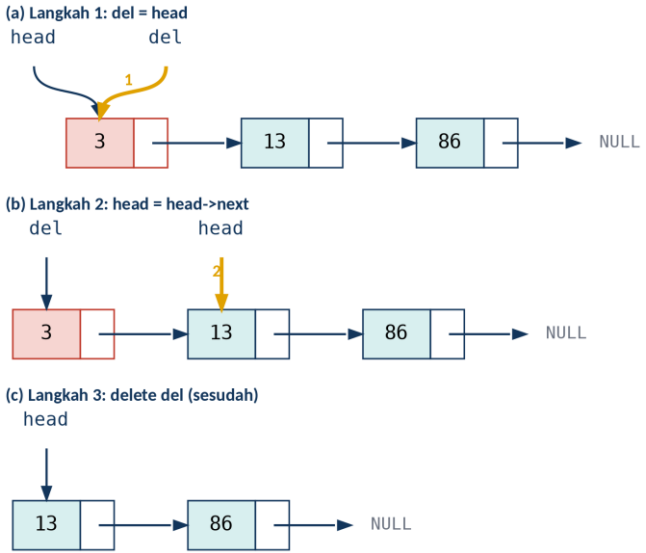
Menghapus dulu, lalu `head = head->next`, membaca simpul yang sudah dibebaskan.

Memindahkan `head` dulu tanpa menyimpan alamatnya membocorkan simpul itu.

Simpan, pindahkan, hapus — setiap penghapusan hari ini mengikutinya.

Pseudokode 7.4*: list satu simpul menjadi kosong lewat baris 4. Biaya* $O(1)$*. Buku: Subbab 7.5.5.*

Penghapusan Depan, Langkah demi Langkah



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.

***Gambar 7.6**: sesudah langkah 2 simpul merah bukan lagi bagian list, tetapi ia masih ada dan `del` masih menunjuknya; langkah 3 melepaskannya. Buku: Subbab 7.5.5.*

`backDeletion`: Membuang Simpul Terakhir

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
```

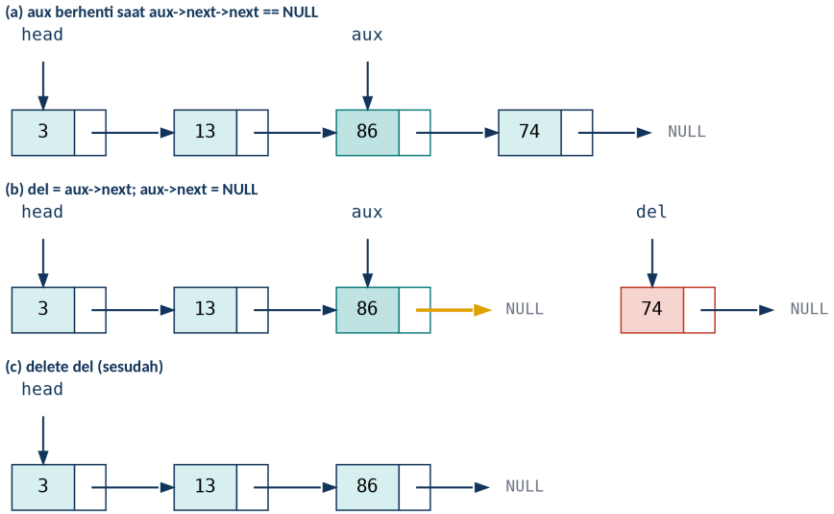
`aux` harus berhenti pada simpul **kedua dari belakang**, tempat `aux.next.next` bernilai NIL.

Satu simpul adalah kasus tersendiri: `aux.next.next` akan mendereferensi NULL, dan yang harus berubah adalah `head`.

Baris 11 paling sering terlupa: tanpanya, 86 masih menunjuk simpul yang sudah dibebaskan.

***Pseudokode 7.5**: $n - 2$ langkah, $\Theta(n)$ — dan pointer tail tidak akan memperbaikinya. Buku: Subbab 7.5.6.*

Penghapusan Belakang, Langkah demi Langkah



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.

***Gambar 7.7**: (a) `aux` berhenti di 86 karena simpul sesudahnya memiliki `next` NULL; (b) tautannya diputus dan 74 diingat dalam `del`; (c) simpul itu dilepaskan. Buku: Subbab 7.5.6.*

`clear` dan Biaya Sejauh Ini

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

***Tabel 7.2**: `clear` adalah penghapusan depan berulang, $\Theta(n)$, dan meninggalkan list kosong yang sah. Panggil sebelum pointer terakhir ke list lenyap, atau setiap simpul bocor. Buku: Subbab 7.5.7 dan 7.5.8.*

Program Lengkapnya: Penghapusan Belakang

Kode 7.2 Dari `sll_head.cpp`: `backDeletion`, dengan kasus satu simpul dan tautan yang diputus.

```
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 7.2**: baris 83 adalah baris 11 dari **Pseudokode 7.5**. Hasil jalannya menyisakan 13, 86. Buku: Subbab 7.5.8.*

List yang Sama dalam Java dan Python

Kode 7.3 Dari `SllHead.java`: penghapusan depan tanpa `delete`.

```java
    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()");
    }
```

Kode 7.4 Dari `sll_head.py`: hal yang sama dalam Python.

```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()")
```

Kode 7.3, ***Kode 7.4****: simpul pertama yang lama menjadi tak terjangkau dan garbage collector mengambilnya kembali;* `clear` *cukup* `head = null`*. Tidak ada pemakaian sesudah pembebasan, tetapi simpul masih dapat hilang (memindahkan* `head`*) atau tetap hidup karena referensi yang terlupa.* `__slots__` *menghemat memori dan menangkap salah ketik. Buku: Subbab 7.5.9.*

`search` dan `insertAfter`

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
```

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
```

***Pseudokode 7.6**: `aux ≠ NIL` didahulukan (short-circuit), dan yang dikembalikan adalah simpul-nya, siap untuk **Pseudokode 7.7**. $\Theta(n)$ pada kasus terburuk; tidak ada binary search $O(\lg n)$, karena mencapai tengah saja berbiaya $n/2$ langkah (Persamaan (7.2)). `insertAfter` bernilai $O(1)$ hanya karena pemanggil sudah memegang `p`. Buku: Subbab 7.6.1 dan 7.6.3.*

`length`, Secara Iteratif dan Rekursif

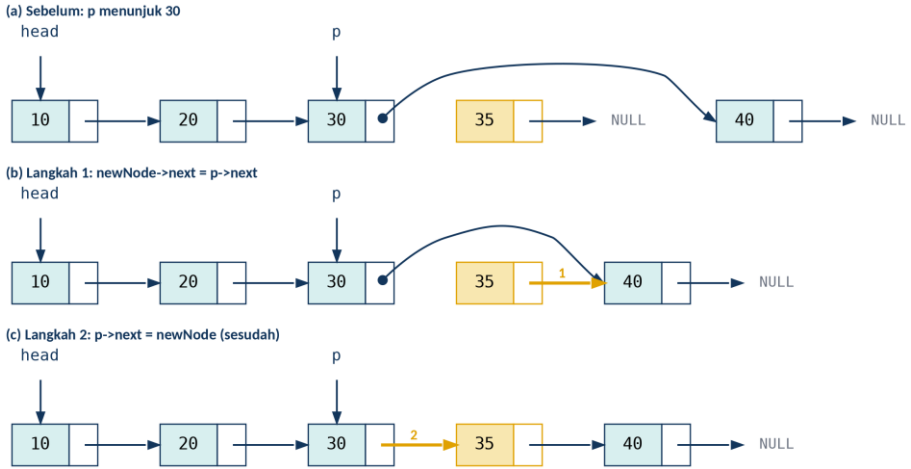
DEFINISI (rekursif). Sebuah list adalah *kosong* (pointernya NULL), atau sebuah *simpul pertama* yang diikuti sebuah *list* (list yang dimulai pada `next` simpul pertama).

$$L(p) = \begin{cases} 0 & \text{bila } p = \text{NULL} \\ 1 + L(p.\text{next}) & \text{selainnya} \end{cases} \tag{7.6}$$

```
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
}
```

Persamaan (7.6): $L(20) = 1 + (1 + (1 + 0)) = 3$. Kedua versi memakan waktu $\Theta(n)$, tetapi rekursi menahan $n + 1$ panggilan di call stack (8 MB di Linux): list berisi jutaan simpul dapat meluapkannya. Buku: Subbab 7.6.2.

Penyisipan di Tengah, Langkah demi Langkah



Gambar 7.8 `insertAfter(p, 35)` dengan `p` di 30: (a) sebelum, (b) simpul baru menunjuk penerus `p`, (c) `p` menunjuk simpul baru.

***Gambar 7.8**: pada panel (b) simpul baru sudah menunjuk 40 sementara 30 pun masih menunjuk 40; tidak ada yang rusak sampai langkah 2 menautkannya. Membalik urutan langkah membuat simpul baru menunjuk dirinya sendiri. Menyisipkan sebelum sebuah simpul memerlukan pointer yang tertinggal (Soal Latihan 7.9). Buku: Subbab 7.6.3.*

`insertSorted`: Menjaga List Tetap Terurut

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)
```

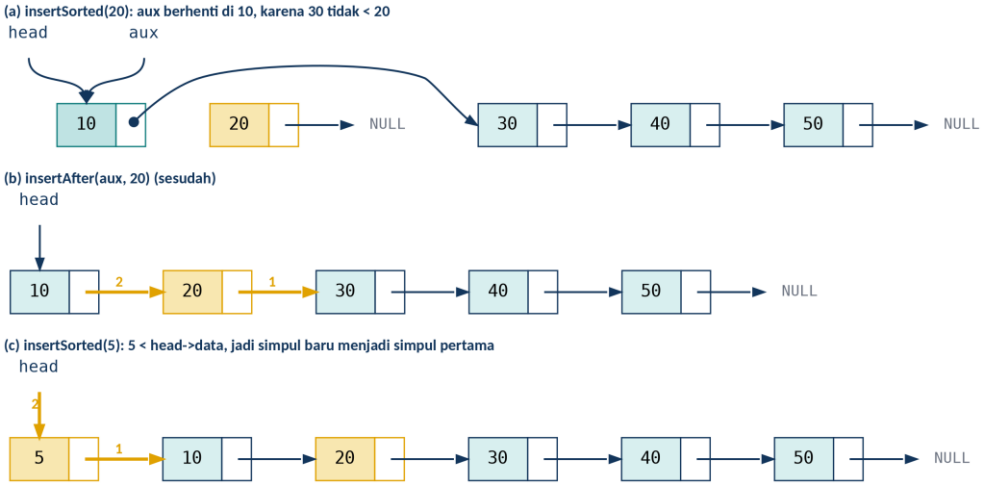
`aux` berjalan selama nilai simpul *berikutnya* lebih kecil; ia berhenti pada simpul terakhir yang lebih kecil.

Kasus khusus (baris 1): list kosong, atau nilai yang lebih kecil daripada nilai pertama, mengubah `head`.

`<` menaruh kunci yang sama sesudah kunci yang sudah ada: penyisipannya **stabil**.

***Pseudokode 7.8**: $\Theta(n)$ pada kasus terburuk. Menyisipkan n nilai dengan cara ini adalah insertion sort pada list — $\Theta(n^2)$, tetapi tanpa satu elemen pun digeser. Buku: Subbab 7.6.4.*

Penyisipan Terurut, Kedua Kasusnya



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.

***Gambar 7.9**: untuk 20, `aux` berhenti di 10 karena simpul berikutnya berisi 30; 5 lebih kecil daripada setiap nilai dan menjadi simpul pertama. Buku: Subbab 7.6.4.*

`deleteKey`: Penghapusan Berdasarkan Nilai

Pseudokode 7.9 `DELETE-KEY`: buang simpul pertama yang datanya sama dengan `key`, dengan pointer pengekor `prev`.

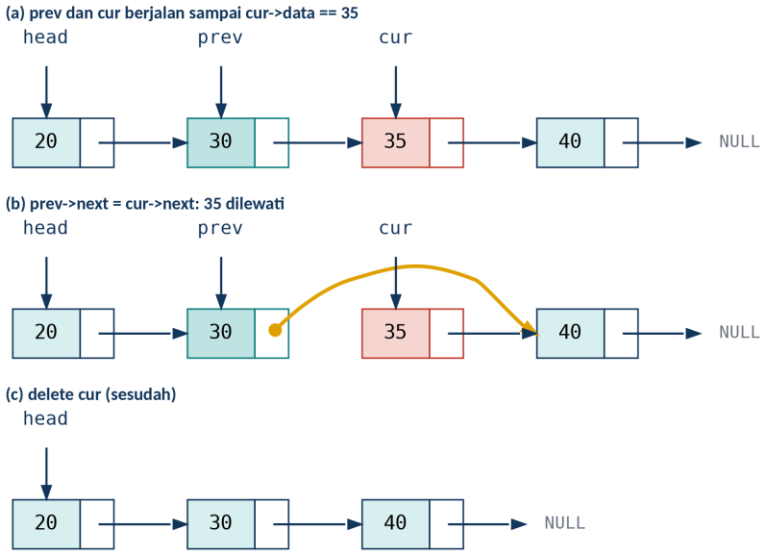
```
DELETE-KEY(L, key):
 1.  prev ← NIL
 2.  cur ← L.head
 3.  while cur ≠ NIL and cur.data ≠ key
 4.      prev ← cur
 5.      cur ← cur.next
 6.  if cur = NIL
 7.      return FALSE           // key tidak ada di list
 8.  if prev = NIL              // cur adalah simpul pertama
 9.      L.head ← cur.next
10.  else
11.      prev.next ← cur.next   // lompati cur
12.  free cur
13.  return TRUE
```

Simpul pertama: `prev` tetap NIL, `head` berpindah. **Tengah:** baris 11 melompati `cur`.

Simpul terakhir: `prev.next` menjadi NIL — penghapusan belakang tanpa kasus tambahan. **Tidak ada:** tidak ada yang berubah.

Pseudokode 7.9*: dua pointer berjalan bersama,* `prev` *satu simpul di belakang* `cur`*. List kosong adalah kasus "tidak ada" dengan gelung yang tidak pernah mulai.* $\Theta(n)$ *pada kasus terburuk. Buku: Subbab 7.6.5.*

Penghapusan Berdasarkan Kunci, Langkah demi Langkah



Gambar 7.10 `deleteKey(35)`: (a) `prev` dan `cur` berhenti dengan `cur` di 35, (b) `prev->next` melompatinya, (c) simpul itu dihapus.

Gambar 7.10: *`cur` pada 35, `prev` pada 30; `prev->next` melompati 35, sehingga penelusuran dari `head` tidak lagi menjumpainya; lalu simpul itu dilepaskan. Buku: Subbab 7.6.5.*

Membalik List Secara Iteratif

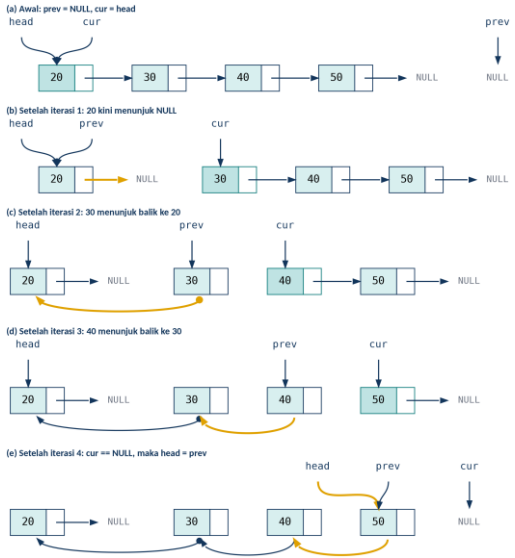
Pseudokode 7.10 `REVERSE-ITERATIVE`: balik list dalam satu lintasan dengan membalik setiap panah `next`.

```
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
```

> **INVARIAN GELUNG.** Pada awal setiap iterasi, `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; keduanya bersama-sama memuat setiap simpul tepat satu kali.

***Pseudokode 7.10**: baris 4 menyimpan satu-satunya catatan tentang sisa list sebelum baris 5 menimpanya. Ketika `cur` bernilai NIL, `prev` mengepalai seluruh list yang terbalik. Waktu $\Theta(n)$, memori $O(1)$. Buku: Subbab 7.6.6.*

Pembalikan Iteratif, Ditelusuri



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.

***Gambar 7.11**: sesudah setiap iterasi satu panah lagi menunjuk ke belakang. Tidak ada data yang dipindahkan. Buku: Subbab 7.6.6.*

Membalik List Secara Rekursif

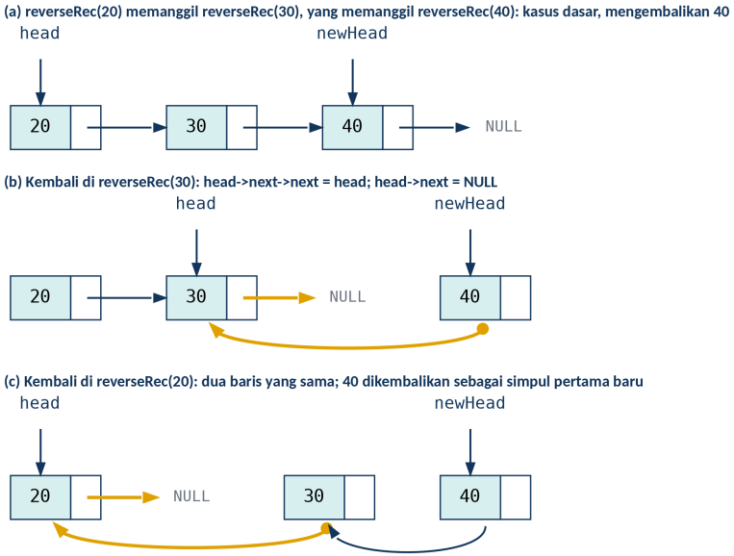
Pseudokode 7.11 `REVERSE-RECURSIVE`: balikkan sisa list, lalu gantungkan simpul pertama pada ujungnya.

```
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
```

$$R(n) = R(n-1) + c, \quad R(1) = c \quad \Rightarrow \quad R(n) = c\,n = \Theta(n) \tag{7.7}$$

***Pseudokode 7.11**: sesudah baris 3, `head.next` masih menunjuk simpul kedua yang lama, kini simpul terakhir dari sisa yang terbalik, sehingga `head.next.next ← head` tidak memerlukan penelusuran. Persamaan (7.7): waktu $\Theta(n)$, tetapi tumpukan $\Theta(n)$. "Selesaikan sisanya, lalu benahi simpul pertama" muncul lagi di Bab 10 dan 11. Buku: Subbab 7.6.7.*

Pembalikan Rekursif, Ditelusuri



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.

***Gambar 7.12**: panggilan terdalam mengembalikan 40; setiap panggilan yang kembali membalik satu panah dan memutus tautan maju simpulnya sendiri. Buku: Subbab 7.6.7.*

Biaya Operasi-Operasi Baru

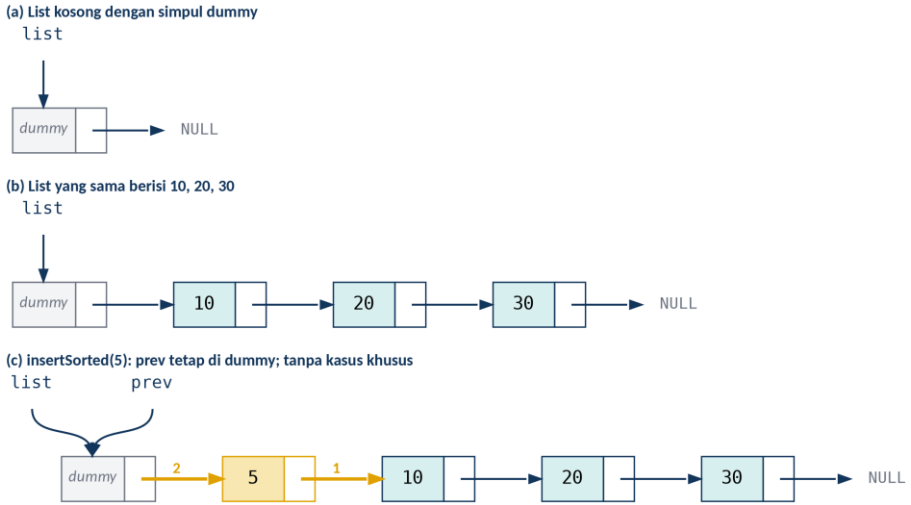
Tabel 7.3 Ongkos operasi-operasi Subbab 7.6 pada list berisi n simpul.

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`	$\Theta(n)$	$O(1)$

***Tabel 7.3**: setiap penelusuran bernilai $\Theta(n)$; hanya `insertAfter` dengan `p` yang sudah diketahui bernilai $O(1)$. Versi rekursif membayar $\Theta(n)$ di call stack. Buku: Subbab 7.6.9.*

Simpul Kepala *Dummy*

DEFINISI. *Single linked list dengan simpul kepala dummy* selalu diawali satu simpul tambahan yang datanya tidak bermakna; elemen nyata pertama adalah `dummy->next`, dan list kosong jika dan hanya jika `dummy->next` bernilai NULL.



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.

***Gambar 7.13**: semua kasus khusus sejauh ini muncul karena simpul pertama ditunjuk oleh `head`, bukan oleh medan `next`. Dengan dummy, menyisipkan 5 sebelum 10 adalah penyisipan sesudah dummy. Buku: Subbab 7.7.1.*

Operasi Tanpa Kasus Khusus

Kode 7.5 Dari `sll_dummy.cpp`: penyisipan dan penghapusan tanpa kasus khusus.

```
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};
}

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;
}
```

```
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
```

***Kode 7.5**: menghapus 5, elemen nyata pertama, memakai kode yang sama dengan menghapus 20 atau 99. Penelusuran mulai di `dummy->next`. Harganya: satu simpul per list. Buku: Subbab 7.7.2.*

Cara Ketiga: Pointer ke Tautan

Kode 7.6 Dari `sll_link.cpp`: `Node **link` mulai sebagai `&head` lalu menjadi `&node->next`.

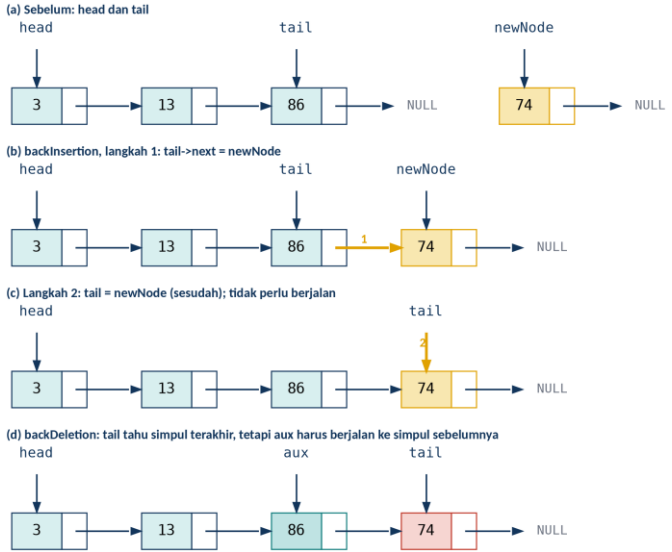
```
void insertSorted(Node **link, int x) {
    while (*link != nullptr && (*link)->data < x) link = &(*link)->next;
    *link = new Node{x, *link};          // rewrites head OR some node's next
}

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;
}
```

```
sorted:          5 -> 10 -> 20 -> 20 -> 20 -> 30 -> NULL
deleteKey(5):    10 -> 20 -> 20 -> 20 -> 30 -> NULL
```

***Kode 7.6**: `*link = ...` mengubah `head` atau sebuah medan `next` dengan baris yang sama. Tidak ada padanannya di Java atau Python; di sana dummy adalah alatnya. Varian mana? Pointer head: paling sederhana. Dummy: kode terpendek, satu simpul ekstra. Pointer ke tautan: kode pendek, hanya C dan C++. Buku: Subbab 7.7.3 dan 7.7.4.*

Pointer *Head* dan *Tail*



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`.

***Gambar 7.14**: penyisipan belakang menempel pada `tail` lalu memindahkannya: $O(1)$. Perpindahan antara "kosong" dan "satu simpul" harus mengubah kedua pointer. Buku: Subbab 7.8.1.*

Mengapa `tail` Tidak Mempercepat Penghapusan Belakang

> **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.

`tail` memberi tahu simpul *mana* yang dihapus, tetapi simpul **sebelum**-nya harus menjadi simpul terakhir yang baru.

Simpul bertaut tunggal mencatat siapa yang datang sesudahnya, tidak pernah siapa yang datang sebelumnya.

Jadi kita berjalan dari `head` sampai simpul yang `next`-nya `tail`: $n - 2$ langkah, $\Theta(n)$.

Pointer ke simpul kedua dari belakang hanya menolong sekali. Perbaikan sejatinya: tautan ke pendahulu di setiap simpul — Bab 9.

*Panel (d) dari **Gambar 7.14** memperlihatkan penelusurannya. Buku: Subbab 7.8.2.*

Varian-Varian Dibandingkan

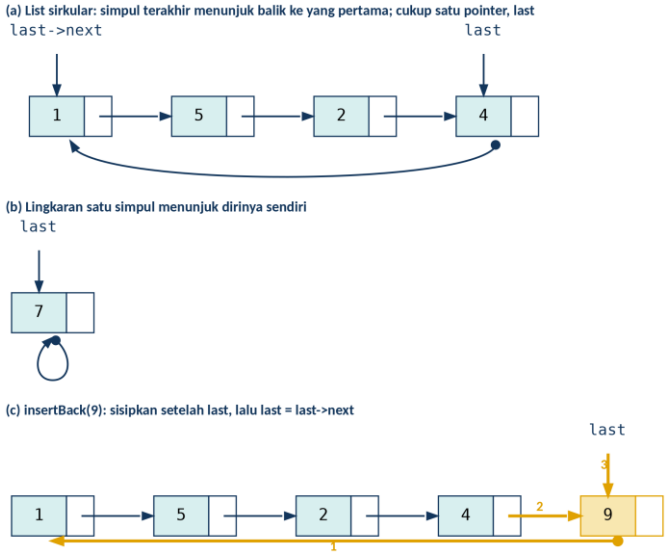
Tabel 7.4 Varian-varian *single linked list* dibandingkan, untuk list berisi n simpul.

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

```
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)
```

***Tabel 7.4**: hanya penghapusan belakang yang bertahan terhadap setiap varian. Hasil jalan `sll_head_tail.cpp`: pada list berisi 5, 4, dan 3 simpul, penelusurannya memakan 3, 2, dan 1 langkah, tepat $n - 2$. Buku: Subbab 7.8.3.*

Single Linked List Melingkar



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`.

***Gambar 7.15**: satu pointer, `last`, mencapai kedua ujung, karena simpul pertama adalah `last->next`. Penyisipan depan dan belakang serta penghapusan depan bernilai $O(1)$; penghapusan belakang tetap $\Theta(n)$. Lingkaran satu simpul menunjuk dirinya sendiri. Buku: Subbab 7.9.1.*

Penelusuran Memerlukan Tanda Berhenti yang Baru

```
  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
```

Kode 7.7 Dari `csll.cpp`: `deleteAfter` menjaga `last` tetap benar.

```
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;
}
```

*`while (aux != nullptr)` akan mencetak lingkaran selamanya; `while (aux != last->next)` akan berhenti sebelum mengunjungi apa pun. `do-while` menguji sesudah badannya. **Kode 7.7**: `del == prev` mendeteksi simpul satu-satunya. Buku: Subbab 7.9.1 dan 7.9.2.*

Sesi Membaca, Menit demi Menit

```
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)
```

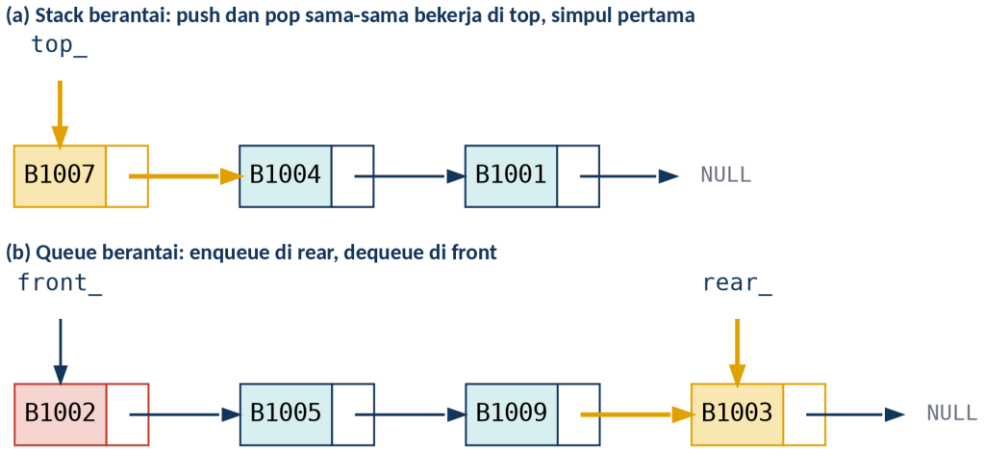
Dewi (1 menit, pemegang kartu prioritas) selesai pada $t = 1$; Budi selesai pada $t = 5$.

Ani dan Citra terus berputar dalam lingkaran yang menyusut tanpa kode “akhir list”.

Total 12 menit $= 1 + 5 + 2 + 4$: tidak ada waktu yang terbuang.

`prev` berdiri pada simpul sebelum pembaca yang sedang dilayani, seperti yang diperlukan `deleteAfter`: layanan bergiliran (round-robin) dengan jatah waktu 2 menit. Josephus (Soal Latihan 7.11) dan deteksi siklus (Soal Latihan 7.6) adalah dua soal klasik melingkar lainnya. Buku: Subbab 7.9.3.

Stack dan *Queue* di Atas Linked List



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_`.

***Stack*:** `push` = penyisipan depan, `pop` = penghapusan depan, keduanya $O(1)$; satu pointer, `top_`.

***Queue*:** `enqueue` di *tail* (`rear_`), `dequeue` di *head* (`front_`), keduanya $O(1)$. Arah sebaliknya membuat `dequeue` menjadi penghapusan belakang $\Theta(n)$.

***Gambar 7.16**: tanpa kapasitas dan tanpa aritmetika indeks, tidak seperti Bab 5. Buku: Subbab 7.10.1.*

Queue Berantai

Kode 7.8 Dari `linked_stack_queue.cpp`: `dequeue` untuk antrean pemesanan Pustaka Ceria.

```
    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;
    }
```

```
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
```

***Kode 7.8**: `if (front_ == nullptr) rear_ = nullptr;` — tanpanya, `rear_` menunjuk simpul yang sudah dibebaskan, dan versi yang menguji `rear_` akan menulis ke memori yang sudah dibebaskan (Soal Latihan 7.12). Buku: Subbab 7.10.2.*

Tiga Fitur Keamanan Kelas-Kelasnya

Kode 7.9 Dari `linked_stack_queue.cpp`: tipe simpul bersifat privat, penyalinan dilarang, destruktor membebaskan setiap simpul.

```
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
```

Simpul privat: tidak ada kode luar yang dapat merusak tautan.

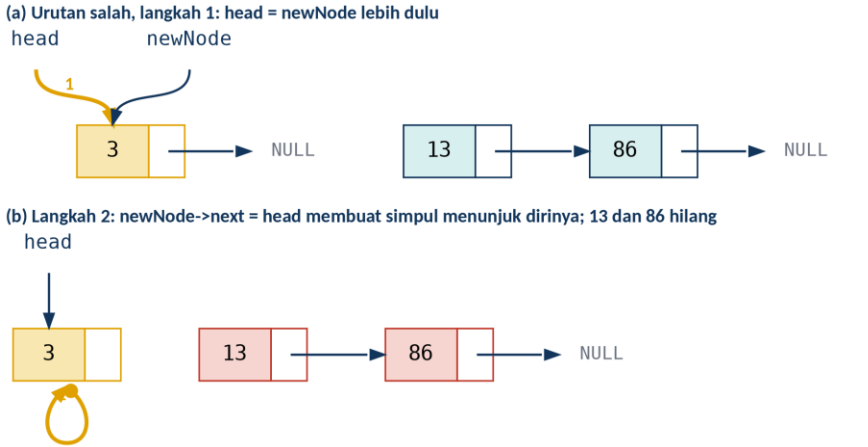
Destruktor (RAII): simpul yang tersisa dalam struktur pada akhir `main` dilepaskan secara otomatis.

Tanpa salinan (aturan tiga): salinan bawaan akan berbagi `top_`, dan kedua destruktor akan menghapus setiap simpul.

***Kode 7.9**: Java dan Python tidak memerlukan destruktor. Berantai atau berbasis larik? Larik memakai lebih sedikit memori dan lebih cepat dalam praktik; `std::stack` memakai `std::deque`. Berantai unggul untuk elemen besar atau bila kapasitas tetap tidak dapat diterima. Buku: Subbab 7.10.2 dan 7.10.3.*

Empat *Bug* Memori, Satu Kesalahan Urutan

Kebocoran memori; pemakaian sesudah pembebasan; tautan menggantung; pembebasan ganda.



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.

***Gambar 7.17**: `head` berpindah lebih dulu, lalu simpul baru menunjuk dirinya sendiri; 13 dan 86 bocor. Keempat bug lain: simpul tak pernah dihapus, dipakai sesudah `delete`, masih ditunjuk `next`/`tail`, atau dihapus dua kali — biasanya tanpa gejala. Buku: Subbab 7.11.1.*

Valgrind: Kebocoran

```
==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== 48 (16 direct, 32 indirect) bytes in 1 blocks are definitely lost in loss record 2 of 2
==1776==    by 0x10920F: build() (memory_bugs.cpp:17)
==1776==    by 0x109369: main (memory_bugs.cpp:42)
==1776== LEAK SUMMARY:
==1776==    definitely lost: 16 bytes in 1 blocks
==1776==    indirectly lost: 32 bytes in 2 blocks
```

Baca laporannya **dari bawah ke atas**: 16 bita hilang *secara pasti* (*definitely*), 32 hilang *secara tidak langsung* (*indirectly*).

Hanya simpul pertama yang ditunjuk `head`; dua lainnya hanya terjangkau dari simpul yang hilang.

Dialokasikan oleh `build()` di baris 17, dipanggil dari `main` di baris 42. Perbaiki kebocoran pastinya dan sisanya ikut beres.

`memory_bugs.cpp` menyalakan satu bug per argumen: `leak`, `uaf`, `dangle`, `double`. Kompilasi dengan `-g -O0`; Valgrind berjalan 20 sampai 50 kali lebih lambat. Buku: Subbab 7.11.1 dan 7.11.2.

Valgrind: Pemakaian Sesudah Pembebasan

```
==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==    by 0x1092C3: clearRight(Node*&) (memory_bugs.cpp:30)
==1786==    by 0x1093CF: main (memory_bugs.cpp:48)
==1786==  Block was alloc'd at
==1786==    by 0x10920F: build() (memory_bugs.cpp:17)
==1786==    by 0x109369: main (memory_bugs.cpp:42)
1 2 3
2 3
```

Program mencetak `2 3`, nilai yang benar, dari simpul yang sudah dihapus. Tiga jejak: pembacaan yang tidak sah (`show`, baris 22), tempat blok dibebaskan (`clearRight`, baris 30), dan tempat ia dialokasikan (`build`, baris 17). Dengan `-O2` dan tanpa Valgrind program itu crash. Buku: Subbab 7.11.2.

Sertifikatnya, dan AddressSanitizer

```
==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` dan `ERROR SUMMARY: 0 errors`: setiap program C++ di bab ini menghasilkan kedua baris itu.

AddressSanitizer: kompilator menyisipkan pemeriksaan dan meracuni *shadow memory* di sekitar setiap blok; sekitar dua kali lebih lambat; Linux, macOS, dan Windows.

`-fsanitize=address,undefined`; LeakSanitizer melaporkan kebocoran saat program selesai.

Buku: Subbab 7.11.2 dan 7.11.3.

ASan: Tautan Menggantung

```
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:
    #1 0x55d758b73831 in backDeletionWrong(Node*) memory_bugs.cpp:37
    #2 0x55d758b73c8d in main memory_bugs.cpp:51
previously allocated by thread T0 here:
    #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*)
```

`backDeletionWrong` menghapus simpul terakhir di baris 37 tetapi membiarkan tautannya; `show` membaca `data` pada offset 0 dari simpul 16 bita. Untuk `double`, ASan berhenti pada pembacaan `next` (`8 bytes inside`) sebelum `delete` kedua; "direct" dan "indirect" milik LeakSanitizer sepadan dengan "definitely" dan "indirectly" milik Valgrind. Buku: Subbab 7.11.3.

Daftar Periksa untuk Setiap Fungsi List

1. **Simpan, lalu pindahkan, lalu hapus.**
2. **Jangan sisakan pointer ke simpul yang sudah dibebaskan:** `head`, `tail`, `next` milik pendahulu, pointer pembantu yang disimpan.
3. **Setiap `new` punya tepat satu pemilik yang menghapusnya;** penyalinan menyalin simpul-simpulnya atau dilarang.
4. **Periksa keempat bentuk:** kosong, satu simpul, serta simpul pertama, tengah, dan terakhir.
5. **Jalankan alatnya** setiap kali: ASan atau `valgrind --leak-check=full`.

Rangkaian uji yang belum pernah dijalankan di bawah pemeriksa memori membuktikan jauh lebih sedikit daripada kelihatannya. Buku: Subbab 7.11.4.

C++: `std::forward_list` dan `std::list`

Kode 7.10 Dari `std_lists.cpp`: `std::forward_list` yang bertaut tunggal.

```
    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);
```

```
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
```

***Kode 7.10**: `insert_after`, `before_begin()` (gagasan dummy), tanpa `push_back`, tanpa `size()`. `std::list` bertaut ganda. Buku: Subbab 7.12.1.*

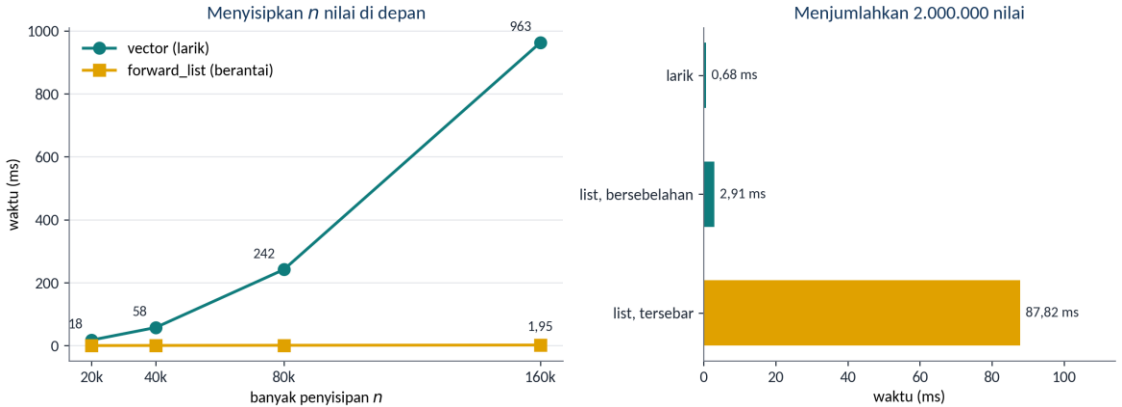
Pustaka-Pustaka Dibandingkan

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

***Tabel 7.5**: `LinkedList` Java bertaut ganda; `ArrayList` dan `ArrayDeque` biasanya menang. `list` Python adalah larik dinamis; `deque` adalah double linked list berisi blok 64 slot. Simpul Python buatan sendiri memakan sekitar 48 bita. Buku: Subbab 7.12.1–7.12.3.*

Mengukur: Larik Melawan List



Gambar 7.18 Hasil pengukuran: 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: melipatduakan n mengalikan waktu `vector` sekitar 4 (Persamaan (7.4)), waktu list sekitar 2; pada $n = 160.000$ list sekitar 500 kali lebih cepat.

Penelusuran: larik 0,68 ms, list bersebelahan 2,91 ms, list tersebar 87,82 ms — semuanya $\Theta(n)$; *cache*-lah yang menentukan.

***Gambar 7.18**: untuk sebagian besar beban kerja larik menang, karena menemukan posisi adalah penelusuran yang lambat. Buku: Subbab 7.12.4.*

Profitina dalam Praktik

Umpan yang tumbuh di depan: unggahan baru masuk ke atas — `frontInsertion`; simpan k yang terbaru dengan `deque` berpanjang maksimum.

Antrean pemindaian ulang: pekerjaan yang jatuh tempo menunggu dalam *queue* FIFO — `enqueue` di belakang, `dequeue` di depan.

Bergiliran dengan adil: layanan bergiliran atas merek-merek, list melingkar.

Riwayat yang hanya bertambah: tren skor mingguan — `backInsertion` dengan *tail* dan penelusuran maju.

Kenali ujung mana yang disentuh operasi-operasinya, lalu pilih struktur yang operasi murahnya persis itu. Buku: Subbab 7.13.

Kesalahan yang Sering Terjadi

Memindahkan `head` untuk menyusuri list; kondisi gelung yang salah (`aux`, `aux->next`, `aux->next->next`).

Mendereferensi NULL; penugasan dengan urutan yang salah (**Gambar 7.17**); menghapus sebelum memindahkan.

Lupa memutus tautan terakhir, atau pointer kedua (`tail`, `rear_`).

Uji NULL untuk menghentikan penelusuran melingkar; membocorkan list; memercayai keluaran yang benar.

Rekursi atas list yang sangat panjang; memilih list untuk akses acak (Persamaan (7.3)).

Buku: Subbab 7.14.

Ringkasan: Lima Hal yang Perlu Diingat

1. Simpul memegang data dan `next`; list dicapai dari `head` dan berakhir di NULL. Akses berdasarkan posisi $\Theta(n)$; perubahan pada posisi yang dipegang $O(1)$.

2. Sisip: tautkan simpul baru ke penerusnya, lalu pendahulunya ke simpul baru. Hapus: simpan, pindahkan setiap pointer melewatinya, hapus paling akhir.

3. Simpul *dummy* atau pointer ke tautan menghapus kasus khusus simpul pertama; *tail* membuat penyisipan belakang $O(1)$, tidak pernah penghapusan belakang.

4. List melingkar lewat `last` mencapai kedua ujung; *stack* dan *queue* berantai bernilai $O(1)$ dan tidak pernah penuh.

5. Buktikan keamanan memori dengan Valgrind atau ASan; utamakan larik kecuali operasinya berada pada posisi yang sudah dipegang.

Buku: Subbab 7.15.

Pertemuan Berikutnya

Bab 8 — Evaluasi dan Pembahasan II: Stack, Pointer, dan Linked List dalam Penerapan. Dua belas soal dari Bab 5, 6, dan 7, diselesaikan dari halaman kosong.

Gambar pointernya lebih dulu, telusuri dengan tangan, baca tiga program, dan pelajari satu penyelesaian keliru yang menggoda di bawah pemeriksa memori.

Sebelum itu: kerjakan Soal Latihan 7.1, 7.2, 7.5, dan 7.9, lalu jalankan satu *bug* dari `memory_bugs.cpp` di bawah AddressSanitizer.

Bawalah keempat bentuk dari daftar periksa: kosong, satu simpul, pertama, tengah, terakhir. Buku: Subbab 7.16.

Kuliah 8 — Evaluasi dan Pembahasan II: Stack, Pointer, dan Linked List dalam Penerapan

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 8

Dosen: Irfan Subakti — yifana@gmail.com

Tidak ada struktur baru hari ini: dua belas soal tentang Bab 5, 6, dan 7, masing-masing digambar sebelum dikodekan, ditelusuri dengan tangan, diselesaikan dalam C++ (beserta Java dan Python), dipatahkan oleh penyelesaian keliru yang menggoda, dan ditutup dengan varian uji diri. Buku: Bab 8, Subbab 8.1–8.15.

Cakupan Pertemuan Ini

1. **Sebuah metode** — tujuh langkah untuk soal apa pun yang berisi kotak-kotak yang menunjuk ke kotak lain.
2. **Lima soal yang saling terkait** — melengkapi stack larik, menelusuri list dua arah, genap naik dan ganjil turun, membalik kata, membuang nilai ganda.
3. **Tujuh soal baru** — mencocokkan tanda kurung, queue sirkular, infiks ke postfiks, tiga cara mengirim argumen, pembalikan di tempat, penggabungan list terurut, queue berantai.
4. **Menengok ke belakang** — soal-soal berdampingan, Profitina, kesalahan yang sering terjadi.

Untuk bug memori, kegagalannya dilaporkan oleh pemeriksa memori, dan kita membaca laporannya bersama. Buku: Subbab 8.1.

Titik Tolak Kita

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

Tabel 8.1: Bab 5 memberi stack dan queue, Bab 6 pointer dan tiga cara mengirim argumen, Bab 7 singly linked list. Buku: Subbab 8.1.

Satu Gagasan di Balik Hampir Setiap Soal

> **WAWASAN KUNCI.** *Sebelum mengubah sebuah tautan, pastikan Anda masih memiliki jalan menuju semua yang akan Anda perlukan sesudah perubahan itu.*

Stack: bacalah butir teratas sebelum `top` digeser darinya.

Penelusuran menurut petunjuk: simpan banyaknya langkah sebelum pointer meninggalkan simpul yang memuatnya.

Pembalikan list: simpan `cur->next` sebelum menimpanya.

Penghapusan: tautkan melompati simpul sebelum membebaskannya. **Queue:** setel ulang `tail` ketika simpul terakhir hilang.

Bila program hari ini keliru, hampir selalu aturan inilah yang dilanggar, dan penyelesaian-penyelesaian keliru melanggarnya dengan semua cara itu. Buku: Subbab 8.1.

Cara Menghadapi Soal Pointer

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?

Tabel 8.2: gambarlah setiap simpul sebagai kotak dengan bagian data dan bagian pointer, setiap variabel pointer sebagai panah berlabel, dan `nullptr` *secara eksplisit. Kasus tepinya hampir selalu sama: kosong, satu simpul, pertama, terakhir, tetangga yang sama. Buku: Subbab 8.2.*

Satu Simpul, Tiga Bahasa

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`

Tabel 8.3: hanya C++ yang membuat Anda menghancurkan simpul, jadi hanya program C++ yang dapat bocor atau memakai simpul yang sudah dibebaskan. Java dan Python tidak dapat mengubah variabel pemanggil, jadi keduanya mengembalikan simpul pertama yang baru: `head = insertSorted(head, ...)`*. Setiap program C++ juga dijalankan di bawah AddressSanitizer dan Valgrind: tanpa peringatan, tanpa akses tidak sah, tanpa satu bita pun bocor. Buku: Subbab 8.2.*

Soal 8.1: Melengkapi Stack Larik

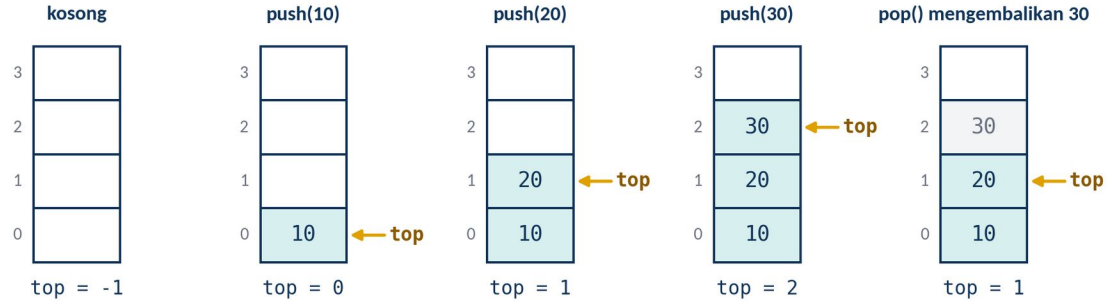
SOAL 8.1. Complete the program: implement (a) `pop()`, (b) `push()`, (c) `peek()` and (d) `isEmpty()` for a class with `int a[MAX]` and `top`, whose constructor sets `top = -1`. The driver pushes 10, 20, 30 and pops once.

DEFINISI. *Invarian stack*: butir-butirnya tepat `a[0]`, ..., `a[top]`, dari dasar sampai puncak, dan $-1 \le$ `top` $\le$ `MAX` -1.

$$\text{ukuran} = \text{top} + 1\,,\ \text{kosong} \Leftrightarrow \text{top} = -1\,,\ \text{penuh} \Leftrightarrow \text{top} = \text{MAX} - 1\,. \tag{8.1}$$

Persamaan (8.1): `top = -1` menentukan segalanya: `top` adalah indeks butir teratas. Periksa di kedua ujung: ukuran 0 tanpa butir, 1000 ketika penuh. Buku: Subbab 8.3.1 dan 8.3.2.

Soal 8.1: Dengan Tangan



Indeks MAX-1 = 999 jauh di atas; hanya empat sel pertama yang digambar. Angka 30 kelabu masih ada di a[2], tetapi bukan lagi butir tumpukan.

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.

Gambar 8.1: `push` berarti "naikkan, lalu simpan"; `pop` berarti "baca, lalu turunkan". Sesudah pop, 30 masih ada di `a[2]` tetapi tidak lagi di antara `a[0]` dan `a[top]`, jadi ia bukan butir lagi. Buku: Subbab 8.3.3.

Soal 8.1: Penyelesaiannya

Kode 8.1 Dari `p801_stack.cpp`: `push` dan `pop`, dilengkapi.

```
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
}
```

```
30 popped from stack
peek -> 20, size 2
pop -> 20
pop -> 10
```

Kode 8.1: stack kosong melempar eksepsi, karena 0 atau -1 adalah butir yang sah. `peek` mengembalikan `a[top]`; semuanya $O(1)$. Buku: Subbab 8.3.4 dan 8.3.5.

Soal 8.1: Penyelesaian Keliru yang Menggoda

```
int pop() { return a[--top]; }   // KELIRU: menggeser top dulu, baru membaca
```

```
20 popped from stack
10 popped from stack
isEmpty -> false
```

Pop pertama menggeser `top` dari 2 ke 1 *sebelum* membaca dan mengembalikan 20, butir *di bawah* puncak; 30 tidak pernah dikembalikan.

Kesalahan cerminnya pada `push`, `a[top++] = x`, menulis ke `a[-1]`: perilaku tak terdefinisi, ditangkap AddressSanitizer.

> **UJI DIRI.** Dengan `top = 0` ("sel kosong berikutnya"): `push` menjadi `a[top++] = x`, `pop` menjadi `return a[--top];`, `peek` menjadi `a[top - 1]`, `isEmpty` menjadi `top == 0`, dan ukuran = `top`. Kedua konvensi benar; mencampurnya tidak pernah benar.

Buku: Subbab 8.3.6 dan 8.3.7.

Soal 8.2: Menelusuri List Dua Arah Menurut Petunjuk

> **SOAL 8.2.** Each input line is *direction step data*: `b` backward or `f` forward, the number of nodes to move, and the letter in the node; input stops at direction `0`. Print the letters met on the walk. Sample: `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` gives `strukdat`.

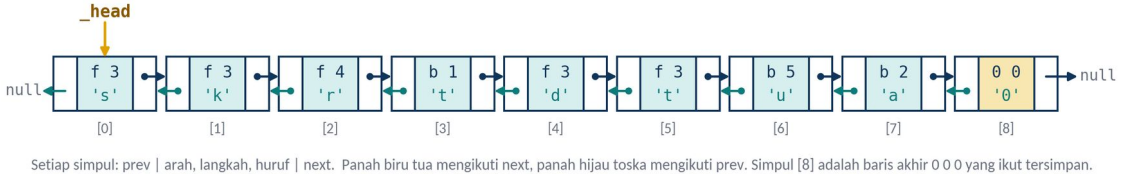
Dua panah per simpul: `next` dan `prev` — sekilas pandang ke Bab 9.

Bergerak mundur adalah `cur = cur->prev`. Menambah di belakang memasang **dua** panah: `last->next = newNode` *dan* `newNode->prev = last`.

> **DEFINISI.** *Invarian list dua arah*: untuk setiap simpul `p` yang memiliki penerus, `p->next->prev == p`; `prev` simpul pertama dan `next` simpul terakhir bernilai `nullptr`.

Buku: Subbab 8.4.1 dan 8.4.2.

Soal 8.2: List yang Dibangun dari Contoh



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.

Gambar 8.2: `do`–`while` yang diberikan membaca satu baris, menyisipkannya, dan baru kemudian menguji `'0'`, jadi baris akhir juga menjadi simpul: sembilan simpul, bukan delapan. Penelusuran berhenti ketika tiba di simpul itu. Buku: Subbab 8.4.2.

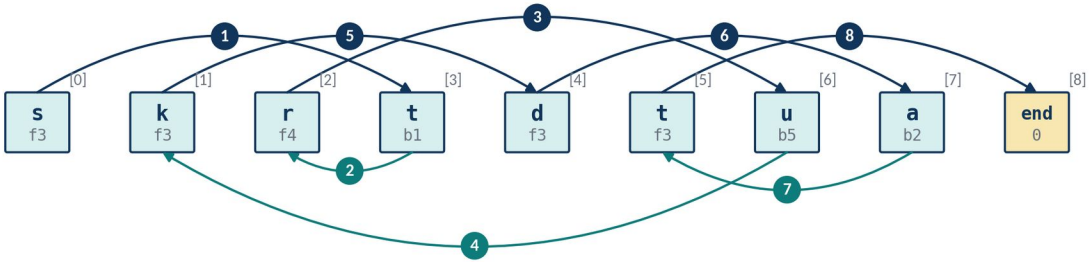
Soal 8.2: Penelusuran, dengan Tangan

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	—

Tabel 8.4: cetak huruf simpul saat ini, lalu bergerak menurut petunjuk simpul itu sendiri. Kolom "Mencetak" terbaca `strukdat`. Buku: Subbab 8.4.3.

Soal 8.2: Delapan Gerakan



Langkah 1 sampai 8 berurutan; huruf tercetak s, t, r, u, k, d, a, t. Atas: maju (next); bawah: mundur (prev).

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.

Gambar 8.3: gerakan maju di atas, mundur di bawah; setiap simpul huruf dikunjungi tepat sekali — sifat masukan ini, bukan sifat setiap masukan. Buku: Subbab 8.4.3.

Soal 8.2: Apa yang Dapat Salah pada Masukan

Tanpa baris akhir: `cin` gagal dan gelung menyisipkan tanpa henti; ujilah apakah pembacaan berhasil.

Gerakan melewati ujung: periksa `cur != nullptr` di dalam gelung gerak.

Petunjuk berputar dalam lingkaran: `f 1` bersebelahan dengan `b 1`. Ke mana penelusuran pergi hanya bergantung pada simpul saat ini, jadi simpul yang berulang akan berulang selamanya.

$$\text{kunjungan} > n \Rightarrow \text{ada simpul yang dikunjungi dua kali} \Rightarrow \text{penelusuran tidak pernah mencapai akhir}. \tag{8.2}$$

Persamaan (8.2) adalah prinsip sarang merpati: kunjungan yang lebih banyak daripada simpul berarti ada simpul yang dikunjungi dua kali. Satu pencacah dan satu perbandingan per kunjungan mengubah program yang dapat macet menjadi program yang selalu berhenti. Buku: Subbab 8.4.4.

Soal 8.2: Penyelesaiannya

Kode 8.2 Dari `p802_deque_walk.cpp`: `traverse`, dengan kedua penjaganya.

```
    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]";
        }
    }
```

```
strukdat
list
aba [the directions go around in a circle]
xy [the walk left the list]
```

Kode 8.2: membangun list berbiaya $\Theta(n^2)$ tanpa tail; penelusuran pada contoh memakan 24 gerakan. Buku: Subbab 8.4.5.

Soal 8.2: Penyelesaian Keliru yang Menggoda

```
  for (int k = 0; k < cur->step; k++)
      cur = (cur->direction == 'b') ? cur->prev : cur->next;   // KELIRU
```

```
stttttttttttttttttt
```

`cur` berubah di dalam gelung, sehingga `cur->step` dan `cur->direction` dibaca dari simpul *yang berbeda* pada setiap putaran; penelusuran terperangkap di antara simpul 2 dan 3.

Perbaikannya: **salin apa yang Anda perlukan dari sebuah simpul sebelum meninggalkannya** — `step` dan `direction` ke variabel lokal.

> **UJI DIRI.** `f 2 l`, `f 2 s`, `b 1 i`, `f 1 t`, `0 0 0` mencetak `list`. Mengubah baris kedua menjadi `f 4 s` mengirim penelusuran ke simpul yang tidak ada: `lis [the walk left the list]`.

Buku: Subbab 8.4.7 dan 8.4.8.

Soal 8.3: Genap Naik, Ganjil Turun, Hanya dengan Linked List

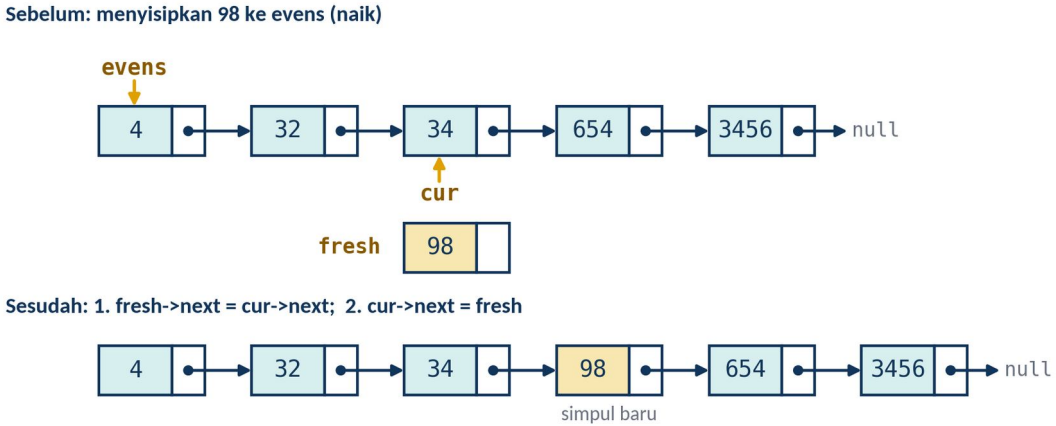
> **SOAL 8.3.** Read N values; print the even ones ascending, then the odd ones descending, with linked lists and no arrays. Input 4, 32, 34, 543, 3456, 654, 567, 87, 6789, 98 gives 4, 32, 34, 98, 654, 3456, 6789, 567, 543, 87.

***Insertion sort* pada linked list**, tanpa penggeseran: `evens` naik, `odds` turun, **satu** fungsi penyisipan yang menerima urutannya sebagai *pointer ke fungsi*, `bool (*before)(int, int)`.

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2). \tag{8.3}$$

Persamaan (8.3): paling banyak 45 langkah untuk sepuluh nilai; untuk sejuta nilai, merge sort. Buku: Subbab 8.5.1–8.5.3.

Soal 8.3: Menyisipkan 98



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.

`cur` melihat `cur->next`, jadi ia tetap berdiri pada **pendahulu**, tempat perubahan terjadi.

Tautkan simpul baru ke 654 dulu, *baru kemudian* 34 ke simpul baru; urutan sebaliknya menghilangkan 654 dan 3456.

`!before(value, cur->next->data)` menaruh nilai yang sama sesudah nilai kembarnya: **stabil**.

Gambar 8.4. Buku: Subbab 8.5.3.

Soal 8.3: Sepuluh Penyisipan

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

Tabel 8.5: mencetak `evens`, lalu `odds`, menghasilkan keluaran yang diharapkan. Buku: Subbab 8.5.3.

Soal 8.3: Penyelesaiannya

Kode 8.3 Dari `p803_evens_odds.cpp`: satu penyisipan terurut untuk kedua urutan.

```
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
```

```
-2
0
4
4
9
```

Kode 8.3: `Node *&head` *membuat* `head = fresh` *sampai ke* `main`*. Java mengirim lambda; Python mengirim fungsinya. Nol adalah genap. Buku: Subbab 8.5.4 dan 8.5.5.*

Soal 8.3: Penyelesaian Keliru yang Menggoda

```
  if (x % 2 == 1) insertSorted(odds, x, descending);    // KELIRU untuk x negatif
  else            insertSorted(evens, x, ascending);
```

```
-7
-3
-2
0
4
4
9
```

Sejak C++11, `/` memotong ke arah nol, jadi `-7 % 2` bernilai -1, bukan 1: -7 dan -3 bergabung dengan bilangan genap. Java sama; `%` Python memberi 1.

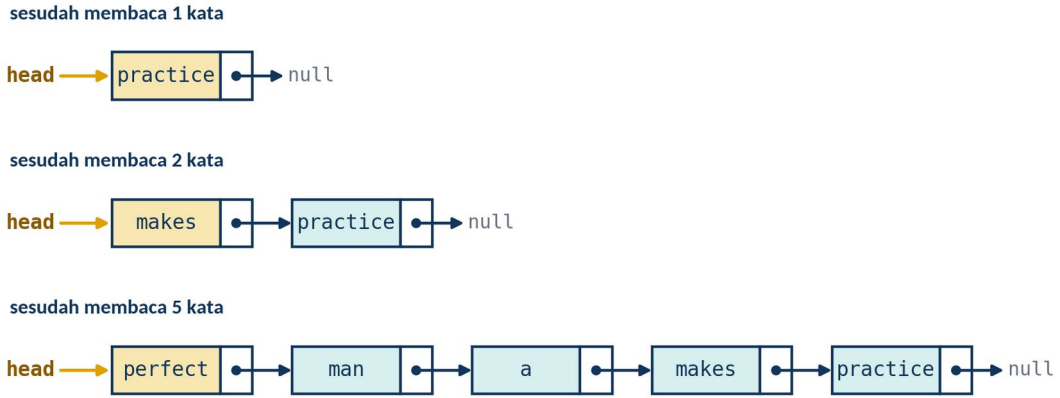
Ujilah paritas dengan `x % 2 == 0` dan biarkan “ganjil” menjadi sisanya.

> **UJI DIRI.** Untuk 5, 0, -4, 15, -9, 8: yang benar -4, 0, 8, 15, 5, -9; yang keliru -9, -4, 0, 8, 15, 5. Enam simpul dibuat, enam dibebaskan.

Buku: Subbab 8.5.6 dan 8.5.7.

Soal 8.4: Membalik Urutan Kata Sebuah Kalimat

SOAL 8.4. Reverse a string using a linked list. Input `practice makes a man perfect`; output `perfect man a makes practice`.



Gambar 8.5 Membangun list Soal 8.4 dengan penyisipan depan: setiap kata baru menjadi simpul pertama, sehingga list membaca kalimatnya dari belakang.

Gambar 8.5: "kata terakhir yang dibaca, kata pertama yang dicetak" adalah stack, dan `frontInsertion` adalah sebuah push. `head = new WordNode{w, head};` mengevaluasi ruas kanan lebih dulu, sehingga simpul baru menerima `head` yang lama. Pembalikan terjadi selama pembangunan. Buku: Subbab 8.6.1–8.6.3.

Soal 8.4: Penyelesaiannya

Kode 8.4 Dari `p804_reverse_words.cpp`: `istringstream`, *push*, lalu cetak dan bebaskan.

```
    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;
    }
```

```
$ ./p804 < p804.txt
perfect man a makes practice
$ ./p804 < p804_check.txt
friends are queues and stacks
```

Kode 8.4: `>>` melompati spasi berapa pun banyaknya; bendera `first` menaruh spasi di antara kata, tidak pernah di akhir. $\Theta(L)$ untuk baris berisi L karakter. Buku: Subbab 8.6.4 dan 8.6.5.

Soal 8.4: Penyelesaian Keliru yang Menggoda

```
  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';
```

```
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
```

Kalimat berisi 100 karakter atau lebih menulis melewati `str`; tidak ada yang mewajibkan program *crash*. AddressSanitizer berhenti pada operasi tulis buruk yang pertama, baris 15. Perbaikannya: `std::string` dengan `getline`.

UJI DIRI. Satu kata, `queue`: satu simpul, mencetak `queue`. Baris kosong: tanpa simpul. Dengan `backInsertion` sebagai ganti `pushFront`: `practice makes a man perfect`, tidak berubah.

Buku: Subbab 8.6.6 dan 8.6.7.

Soal 8.5: Membuang Nilai Ganda dari Sebuah List

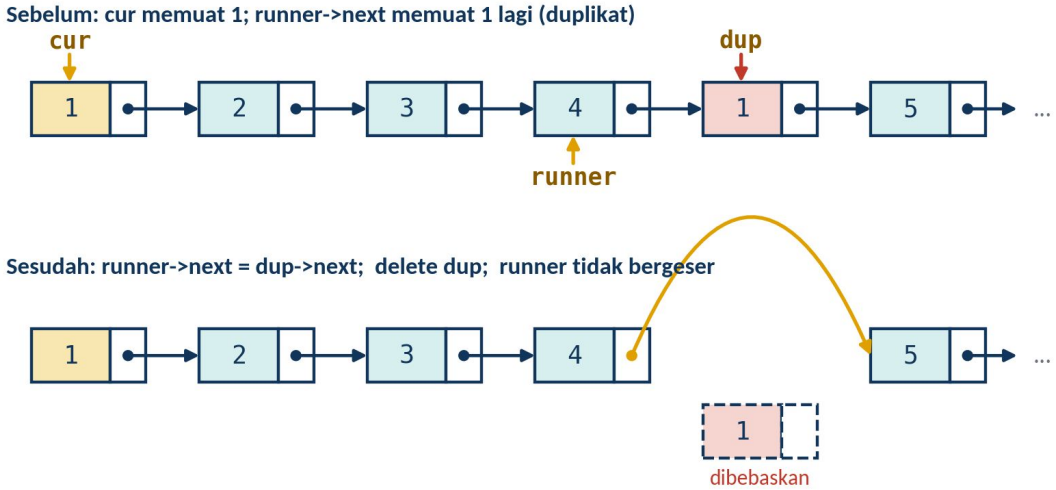
SOAL 8.5. Complete `remove_duplicate()`, which removes the nodes with the same data. Input 1, 2, 3, 4, 1, 5, 6, 3, 2, -1; output `1 2 3 4 1 5 6 3 2 -1`, then `1 2 3 4 5 6`.

Pertahankan kemunculan **pertama** setiap nilai. Penanda akhir -1 yang ikut tersimpan **bukan data**: `slist_popBack` kecil membuangnya.

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2) \text{ perbandingan, dan } O(1) \text{ memori tambahan.} \tag{8.4}$$

Persamaan (8.4): himpunan nilai yang sudah dilihat akan memberi waktu $O(n)$ dengan memori $O(n)$ (Bab 15). Buku: Subbab 8.7.1–8.7.3.

Soal 8.5: Lepas Tautan, Bebaskan, Jangan Bergerak



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.

1. `dup = runner->next;` **2.** `runner->next = dup->next;` **3.** `delete dup;`

Sesudah penghapusan `runner` **tetap di tempat**, karena `next`-nya yang baru belum diperiksa.

Gambar 8.6: membebaskan sebelum langkah 2 akan membaca `dup->next` dari memori yang sudah dibebaskan. Buku: Subbab 8.7.3.

Soal 8.5: Penyelesaiannya

Kode 8.5 Dari `p805_remove_duplicates.cpp`: `remove_duplicate` dengan `cur` dan `runner`.

```
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
            }
        }
    }
}
```

```
$ ./p805 < p805.txt
1 2 3 4 1 5 6 3 2 -1
1 2 3 4 5 6
nodes left: 6
```

Kode 8.5: cabang `else` satu-satunya tempat `runner` bergerak. Java dan Python hanya memerlukan langkah 2. Buku: Subbab 8.7.3–8.7.5.

Soal 8.5: Penyelesaian Keliru yang Menggoda

```
if (runner->next->data == cur->data) {
    SListNode *dup = runner->next;
    runner->next = dup->next;
    delete dup;
}
runner = runner->next;          // KELIRU: tetap maju sesudah penghapusan
```

```
7 7 7 2 7 2 -1
7 7 2
nodes left: 3
```

Pada `7 7 7 2 7 2 -1` angka 7 yang ketiga lolos: sesudah penghapusan, baris yang keliru memindahkan `runner` ke simpul yang tidak pernah dibandingkan. Contoh soal tidak memiliki salinan yang bersebelahan, jadi ia lolos.

UJI DIRI. Pada contoh sepuluh simpul, `remove_duplicate` melakukan $9+7+5+3+2+1+0=27$ perbandingan, di bawah 45 dari Persamaan (8.4), karena setiap penghapusan memperpendek setiap pemindaian berikutnya.

Buku: Subbab 8.7.6 dan 8.7.7.

Soal 8.6: Mencocokkan Tanda Kurung

> **SOAL 8.6.** Putuskan apakah tanda kurung `()`, `[]`, dan `{}` dalam sebaris teks seimbang; jika tidak, sebutkan apa yang salah dan di mana. Ujilah `(a+b)*[c-d]`, `{[()()]}`, `([)]`, `((x)`, `x)]`, dan baris kosong.

Setiap kurung tutup harus cocok dengan kurung buka **terbaru** yang masih terbuka: terakhir masuk, pertama keluar — stack.

Tiga cara gagal: kurung tutup pada stack kosong, kurung tutup yang tidak cocok dengan puncak, kurung buka yang tersisa di akhir.

Untuk `([)]`: `(` dan `[` di-*push*; `)` bertemu `[` di puncak.

Kapasitas stack dipilih saat program berjalan (`new char[capacity]`); destruktornya memanggil `delete[]` pada setiap jalur kembali, dan penyalinan dilarang. Waktu $\Theta(L)$. Buku: Subbab 8.8.1.

Soal 8.6: Penyelesaiannya

Kode 8.6 Dari `p806_brackets.cpp`: `check`, dengan ketiga pesan kegagalannya.

```
        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";
            }
            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();
        }
    }
```

```
"([)]" -> ')' at 3 cannot close '[' from 2
"((x)" -> '(' at 1 is never closed
"x)]" -> ')' at 2 has no opening partner
```

Kode 8.6: posisi dihitung mulai dari 1, sehingga pesannya menunjuk karakter yang tepat. Buku: Subbab 8.8.1.

Soal 8.6: Penyelesaian Keliru yang Menggoda

Mencacah: tambah satu untuk kurung buka, kurangi satu untuk kurung tutup, gagal di bawah nol, berhasil di nol.

```
"(a+b)*[c-d]" -> balanced
"{[()()]}" -> balanced
"([)]" -> balanced
"((x)" -> not balanced
"x)]" -> not balanced
"(]" -> balanced
```

Pencacah tahu *berapa banyak* kurung yang terbuka, bukan *yang mana*, sehingga ia menerima `([)]` dan bahkan `(]`. Mengingat urutan adalah pekerjaan stack.

> **UJI DIRI.** `{(a)[b]}}`: stack kosong di posisi 8, jadi `'}' at 9 has no opening partner`. `[(])`: `']' at 3 cannot close '(' from 2`.

Buku: Subbab 8.8.1.

Soal 8.7: Queue Sirkular untuk Permintaan Peminjaman

SOAL 8.7. Queue berisi empat permintaan peminjaman dalam larik, yang memakai ulang slot yang dikosongkan `dequeue`: *enqueue* `B1001` sampai `B1005`; *dequeue* dua kali; *enqueue* `B1005` dan `B1006`; *dequeue* semuanya.

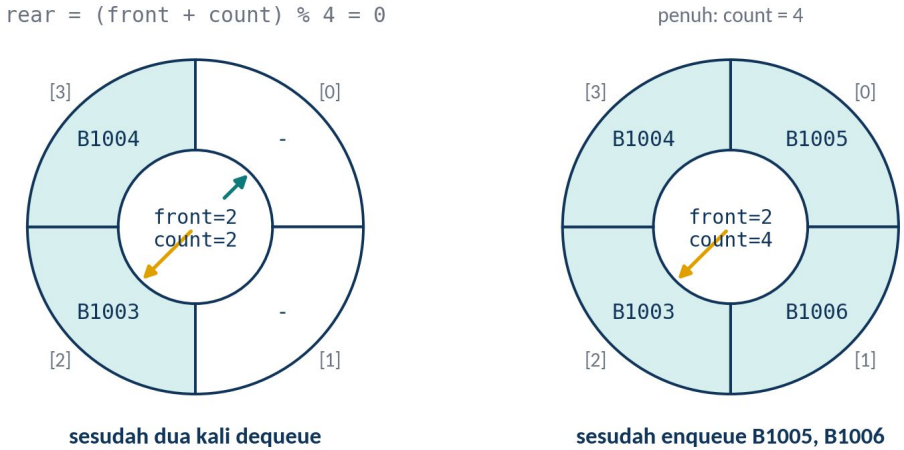
Simpan `front` (permintaan tertua) dan `count` (berapa yang menunggu); sisanya mengikuti.

$$\text{rear} = (\text{front} + \text{count}) \bmod \text{CAPACITY}\,,\ \text{front} \leftarrow (\text{front} + 1) \bmod \text{CAPACITY} \text{ sesudah dequeue}\,. \tag{8.5}$$

Kosong berarti `count == 0`, penuh berarti `count == CAPACITY`: keduanya tidak pernah tertukar.

Persamaan (8.5): sisa bagi membengkokkan larik menjadi cincin. Dengan hanya `front` dan `rear`, cincin penuh dan cincin kosong tampak sama. Buku: Subbab 8.8.2.

Soal 8.7: Cincinnya



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.

Gambar 8.7: sesudah dua dequeue, $(2+2) \bmod 4 = 0$, jadi `B1005` masuk ke sel yang ditinggalkan `B1001`; larik terbaca `B1005 B1006 B1003 B1004`, tetapi urutan queue dimulai di `front` = 2 dan berputar. Buku: Subbab 8.8.2.

Soal 8.7: Penyelesaiannya

Kode 8.7 Dari `p807_circular_queue.cpp`: `enqueue` dan `dequeue` dengan `front` dan `count`.

```
    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;
    }
```

```
  slots [B1005 B1006 B1003 B1004]  front=2 count=4
served in order: B1003 B1004 B1005 B1006
  slots [- - - -]  front=2 count=0
```

Kode 8.7: setiap operasi $O(1)$; menggeser sesudah setiap dequeue akan berbiaya $O(n)$. Di akhir `count` bernilai 0 dan `front` berhenti di 2, yang tidak menjadi soal. Buku: Subbab 8.8.2.

Soal 8.7: Penyelesaian Keliru yang Menggoda

Queue linear Bab 5 dianggap penuh ketika `rear` mencapai sel terakhir:

```
dequeue -> B1001
dequeue -> B1002
waiting: 2 of 4
enqueue B1005 -> refused
```

Dua sel kosong dan permintaan ditolak: “penuh palsu”. Ujinya menanyakan di mana `rear` *berada*; pertanyaannya adalah berapa butir yang *menunggu*.

> **UJI DIRI.** Kapasitas 4: *enqueue* A, B, C; *dequeue* dua kali; *enqueue* D, E, F; *dequeue*. Slot berakhir sebagai `E F - D`, `front` 3, `count` 3; urutannya D, E, F.

Buku: Subbab 8.8.2.

Soal 8.8: Dari Infiks ke Postfiks

SOAL 8.8. `toPostfix` mengubah ekspresi infiks (operand satu huruf atau satu angka, `+ - * /`, tanda kurung) ke postfiks; `evaluate` menghitung ekspresi postfiks berisi angka. Ujilah `3+2*5`, `a+b*c-d`, `8-3-2`, `(1+2)*(7-3)/2`, `9-4/2*3`.

1. Operand → keluaran. **2.** `(` → *push*. **3.** `)` → *pop* ke keluaran sampai `(`, lalu buang.

4. Operator → *pop* dulu setiap operator berderajat **lebih besar atau sama dengan** derajatnya, lalu *push*. **5.** Di akhir, *pop* semuanya.

Derajat: `*` `/` 2, `+` `-` 1, `(` 0. Inilah *shunting-yard algorithm* dari Dijkstra.

"Lebih besar atau sama dengan": `8-3-2` adalah $(8-3)-2=3$*; dengan "lebih besar" saja, keluarannya `832--`, bernilai 7. Buku: Subbab 8.8.3.*

Soal 8.8: `a+b*c-d` dengan Tangan

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-`

*Tabel 8.6: stack menampung operator yang menunggu operand kanannya. Untuk evaluasi, angka di-push; operator mengambil dua bilangan — yang **pertama** diambil adalah operand **kanan** — lalu mem-push hasilnya. Buku: Subbab 8.8.3.*

Soal 8.8: Penyelesaiannya

Kode 8.8 Dari `p808_postfix.cpp`: `evaluate` dengan `std::stack`.

```
    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);
```

```
3+2*5  ->  325*+  =  13
8-3-2  ->  83-2-  =  3
evaluate("3+"): missing operand
```

Kode 8.8: string postfiks yang cacat melempar eksepsi dengan pesan. Java memakai `ArrayDeque`, Python sebuah list; `//` Python membulatkan ke bawah, jadi tandanya dipulihkan dengan tangan. Kedua fungsi $\Theta(L)$. Buku: Subbab 8.8.3.

Soal 8.8: Penyelesaian Keliru yang Menggoda

Menamai bilangan pertama yang diambil `x`, yang kedua `y`, lalu menghitung `x - y`:

```
325*+  =  13
83-2-  =  7
12+73-*2/  =  0
942/3*-  =  -9
```

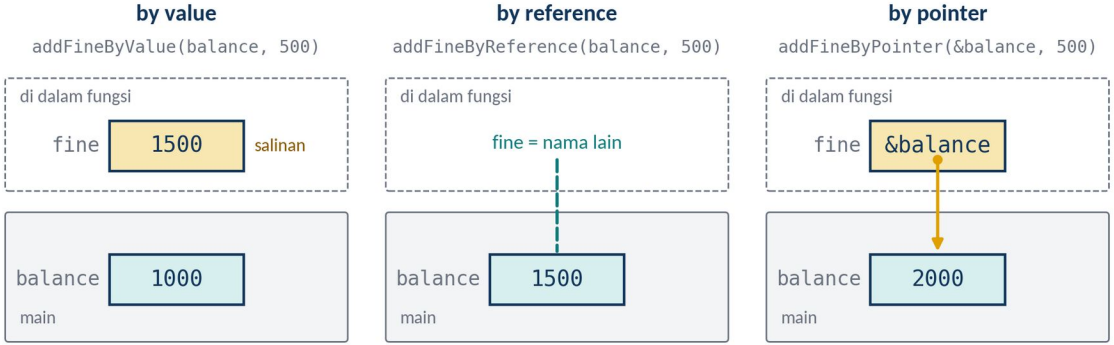
`325*+` tetap memberi 13: penjumlahan dan perkalian tidak peduli urutan. Pengurangan dan pembagian peduli.

> **UJI DIRI.** `(a+b)*(c-d)/e` menjadi `ab+cd-*e/`. `72-3*` bernilai 15; `92/` bernilai 4, dipotong ke arah nol.

Uji dengan `3+2*5` saja akan meloloskan program yang keliru. Buku: Subbab 8.8.3.

Soal 8.9: By Value, by Reference, by Pointer

SOAL 8.9. Seorang anggota menanggung tagihan keterlambatan 1000 dalam `int balance`. Tulislah tiga fungsi yang menambahkan 500, *by value*, *by reference*, dan *by pointer*; cetak `balance` sesudah masing-masing dan apakah parameternya kotak yang sama. Tambahkan dua `swap` dan `larger` yang mengembalikan alamat.



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.

Gambar 8.8: by value, salinannya lenyap saat kembali; by reference, `fine` adalah `balance` (dan tidak ada yang memperingatkan di tempat pemanggilan); by pointer, `&balance` adalah peringatan yang terlihat, dan `*fine` mengikutinya. Buku: Subbab 8.9.1.

Soal 8.9: Penyelesaian C++

Kode 8.9 Dari `p809_three_ways.cpp`: *by value* dan *by reference*, masing-masing memeriksa kotaknya sendiri.

```
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;
}
```

```
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
```

Kode 8.9: by pointer, `*fine += amount` memberi 2000. `*larger(&x, &y) += 10`: pemanggilannya adalah alamat, dan `*` membukanya. Buku: Subbab 8.9.1.

Soal 8.9: Java dan Python

```
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
same object? True
```

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

Tabel 8.7: Java mengirim segalanya by value, tetapi nilai variabel objek adalah referensi; parameter Python adalah nama baru untuk objek yang sama. Buku: Subbab 8.9.1.

Soal 8.9: Penyelesaian Keliru yang Menggoda

```
int *makeFine(int amount) {
    int fine = amount;      // kotak lokal: lenyap ketika fungsi selesai
    return &fine;           // KELIRU: pemanggil menerima alamat kotak yang sudah mati
}
```

```
wrong/p809_wrong.cpp:7:12: warning: address of local variable 'fine' returned [-Wreturn-local-addr]
```

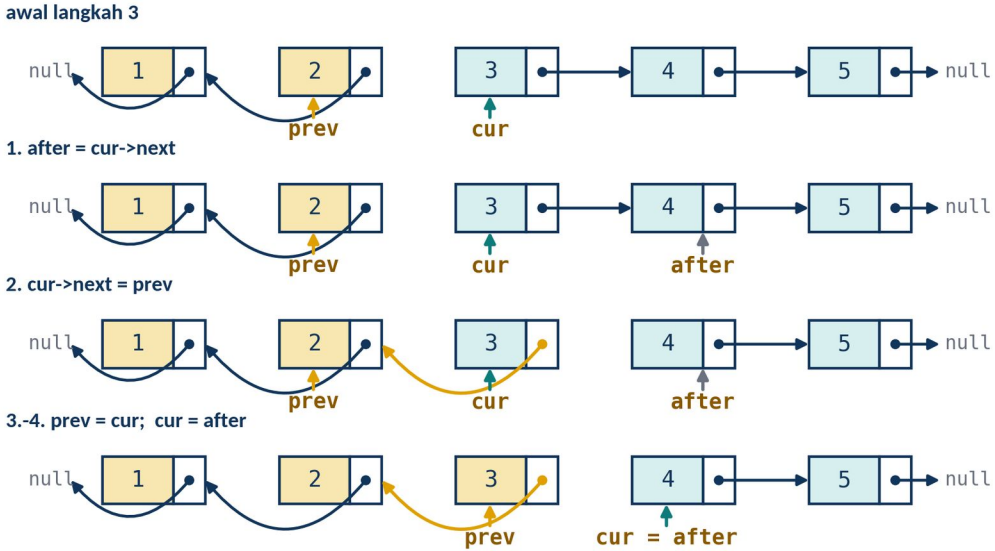
Pemanggil memegang *dangling pointer*; dengan optimasi GCC 13 bahkan mengembalikan *null*. Kembalikan nilainya, atau pakai `new` dan nyatakan siapa yang menghapusnya.

> **UJI DIRI.** `int a = 5, b = 7; int *p = &a; int &r = b; *p = *p + r; r = 1; p = &b; *p += a;` berakhir dengan `a` = 12 dan `b` = 13.

Peringatan seperti inilah alasan setiap program dikompilasi dengan `-Wall -Wextra`. Buku: Subbab 8.9.1.

Soal 8.10: Membalik List di Tempat

SOAL 8.10. Tulislah `reverse`, yang membalik singly linked list *di tempat* — tanpa membuat atau menghapus simpul — dan mengembalikan simpul pertama yang baru. Telusurilah pada 1, 2, 3, 4, 5; uji list satu simpul dan list kosong.



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.

Gambar 8.9. Buku: Subbab 8.10.1.

Soal 8.10: Penyelesaiannya

Kode 8.10 Dari `p810_reverse_list.cpp`: pembalikan di tempat dengan tiga pointer (baris penelusuran dihilangkan).

```
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
    }
    return prev;
}
```

```
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]
```

Kode 8.10: list satu simpul dan list kosong tidak memerlukan kode khusus. Buku: Subbab 8.10.1.

Soal 8.10: Penyelesaian Keliru yang Menggoda

```
  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)
```

```
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).
```

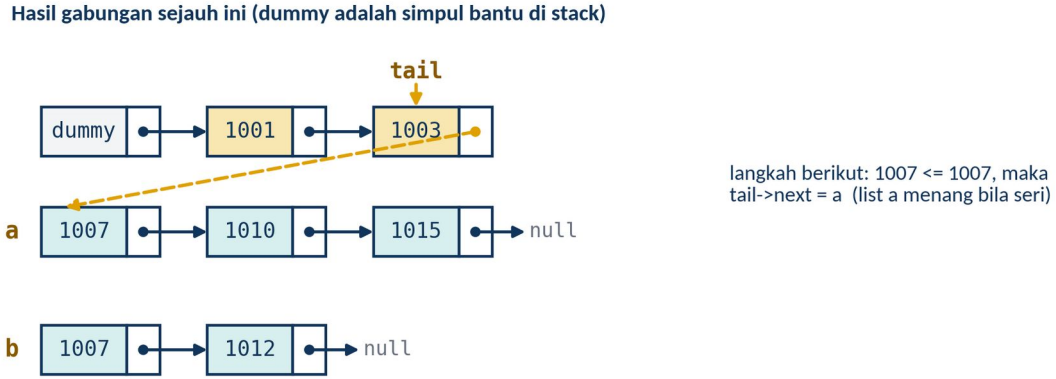
Sesudah `cur->next = prev`, tidak ada yang menunjuk simpul 2; `cur = cur->next` justru *kembali* ke `nullptr`. Gelung berakhir sesudah satu langkah, dan LeakSanitizer melaporkan simpul 2 hilang secara langsung serta simpul 3–5 secara tidak langsung.

> **UJI DIRI.** Untuk 8 → 9: `reverse` mengembalikan 9, yang `next`-nya 8, yang `next`-nya `nullptr`. Gelung berjalan dua kali, jadi setiap penugasan dijalankan dua kali.

Buku: Subbab 8.10.1.

Soal 8.11: Menggabungkan Dua List Terurut

SOAL 8.11. Cabang A memiliki 1003, 1007, 1010, 1015 dan cabang B memiliki 1001, 1007, 1012, keduanya naik. Tulislah `merge(a, b)` dengan menautkan ulang simpul, hitung perbandingannya, dan uji list kosong serta list yang tidak bertumpang tindih.



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`.

Gambar 8.10: kode terkecil yang tersisa selalu berada di salah satu dari dua ujung depan. ***Simpul dummy*** *di stack memori menghapus kasus "simpul pertama":* `tail` *mulai di sana, setiap penempelan adalah* `tail->next = node; tail = node;`*, dan jawabannya* `dummy.next`*. Buku: Subbab 8.10.2.*

Soal 8.11: Penyelesaiannya

Kode 8.11 Dari `p811_merge_sorted.cpp`: penggabungan dengan simpul *dummy*.

```
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;
}
```

```
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)
```

Kode 8.11: 6, 0, 0, dan 3 perbandingan. `<=` membuat penggabungannya stabil. Buku: Subbab 8.10.2.

Soal 8.11: Batasnya, dan Penyelesaian Keliru

$$\text{perbandingan} \le m+n-1\,,\ \text{waktu} = \Theta(m+n)\,,\ \text{memori tambahan} = O(1)\,. \tag{8.6}$$

```
A + B       : 1001 1003 1007 1007 1010 1012  (6 comparisons)
lost        : 1015  (6 comparisons)
```

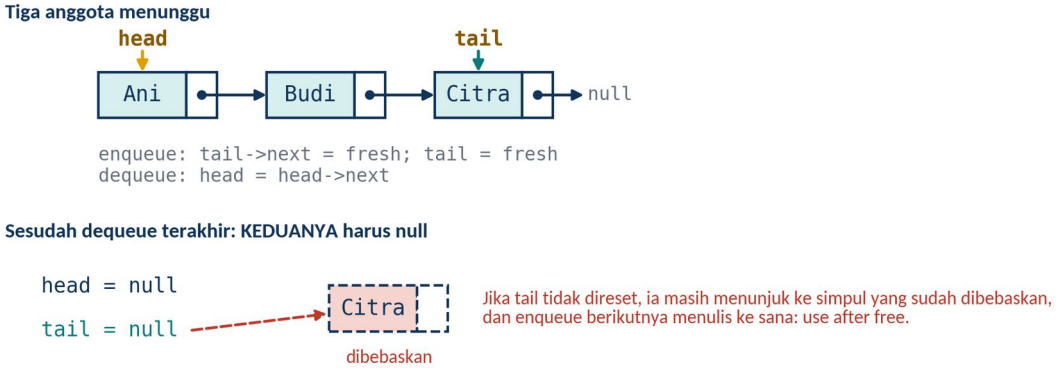
Keliru: melupakan baris 27. Gelung berhenti ketika *salah satu* list kosong; list lainnya masih memiliki simpul.

> **UJI DIRI.** `a` = 2, 4, 6, 8 dan `b` = 5: 2 lawan 5, 4 lawan 5, 6 lawan 5; `b` kosong sesudah 3 perbandingan dan 6, 8 ditempelkan sekaligus. Batasnya $4+1-1=4$.

Persamaan (8.6): setiap perbandingan menempatkan satu simpul. Buku: Subbab 8.10.2.

Soal 8.12: Queue dari Simpul Bertaut

SOAL 8.12. Implementasikan queue berisi nama dengan `head` (dilayani) dan `tail` (bergabung), kedua operasi $O(1)$. *Enqueue* Ani, Budi, Citra; layani ketiganya; *enqueue* Dewi dan Eko; layani satu. `dequeue` pada queue kosong dilaporkan, bukan membuat program terhenti.



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.

*Gambar 8.11: invariannya — `head` pada simpul tertua, `tail` pada yang terbaru, keduanya `nullptr` ketika kosong, `count` simpul. Ketika simpul terakhir pergi, **`tail` juga harus menjadi `nullptr`**. Buku: Subbab 8.10.3.*

Soal 8.12: Penyelesaiannya

Kode 8.12 Dari `p812_linked_queue.cpp`: `dequeue`, dengan baris yang mudah terlupa.

```
    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;
    }
```

```
  queue:  (size 0, head null, tail null)
  queue: Dewi Eko  (size 2, head Dewi, tail Eko)
served Dewi
caught: dequeue from an empty queue
```

Kode 8.12: `enqueue` *memasang kedua pointer pada queue kosong, selain itu* `tail->next = fresh; tail = fresh;`*. Destruktor mendequeue sisanya; penyalinan dilarang. Buku: Subbab 8.10.3.*

Soal 8.12: Penyelesaian Keliru yang Menggoda

Keliru: hapus baris-baris yang menyetel ulang `tail`, dan uji kekosongan lewat `tail`. Sesudah queue kosong, `tail->next = fresh` menulis ke simpul Citra yang sudah dibebaskan.

```
==...==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...)
```

UJI DIRI. *Enqueue* X; *dequeue*; *enqueue* Y; *enqueue* Z; *dequeue*: `head` dan `tail` sama-sama berakhir di Z, `count` 1; sesudah *dequeue* pertama keduanya `nullptr`.

Operasi tulis 8 bita, 32 bita ke dalam blok 40 bita: tepat medan `next` sesudah `std::string` 32 bita — `tail->next` yang basi. Buku: Subbab 8.10.3.

Soal-Soal Dibandingkan, dan Tiga Pola Biaya

Sejumlah tetap perubahan pointer atau indeks bernilai $O(1)$: setiap operasi stack dan queue, larik maupun simpul.

Satu penelusuran bernilai $\Theta(n)$: pembalikan, penggabungan, pencetakan, pencocokan kurung, konversi ke postfiks.

Penelusuran di dalam penelusuran bernilai $\Theta(n^2)$ menurut jumlahan Persamaan (8.3) dan Persamaan (8.4): penyisipan terurut, pembuangan nilai ganda, penambahan di belakang tanpa *tail*.

Bila kelompok ketiga terlalu lambat, tanyakan struktur mana yang mengubah penelusuran dalam menjadi satu langkah: pointer *tail* (Bab 9), *hash set* (Bab 15), *heap* (Bab 14).

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	$\Theta(\mathrm{MAX})$	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	$\Theta(\mathrm{CAPACITY})$	*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

Tabel 8.8 (Lanjutan) *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
8.12	list dengan head dan tail	$O(1)$ per operasi	satu simpul per butir	*enqueue* sesudah queue sempat kosong

Tabel 8.8: bacalah kolom terakhir — masukan yang mematahkan penyelesaian keliru jarang sekali berupa contoh dalam soal. Buku: Subbab 8.11.

Profitina dalam Praktik

Pekerjaan yang menunggu giliran adalah queue: pemindaian ulang terjadwal dan umpan *real-time* per penonton, pertama masuk pertama keluar; queue sirkular berkapasitas tetap membatasi memori yang dapat dihabiskan satu pembaca lambat.

Mengikuti referensi adalah menyusuri list: balasan yang dalam kembali ke utasnya, seperti `prev` pada Soal 8.2 — dengan unggahan terhapus (`nullptr`) dan lingkaran (Persamaan (8.2)) yang harus dijaga.

Simpan yang Anda perlukan sebelum mengubah apa pun: penyusun unggahan menyimpan teks sebelum mengosongkan kotak, dan memulihkannya bila pengiriman gagal.

Servernya ditulis dalam Python, jadi tidak ada yang membebaskan simpul dengan tangan; sepupu bug-bug ini — referensi "butir terakhir" yang basi — ditangkap oleh uji yang membidik kasus tepi. Buku: Subbab 8.12.

Kesalahan yang Sering Terjadi

Menimpa satu-satunya pointer ke sesuatu yang masih diperlukan; **membaca dari simpul sesudah meninggalkannya.**

Membebaskan sebelum melepas tautan; menyisakan pointer ke memori yang sudah dibebaskan (`tail`, alamat variabel lokal yang dikembalikan).

Mencampur dua konvensi indeks; memindahkan pointer pemindai sesudah penghapusan; melupakan sisa penggabungan.

`x % 2 == 1` **untuk ganjil; mengambil operand dengan urutan salah; mencacah alih-alih mengingat.**

***Buffer* berukuran tetap; menyimpan penanda akhir sebagai data; hanya menguji contoh.**

Buku: Subbab 8.13.

Ringkasan: Lima Hal yang Perlu Diingat

1. Tujuh langkah: nyatakan ulang dengan contoh, pilih struktur, gambar, tulis invarian, daftar kasus tepi, urutkan perubahan pointer, uji di bawah pemeriksa memori.
2. Sebelum mengubah tautan, pertahankan jalan menuju semua yang masih diperlukan.
3. Stack: Persamaan (8.1), tanda kurung, *shunting-yard* dengan operand kanan diambil lebih dulu. Queue: Persamaan (8.5) dengan pencacah; queue berantai menyetel ulang `tail`.
4. *By value* menyalin, *by reference* memberi nama lain, *by pointer* mengirim alamat; Java dan Python mengembalikan simpul pertama yang baru.
5. List: berdiri pada pendahulu, lepas tautan lalu bebaskan, tiga pointer untuk membalik, simpul *dummy* untuk menggabungkan (Persamaan (8.6)).

Buku: Subbab 8.14.

Pertemuan Berikutnya

Bab 9 — Double Linked List dan Linked List Sirkular. Setiap simpul juga mengingat simpul sebelumnya: penghapusan belakang $O(1)$, list dengan *tail*, versi melingkar, dan *deque*.

Soal 8.2 adalah cuplikannya: dua panah per simpul, dan keduanya dipasang pada setiap penyisipan.

Sebelum itu: kerjakan Soal Latihan 8.3, 8.5, 8.7, dan 8.11, lalu jalankan sendiri `wrong/p812_wrong.cpp` di bawah AddressSanitizer.

Buku: Subbab 8.15.

Kuliah 9 — Double Linked List dan Linked List Sirkular

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 9

Dosen: Irfan Subakti — yifana@gmail.com

Hari ini setiap simpul mendapat panah kedua yang menunjuk ke belakang, dan dua batasan Bab 7 pun lenyap. Buku: Bab 9, Subbab 9.1–9.15.

Cakupan Pertemuan Ini

1. **Dua panah per simpul** — `prev` dan `next`, invarian dua arah, dan harga pointer kedua dalam memori.
2. **List lewat `head`, lalu lewat `head` dan `tail`** — setiap operasi sehari-hari, digambar sebelum dan sesudah, lengkap dengan ongkosnya.
3. **Simpul kepala, lingkaran, *sentinel*** — tiga perubahan representasi, hingga tidak ada lagi kasus khusus yang tersisa.
4. **Dua aplikasi** — *deque* untuk meja pengembalian Pustaka Ceria, dan panel “baru dipinjam” (pola LRU).
5. **Ongkos yang diukur; keamanan memori; pustaka standar** — *single linked* lawan *double linked* lawan larik, Valgrind dan AddressSanitizer, `std::list`, `LinkedList`, `collections.deque`.

Setiap operasi digambar sebelum dan sesudahnya, setiap ongkos diturunkan, dan setiap program lengkap serta diperiksa memorinya. Buku: Subbab 9.1.

Titik Tolak: Dua Batasan

Berjalan mundur. Simpul *single linked* hanya mengenal simpul sesudahnya; untuk mundur kita harus mulai lagi dari `head` dan berjalan.

Menghapus di belakang. Subbab 7.8 membuat penyisipan di belakang $O(1)$ dengan `tail`, tetapi penghapusan di belakang tetap $\Theta(n)$: simpul *sebelum* yang terakhir harus diubah, dan tidak ada yang tahu letaknya.

Batasan yang sama bersembunyi dalam penghapusan berdasarkan kunci: Subbab 7.6.5 memerlukan pointer pengekor `prev` yang berjalan satu langkah di belakang `cur`.

Rencananya: tambahkan satu medan, `prev`, ke setiap simpul. Berjalan ke dua arah, menghapus simpul terakhir dalam $O(1)$, menghapus simpul *mana pun* yang dipegang dalam $O(1)$.

Soal 8.2 di Bab 8 sudah menyusuri list dua arah maju dan mundur; hari ini gagasan itu kita kembangkan sepenuhnya, beserta harganya: kerja pointer dua kali lipat dan 8 bita lebih per simpul. Buku: Subbab 9.1.

Double Linked List

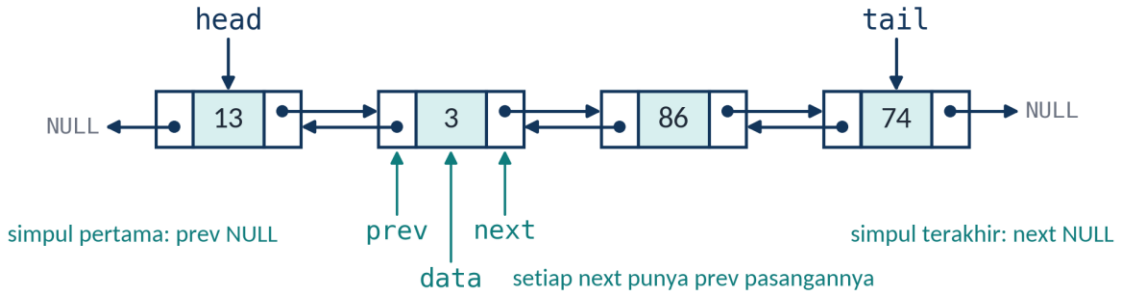
DEFINISI. *Double linked list* adalah barisan simpul yang setiap simpulnya menyimpan satu elemen (*data*-nya) dan dua pointer: `next`, ke simpul sesudahnya, dan `prev`, ke simpul sebelumnya. Pada *double linked list non-circular* (*DLLNC*), `prev` milik simpul pertama dan `next` milik simpul terakhir bernilai null. List dicapai lewat `head`, sering kali bersama `tail`; list kosong tidak punya simpul dan pointer luarnya null.

Bayangkan kereta api: setiap gerbong tersambung ke gerbong di depan dan di belakangnya, sambungan dua tetangga selalu sepakat, dan kondektur dapat berjalan sampai ujung lalu kembali.

Double: dua medan pointer per simpul. *Linked list*: simpul dihubungkan oleh pointer, bukan diletakkan berdampingan. *Non-circular*: kedua rantai berakhir pada null.

Melepas satu gerbong mengubah dua sambungan, memasang satu gerbong mengubah empat: persis jumlah pointer pada penghapusan dan penyisipan. Buku: Subbab 9.2.1–9.2.2.

Anatomi Double Linked List



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.

*Pada **Gambar 9.1** setiap simpul punya tiga kompartemen: `prev`, data, `next`. Panah `next` keluar dari separuh atas dan mengarah ke kanan; panah `prev` keluar dari separuh bawah dan mengarah ke kiri. Di antara dua tetangga selalu ada dua panah. Buku: Subbab 9.2.2 dan Gambar 9.1.*

Invarian Dua Arah

DEFINISI. *Invarian dua arah* dari *double linked list* non-sirkular: untuk setiap simpul p berlaku Persamaan (9.1), ditambah $head.prev = \text{NIL}$ dan $tail.next = \text{NIL}$ bila list tidak kosong.

$$p.next \neq \text{NIL} \;\Rightarrow\; p.next.prev = p \qquad \text{dan} \qquad p.prev \neq \text{NIL} \;\Rightarrow\; p.prev.next = p \tag{9.1}$$

Dengan kata lain: melangkah maju lalu mundur, dan Anda kembali ke tempat semula; begitu pula mundur lalu maju.

Setiap operasi dalam bab ini *mengandalkannya* tanpa memeriksanya: `backDeletion` memercayai `tail->prev`; `deleteNode(p)` memercayai bahwa `p->prev->next` adalah `p`.

Bila invarian itu rusak di satu tempat, operasi berikutnya diam-diam merusak list lebih jauh. Subbab 9.4.5 menulis pemeriksanya; Subbab 9.11 memperlihatkan kerusakannya di memori. Buku: Subbab 9.2.2 dan Persamaan (9.1).

Mendeklarasikan Simpul

Simpul Bab 7 mendapat tepat satu medan tambahan; simpul baru dimulai dengan kedua pointer null (**Kode 9.1**).

Kode 9.1 Awal `dll_node.cpp`: tipe simpul yang merujuk dirinya sendiri, dan `makeNode`.

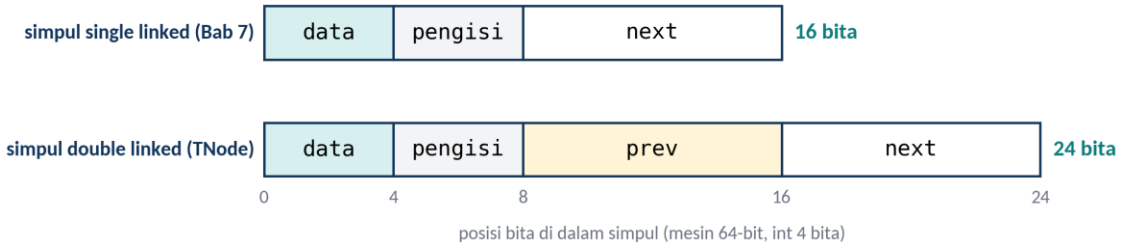
```
// dll_node.cpp -- a doubly linked node: declared, created, and linked by hand.
#include <cstdio>

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;
}
```

`prev` dan `next` sama-sama bertipe "pointer ke `struct MyNode`". Menurut kebiasaan `prev` ditulis lebih dulu, sesuai urutan kiri-ke-kanan pada diagram. Program itu lalu menaut tiga simpul dengan tangan, setiap tautan maju dengan kembaran mundurnya. Buku: Subbab 9.2.3.

Harga Pointer Kedua



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.

***Gambar 9.2**: `int` menempati bita 0–3, padding 4–7 menjaga setiap pointer di alamat kelipatan 8, `prev` di 8–15 dan `next` di 16–23. Program mencetak `sizeof(TNode) = 24 bytes`. Buku: Subbab 9.2.4 dan Gambar 9.2.*

Rumus Memori

Dengan data d bita, pointer p bita, dan tambahan alokator a bita per simpul, Persamaan (9.2) membandingkan kedua list:

$$M_{\text{DLL}} = n\,(d + 2p + a) \quad \text{bita, dibandingkan} \quad M_{\text{SLL}} = n\,(d + p + a). \tag{9.2}$$

Selisihnya tepat $n\,p$: satu pointer per elemen, 8 MB untuk sejuta elemen.

Untuk data `int` pajaknya berat (24 bita simpul untuk 4 bita data); untuk rekaman buku 200 bita hanya 4 persen.

Aturan praktis Bab 7 berlaku lebih keras lagi: double linked list berelemen kecil itu mahal, berelemen rekaman besar tidak. Buku: Subbab 9.2.4 dan Persamaan (9.2).

`frontInsertion`: Tiga Penugasan

Pseudokode 9.1 mengubah tiga pointer, padahal *single linked list* hanya dua: simpul pertama yang lama kini harus menunjuk *balik* ke simpul baru.

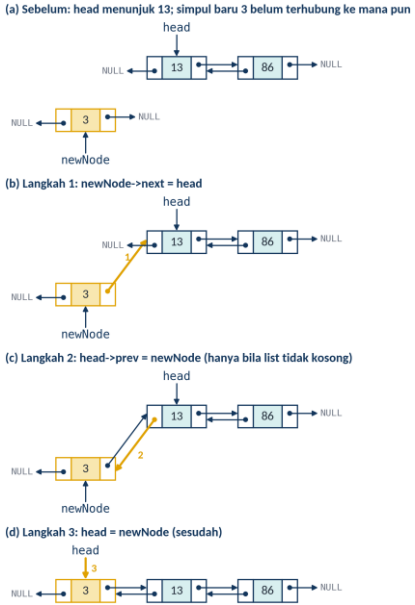
Pseudokode 9.1 `FRONT-INSERTION` pada *double linked list* yang dicapai lewat `head`: tiga penugasan pointer, dan yang tengah hanya bila list tidak kosong.

```
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
```

Baris 3 adalah *kasus khusus* pertama dalam bab ini: list kosong tidak punya simpul pertama lama yang perlu diperbarui.

Langkah 3 harus terakhir, sebab langkah 1 dan 2 membaca `head` untuk menemukan simpul pertama yang lama; langkah 1 dan 2 boleh ditukar. Ongkosnya $O(1)$. Fungsi menerima `head` secara referensi, `TNode *&head`, bila mungkin mengubahnya. Buku: Subbab 9.3.2.

Penyisipan di Depan, Langkah demi Langkah



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.

*Pada **Gambar 9.3**, sesudah langkah 2 simpul baru sudah tertaut benar ke 13 dalam dua arah, tetapi list belum mengetahuinya sampai langkah 3 memindahkan `head`. Buku: Subbab 9.3.2 dan Gambar 9.3.*

`backInsertion` Hanya dengan `head`

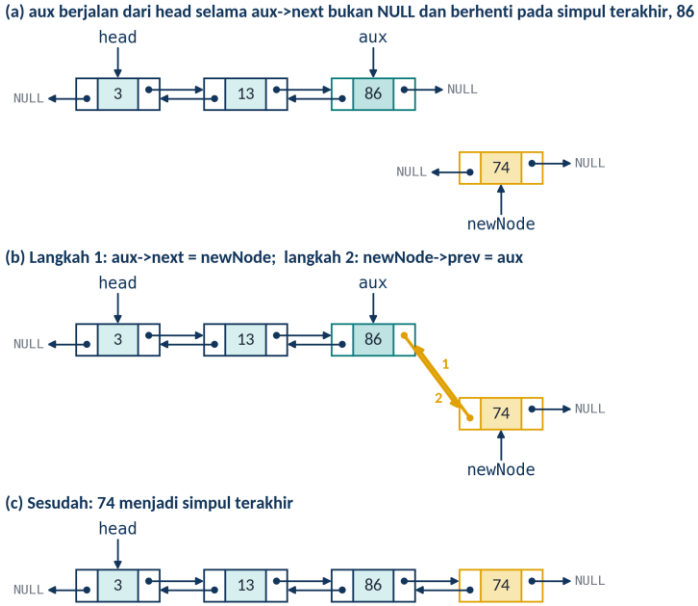
Pseudokode 9.2: berjalan ke simpul terakhir persis seperti Subbab 7.5.3, lalu menaut ke dua arah.

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
```

Baris 9 adalah satu-satunya baris baru: tanpanya penyusuran maju menampilkan 74, tetapi penyusuran mundur yang dimulai dari 74 langsung berhenti. Ongkosnya $\Theta(n)$ untuk perjalanan itu: hanya pointer ke belakang yang dapat memendekkannya. Buku: Subbab 9.3.3.

Penyisipan di Belakang, Digambar



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.

Gambar 9.4 *menelusuri penyisipan 74 ke dalam 3, 13, 86; panel (a) adalah perjalanan Bab 7. Buku: Subbab 9.3.3 dan Gambar 9.4.*

Menyusuri Mundur

Pseudokode 9.3: fase 1 berjalan maju ke simpul terakhir, fase 2 mengikuti `prev` sampai ke depan.

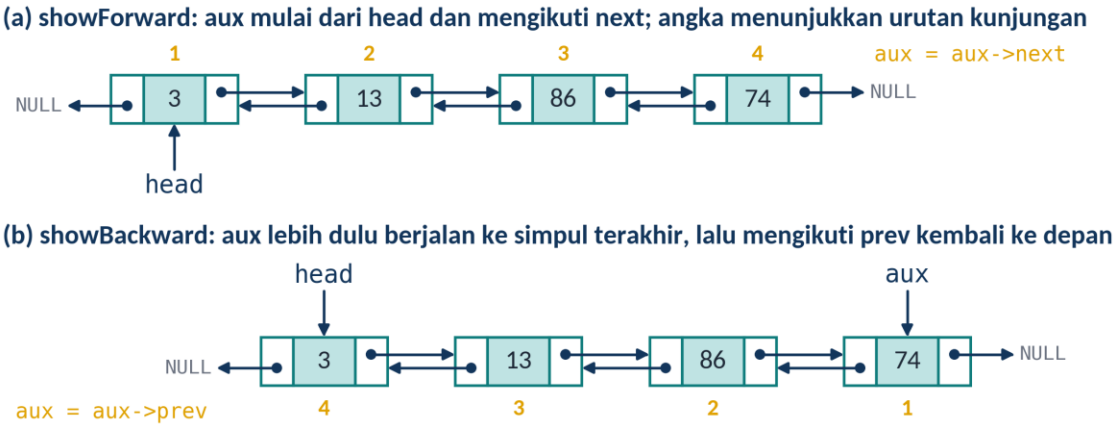
Pseudokode 9.3 `SHOW-BACKWARD` dengan pointer *head* saja: berjalan ke simpul terakhir, lalu mengunjungi setiap simpul dari yang terakhir sampai yang pertama.

```
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
```

Fase 1 menguji `aux.next` (ia harus *berhenti pada* simpul terakhir); fase 2 menguji `aux` (ia harus *mengunjungi* setiap simpul).

Sekitar $2n$ langkah dan tanpa memori tambahan. Single linked list memerlukan waktu $\Theta(n^2)$ atau stack $\Theta(n)$ untuk mencetak mundur. Dengan `tail`, fase 1 hilang. Buku: Subbab 9.3.4.

Kedua Penyusuran, Digambar



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.

*Angka-angka pada **Gambar 9.5** memberi urutan kunjungan di setiap arah. Buku: Subbab 9.3.4 dan Gambar 9.5.*

`frontDeletion`: Kosongkan `prev` yang Baru

Pseudokode 9.4 adalah simpan, pindahkan, hapus dari Subbab 7.5.5, ditambah satu penugasan.

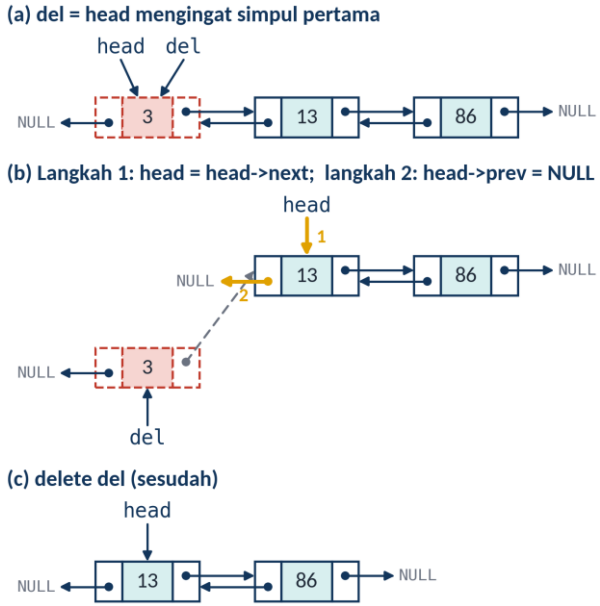
Pseudokode 9.4 `FRONT-DELETION`: ingat simpul pertama, pindahkan `head`, kosongkan `prev` milik simpul pertama baru, lepaskan simpul lama.

```
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
```

PERANGKAP. Baris 6 adalah baris yang dilupakan pemrogram yang terbiasa dengan *single linked list*. List masih tercetak benar ke depan, tetapi `prev` milik 13 menyimpan alamat simpul yang sudah dilepas: pointer menggantung *di dalam* list.

Baris 5 lagi-lagi kasus khusus: list bersimpul satu tidak punya simpul pertama baru yang perlu diperbarui. Ongkosnya $O(1)$. Buku: Subbab 9.3.5.

Penghapusan di Depan, Digambar



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.

***Gambar 9.6**: `del` mengingat 3, `head` pindah ke 13 dan `prev` milik 13 menjadi NULL, lalu 3 dilepas. Buku: Subbab 9.3.5 dan Gambar 9.6.*

`backDeletion` Hanya dengan `head`

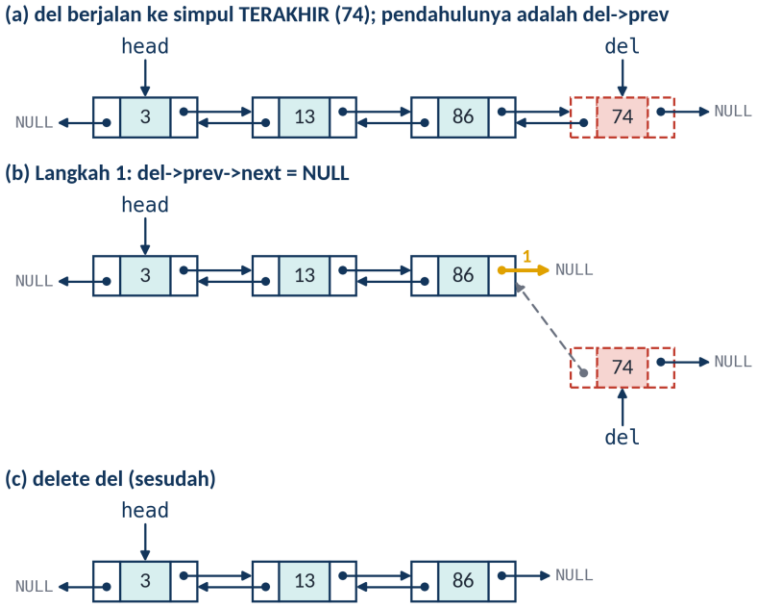
Pseudokode 9.5: penolong berjalan sampai simpul terakhir itu sendiri; pendahulunya cukup `del.prev`. Tidak ada lagi `aux.next.next` yang canggung.

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
```

Kasus bersimpul satu dikenali dari `del.prev = NIL` sesudah perjalanan. Kodenya lebih sederhana, tetapi perjalanannya tetap $\Theta(n)$: satu-satunya pintu masuk ada di depan. Buku: Subbab 9.3.6.

Penghapusan di Belakang, Digambar



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.

***Gambar 9.7**: `del` berjalan ke 74, pendahulunya 86 berhenti menunjuknya, dan simpul itu dihapus. Buku: Subbab 9.3.6 dan Gambar 9.7.*

List yang Sama dalam Python

Kode 9.2: tidak ada `delete`, namun `front_deletion` tetap mengosongkan `prev` milik simpul pertama yang baru.

Kode 9.2 `front_deletion` dari `dll_head.py`.

```
    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()")
```

Referensi tidak pernah dapat menunjuk memori yang sudah dilepas, tetapi tautan mundur yang basi merusak invarian *dan* membuat simpul lama tetap hidup: kesalahan logika sekaligus kebocoran memori.

`DllHead.java` melakukan hal yang sama dengan `head.prev = null`; keduanya mencetak keluaran yang persis sama dengan program C++. Buku: Subbab 9.3.9.

Head dan Tail: Hapus di Belakang dalam $O(1)$

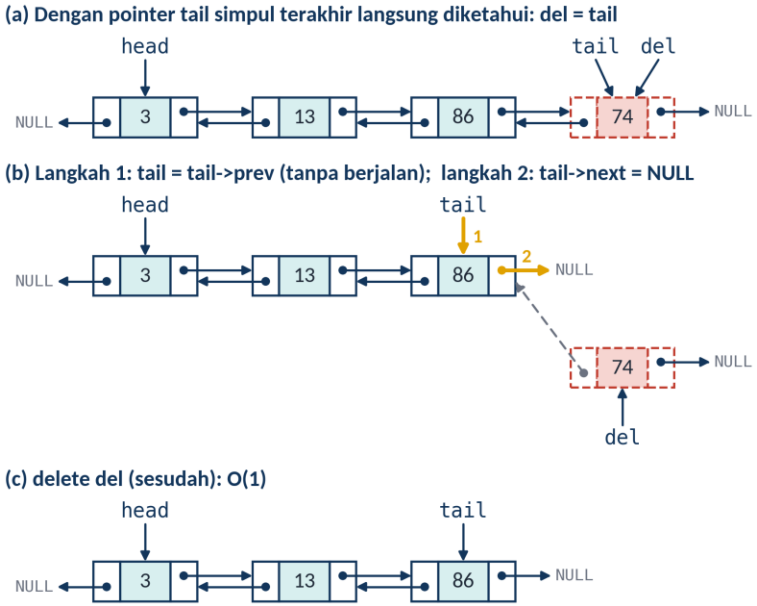
Pseudokode 9.6: simpul sebelum yang terakhir adalah `tail.prev`, jadi sama sekali tidak ada perjalanan.

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. Penghapusan di belakang adalah cermin penghapusan di depan: tukar `head`/`tail` dan `next`/`prev`. `head` dan `tail` kini sama-sama null atau sama-sama tidak null. Buku: Subbab 9.4.1.

Hapus di Belakang dengan Tail, Digambar



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.

***Gambar 9.8**: `tail` mundur satu langkah lewat `prev`, 86 menjadi simpul terakhir, 74 dilepas, dalam jumlah langkah tetap berapa pun panjangnya. Buku: Subbab 9.4.1 dan Gambar 9.8.*

`insertAfter`: Empat Panah

Di antara `p`, simpul baru `n`, dan penerus s ada empat panah: $p \to n$, $n \to s$ ke depan; $s \to n$, $n \to p$ ke belakang (**Pseudokode 9.7**).

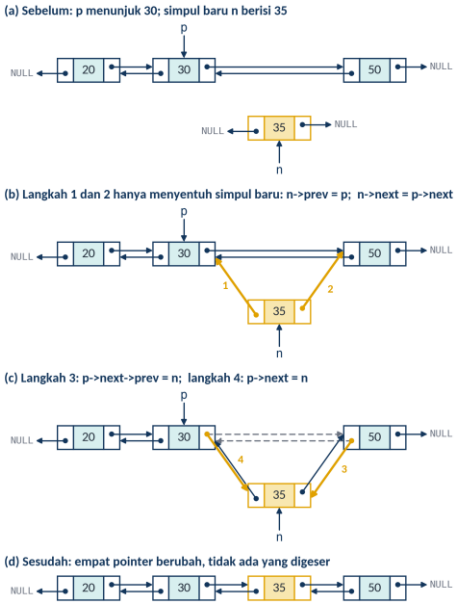
Pseudokode 9.7 `INSERT-AFTER`: menautkan simpul baru sesudah simpul `p` dengan empat penugasan pointer, dalam waktu $O(1)$; langkah 4 harus sesudah langkah 2 dan 3.

```
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
```

> **ATURAN AMAN.** Pasang dulu kedua pointer milik simpul baru, lalu perbarui tetangganya, dan perbarui tautan yang Anda pakai untuk menemukan tetangga itu *paling akhir*.

Baris 3 dan 7 membaca `p.next`; baris 8 menimpanya. Bila baris 8 dijalankan lebih dulu, simpul baru menunjuk dirinya sendiri dan 50 hilang. `insertBefore` adalah cerminnya, mustahil dalam $O(1)$ pada single linked list. Buku: Subbab 9.4.2.

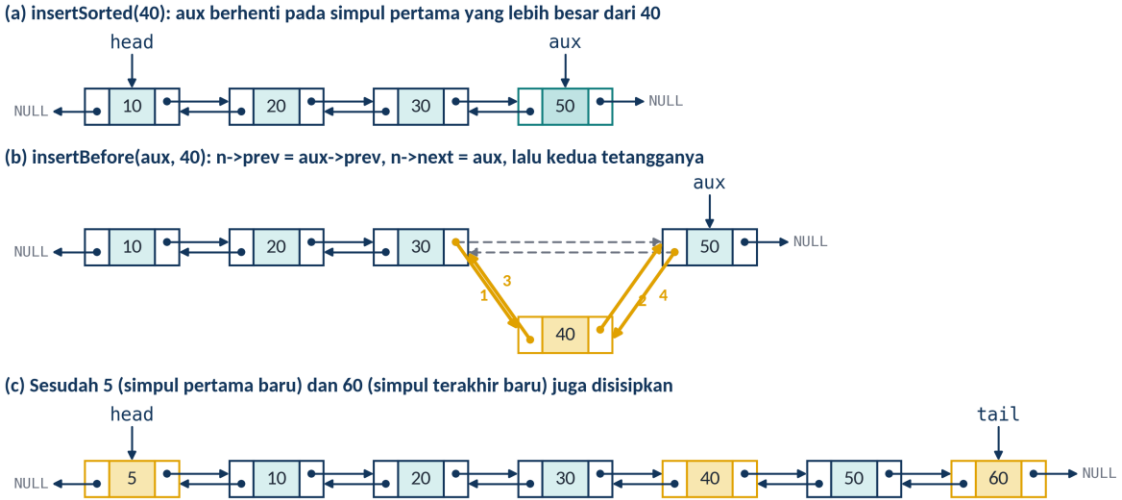
Sisip Sesudah, Langkah demi Langkah



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.

***Gambar 9.9**: langkah 1 dan 2 hanya menyentuh simpul baru; langkah 3 dan 4 mengarahkan ulang tetangganya; panah putus-putus abu-abu adalah tautan yang digantikan. Buku: Subbab 9.4.2 dan Gambar 9.9.*

Penyisipan Terurut, Digambar



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.

Gambar 9.10*: berjalan ke simpul pertama yang tidak lebih kecil dari x lalu menyisip sebelum-nya; untuk 40* `aux` *berhenti pada 50, lalu 5 masuk di depan dan 60 di belakang. Tetap* $\Theta(n)$*, tetapi lebih sederhana. Buku: Subbab 9.4.3 dan Gambar 9.10.*

`deleteNode`: Dua Penugasan

Pseudokode 9.8: pendahulunya `p.prev`, penerusnya `p.next`; buat keduanya melompati `p`.

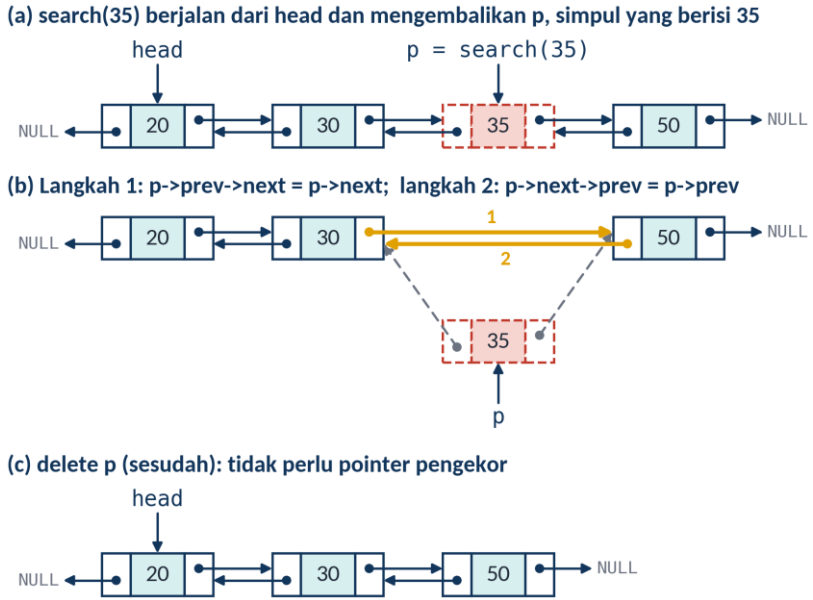
Pseudokode 9.8 `DELETE-NODE`: melepas simpul `p` dalam waktu $O(1)$; tidak perlu pencarian maupun pointer pengekor.

```
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
```

`frontDeletion` adalah `DELETE-NODE(L, L.head)`; `backDeletion` adalah `DELETE-NODE(L, L.tail)`; `deleteKey` adalah pencarian yang diikuti `DELETE-NODE`.

Bila pointer datang dari hash map, iterator, atau simpul itu sendiri, single linked list harus mencari pendahulunya, $\Theta(n)$; double linked list tidak, $O(1)$. Buku: Subbab 9.4.4.

Hapus Berdasarkan Kunci, Digambar



Gambar 9.11 `deleteKey(35)`: (a) `search` mengembalikan `p`, (b) pendahulu dan penerus ditautkan satu sama lain, (c) `p` dihapus.

*Pada **Gambar 9.11**, pointer milik `p` sendiri dibaca tetapi tidak pernah diubah: simpul itu ditinggalkan tetangganya lalu dilepas. Buku: Subbab 9.4.4 dan Gambar 9.11.*

`deleteNode` dalam C++

Kode 9.3 `deleteNode`, kedua penghapusan di ujung, dan `deleteKey` dari `dll_ops.cpp`.

```
// 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 9.3: satu prosedur dengan dua uji mencakup setiap penghapusan dalam list.

`checkLinks` dalam program yang sama menyusuri list sekali dan memverifikasi Persamaan (9.1); memanggilnya sesudah setiap operasi dalam pengujian langsung menangkap penugasan mundur yang terlupa. Buku: Subbab 9.4.4–9.4.5.

Contoh Terurai: Menelusuri `dll_ops.cpp`

```
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
deleteKey(35):            1 5 10 20 30 40 50 60
front and back deletion:  5 10 20 30 40 50
```

`insertBefore(10, 5)` mengubah `head`; penyisipan terurut mengenai bagian tengah (40), depan (1), dan belakang (60).

Baris backward adalah baris maju yang dibalik, dan `checkLinks` setuju; ketiga penghapusan membuang simpul tengah, depan, dan belakang, masing-masing $O(1)$ begitu simpulnya diketahui. Buku: Subbab 9.4.6.

Ongkos dengan Head dan Tail

Tabel 9.1 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 9.8**)
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

***Tabel 9.1**: setiap operasi pada posisi yang diketahui menjadi $O(1)$, dengan harga kerja pointer dua kali lipat. Buku: Subbab 9.4.6.*

Simpul Kepala Menghapus Kasus di Depan

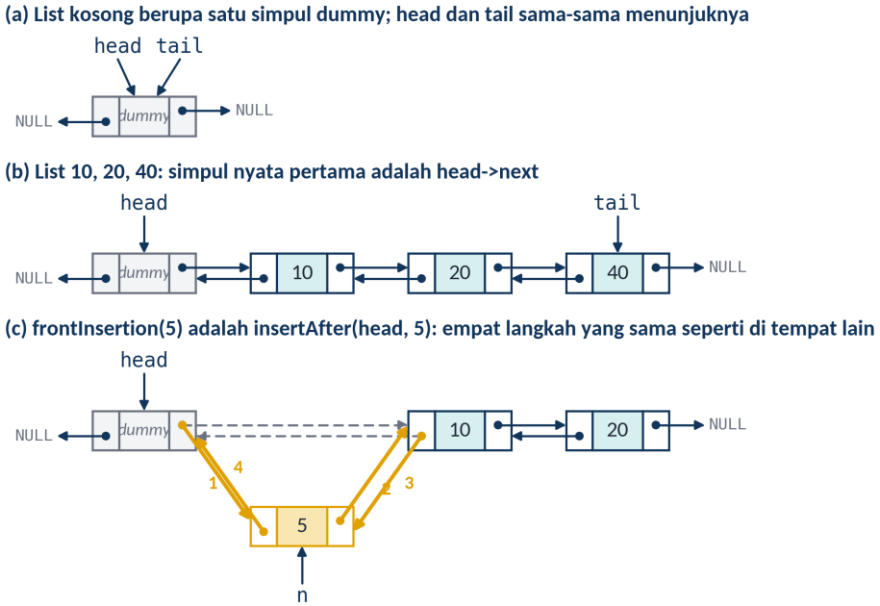
Semua kasus khusus sejauh ini punya satu penyebab: simpul di *ujung* punya pointer null di satu sisi, sehingga operasi harus menulis ke `head` atau `tail` sebagai gantinya.

> **DEFINISI.** *Double linked list dengan simpul kepala* menyimpan satu simpul tanpa data secara permanen di depan; `head` selalu menunjuknya, simpul nyata pertama adalah `head->next`, dan list kosong hanya terdiri atas simpul kepala.

Penyisipan di depan menjadi `insertAfter(head, x)`; `deleteNode(p)` tidak pernah menguji `p->prev`. Satu uji tersisa, di belakang.

Pada list kosong `tail` menunjuk simpul kepala, sehingga `backInsertion(x)` adalah `insertAfter(tail, x)`. `destroy` juga harus melepas simpul kepala; melupakannya adalah kebocoran yang nyata. Buku: Subbab 9.5.1–9.5.2.

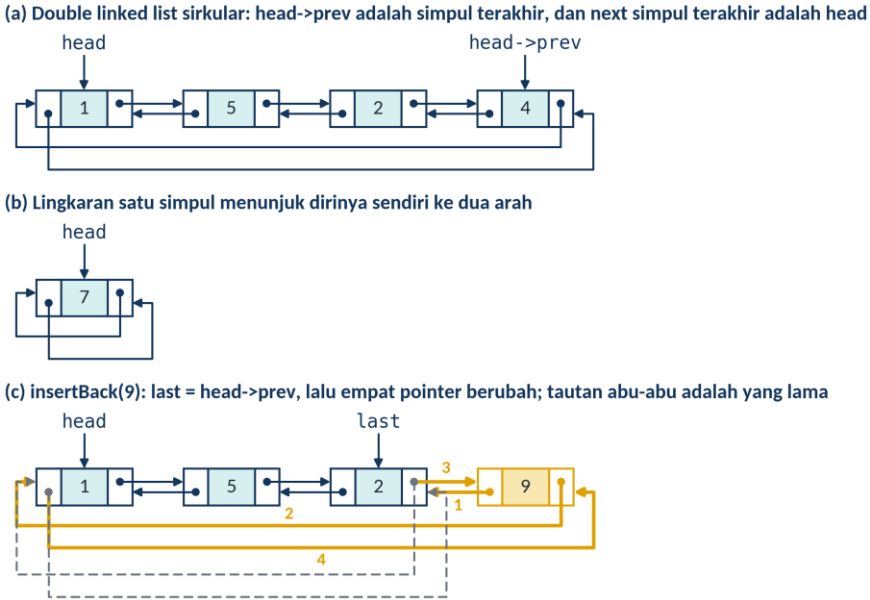
Simpul Kepala, Digambar



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.

***Gambar 9.12**: karena setiap simpul nyata kini punya pendahulu, tidak ada operasi yang pernah menulis ke `head` lagi. Buku: Subbab 9.5.1 dan Gambar 9.12.*

Menutup Kedua Rantai: Lingkaran



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`.

***Gambar 9.13**: `next` milik simpul terakhir adalah simpul pertama dan `prev` milik simpul pertama adalah simpul terakhir. Tanpa `tail` pun simpul terakhir adalah `head->prev`: satu pointer memberi akses $O(1)$ ke kedua ujung. Buku: Subbab 9.6.1 dan Gambar 9.13.*

Tanda Berhenti yang Baru

Tidak ada null yang dapat ditunggu; berhentilah saat kembali ke titik awal, dan kunjungi titik awal itu juga, jadi pakailah `do`–`while`:

```
const Node *p = head;
do {
    visit(p);
    p = p->next;          // backward: start at head->prev and follow prev
} while (p != head);
```

Ujilah list kosong *sebelum* perulangan. Penghapusan tanpa *sentinel* masih punya dua kasus khusus: satu-satunya simpul, dan simpul pertama.

Korsel di lobi Pustaka Ceria (`cdll.cpp`) adalah list ini dengan sebuah kursor: tombol Next dan Previous berputar melewati sambungan tanpa satu uji pun. Buku: Subbab 9.6.2–9.6.3.

Sentinel

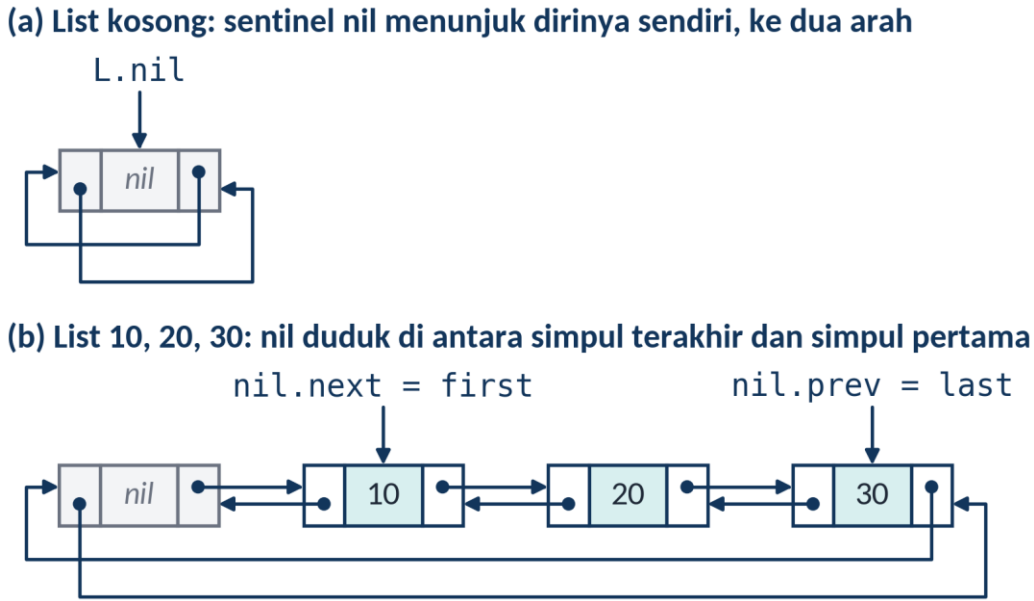
> **DEFINISI.** *Double linked list sirkular dengan sentinel* menyimpan satu simpul tanpa data `nil` secara permanen di dalam lingkaran. `nil.next` adalah simpul pertama dan `nil.prev` simpul terakhir. List kosong adalah `nil` seorang diri, dengan `nil.next = nil.prev = nil`.

Tidak ada pointer yang pernah null: setiap simpul, termasuk `nil`, berada di dalam lingkaran.

Tidak ada pointer luar yang pernah berubah: simpul pertama dan terakhir dibaca sebagai `nil.next` dan `nil.prev`.

Inilah representasi Cormen dkk. dalam Introduction to Algorithms, dan representasi di dalam sebagian besar linked list produksi. Buku: Subbab 9.7.1–9.7.2.

Sentinel, Digambar



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.

***Gambar 9.14**: simpul kepala dari Subbab 9.5 dan lingkaran dari Subbab 9.6, digabungkan. Buku: Subbab 9.7.1 dan Gambar 9.14.*

Sisip dan Hapus Tanpa `if`

Pseudokode 9.9 `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 9.10 `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
```

Pseudokode 9.9 *berisi empat langkah* ***Pseudokode 9.7*** *tanpa ujinya;* ***Pseudokode 9.10*** *hanya dua penugasan. Buku: Subbab 9.7.2.*

Contoh Terurai: Dua Kasus Berbahaya

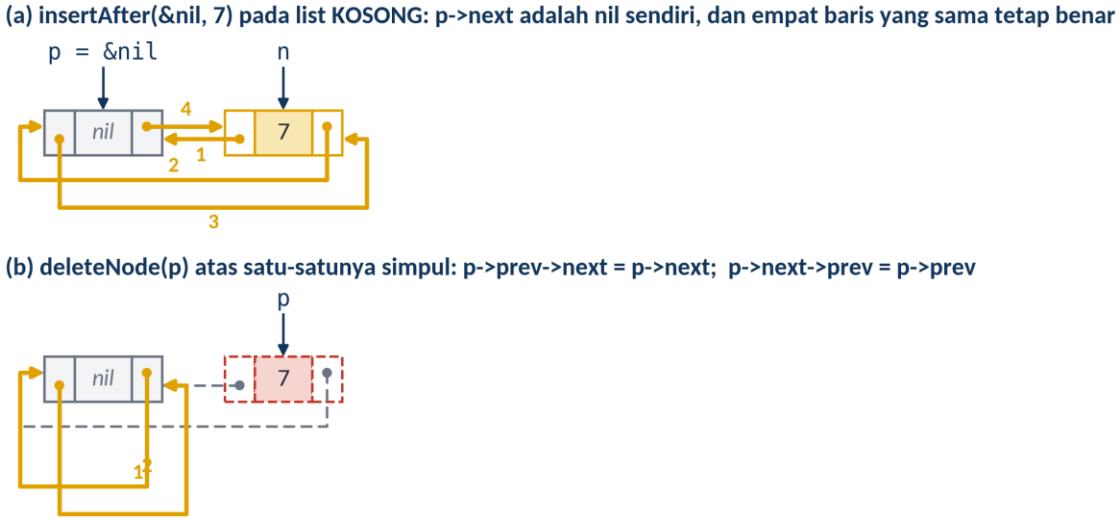
Sisip ke list kosong, $p = nil$: baris 2 memasang $n.prev = nil$; baris 3 memasang $n.next = nil$; baris 4 memasang $nil.prev = n$; baris 5 memasang $nil.next = n$. List berelemen satu yang benar.

Hapus satu-satunya simpul, $p.prev = p.next = nil$: baris 1 memasang $nil.next = nil$; baris 2 memasang $nil.prev = nil$. List kosong lagi.

> **MENGAPA INI BERHASIL.** Kasus khusus adalah tempat struktur tidak seragam. *Sentinel* membuatnya seragam: setiap simpul nyata punya dua tetangga nyata.

Kerjakan kedua kasus di papan dengan baris-baris kedua prosedur itu; halaman berikut memperlihatkan gambarnya. Buku: Subbab 9.7.2.

Kedua Kasus, Digambar



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.

***Gambar 9.15**: kasus yang dulu memerlukan kode khusus kini ditangani kode biasa. Buku: Subbab 9.7.2 dan Gambar 9.15.*

Mencari dengan Kunci di dalam Sentinel

Pseudokode 9.11 `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
```

$$C_{\text{biasa}}(n) = 2n + 1, \qquad C_{\text{sentinel}}(n) = n + 1 \quad \text{uji seluruhnya.} \tag{9.3}$$

Pseudokode 9.11 tidak mungkin melewati ujung, sehingga uji akhir keluar dari perulangan; Persamaan (9.3) menghitung uji pada pencarian yang gagal.

Tetap $\Theta(n)$ langkah; sentinel membagi dua faktor konstantanya. Knuth menganalisis trik yang sama untuk larik, dengan kunci diletakkan tepat sesudah elemen terakhir. Buku: Subbab 9.7.3 dan Persamaan (9.4).

List Ber-Sentinel dalam C++

Kode 9.4 Dari kelas `SentinelList` dalam `cdll_sentinel.cpp`: `insertAfter` dan `deleteNode` tidak berisi `if`, dan operasi ujung hanya satu baris.

```
    // 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 9.4**: sentinel `nil_` adalah anggota objek, sehingga tidak mungkin bocor; penyalinan dilarang, sebab tetangga sentinel salinan akan menunjuk balik ke yang asli. Baris terakhir keluarannya mencetak `nil points at itself: yes`. Buku: Subbab 9.7.4.*

Kasus Khusus, Dihitung

Tabel 9.2 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

***Tabel 9.2**: hanya sentinel yang menghapus semuanya, dengan harga satu simpul. Buku: Subbab 9.7.5.*

Deque untuk Meja Pengembalian

DEFINISI. *Deque* (*double-ended queue*) mendukung `pushFront`, `pushBack`, `popFront`, `popBack`, `front`, `back`, `isEmpty`, dan `size`. Bila hanya dipakai lewat `pushFront`/`popFront` ia adalah *stack*; lewat `pushBack`/`popFront`, *queue*.

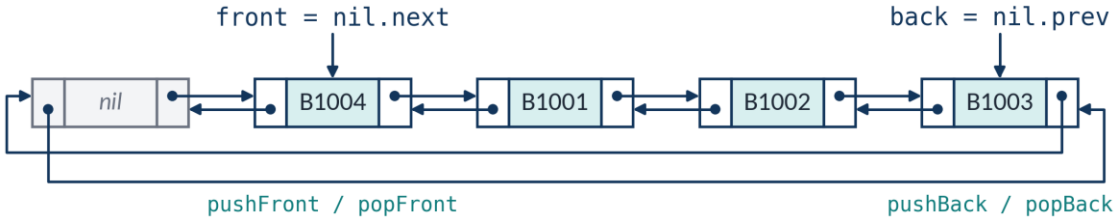
Buku yang dikembalikan masuk ke belakang troli; buku yang dipesan pembaca melompat ke **depan**; pindaian yang keliru diambil dari **belakang**.

Pada list ber-*sentinel* setiap operasi *deque* bernilai $O(1)$ menurut Persamaan (9.4), tanpa batas kapasitas:

$$\begin{aligned} \text{FRONT-INSERTION}(x) &= \text{LIST-INSERT-AFTER}(nil, x), \\ \text{BACK-INSERTION}(x) &= \text{LIST-INSERT-AFTER}(nil.prev, x), \\ \text{FRONT-DELETION}() &= \text{LIST-DELETE}(nil.next), \\ \text{BACK-DELETION}() &= \text{LIST-DELETE}(nil.prev). \end{aligned} \tag{9.4}$$

Kedua penghapusan mensyaratkan list tidak kosong, sebuah prasyarat dan bukan kasus khusus: `deque.cpp` *melempar* `std::underflow_error` *alih-alih melepas sentinel-nya sendiri. Buku: Subbab 9.7.2 dan 9.8.1–9.8.2.*

Troli Pengembalian, Digambar



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 $O(1)$.

***Gambar 9.16**: depan adalah `nil.next`, belakang adalah `nil.prev`. Buku: Subbab 9.8.2 dan Gambar 9.16.*

“Baru Dipinjam”: Pola LRU

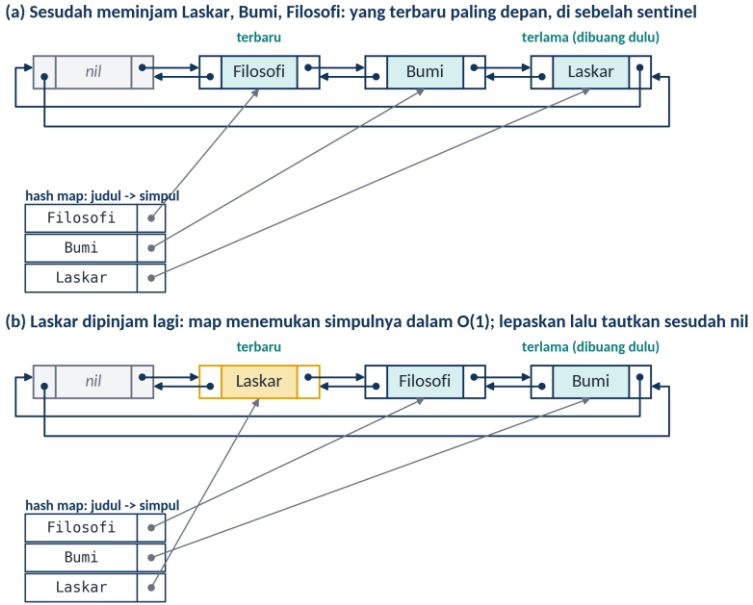
Panel kios menampilkan $k = 4$ judul *berbeda* yang paling baru dipinjam, yang terbaru di atas.

Judul yang dipinjam lagi naik ke puncak; bila judul baru membuat panel penuh, judul yang **paling lama tidak dipakai** jatuh keluar.

Dua struktur bekerja sama: **list ber-*sentinel*** menyimpan urutan kebaruan, **hash map** memetakan setiap judul ke *alamat simpulnya*.

Kebijakan yang sama menjalankan cache peramban, cache halaman basis data, dan tabel halaman sistem operasi. Map menyerahkan simpul di tengah; hanya simpul double linked yang dapat keluar dari tengah tanpa pencarian. Buku: Subbab 9.9.1–9.9.2.

Panel dan Map-nya



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.

***Gambar 9.17**: map menemukan simpulnya; list memindahkannya. Buku: Subbab 9.9.2 dan Gambar 9.17.*

BORROW dalam Waktu Harapan $O(1)$

Pseudokode 9.12 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
```

$$T_{\text{BORROW}} = O(1) \text{ harapan, berapa pun } k, \tag{9.5}$$

Pseudokode 9.12 *adalah satu pencarian di map ditambah sejumlah tetap penugasan pointer, sehingga berlaku Persamaan (9.5). Larik atau single linked list harus membayar pencarian $\Theta(k)$ setiap kali. Buku: Subbab 9.9.2 dan Persamaan (9.5).*

`borrow` dalam C++

Kode 9.5 Metode `borrow` dari `recently_borrowed.cpp`: pindahkan ke depan, atau sisipkan lalu usir yang tertua.

```
    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;
```

***Kode 9.5**: `unlink` dan `linkFront` adalah kedua prosedur sentinel; simpul dipindahkan tanpa dilepas. Buku: Subbab 9.9.3.*

Contoh Terurai: Delapan Peminjaman, $k = 4$

```
borrow Laskar Pelangi
borrow Bumi Manusia
borrow Filosofi Kopi
borrow Laskar Pelangi
   panel: [Laskar Pelangi] [Filosofi Kopi] [Bumi Manusia]
borrow Cantik Itu Luka
borrow Ronggeng Dukuh Paruk  (drops off: Bumi Manusia)
borrow Filosofi Kopi
borrow Negeri 5 Menara  (drops off: Laskar Pelangi)
   panel: [Negeri 5 Menara] [Filosofi Kopi] [Ronggeng Dukuh Paruk] [Cantik Itu Luka]
```

Telusuri di papan: Laskar naik dari dasar ke puncak tanpa ada yang jatuh; Bumi Manusia, yang tak tersentuh sejak peminjaman kedua, yang pertama jatuh; di akhir Laskar menjadi yang paling lama. Buku: Subbab 9.9.3.

Larik, Single Linked, Double Linked

Tabel 9.3 Larik dinamis, *single linked list*, dan *double linked list* dibandingkan, untuk n elemen (baris terpilih).

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 belakang	$O(1)$	$\Theta(n)$	$O(1)$
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, terurut	$\Theta(\lg n)$ (pencarian biner)	$\Theta(n)$	$\Theta(n)$
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

***Tabel 9.3**: "posisi yang dipegang" berarti program sudah punya pointer, iterator, atau indeks. Dari ujung terdekat, indeks k berongkos $\min(k, n-1-k)$ langkah, rata-rata sekitar $n/4$: tetap $\Theta(n)$. Buku: Subbab 9.10.1 dan Tabel 9.4.*

Mengukur Apa yang Dibeli `prev`

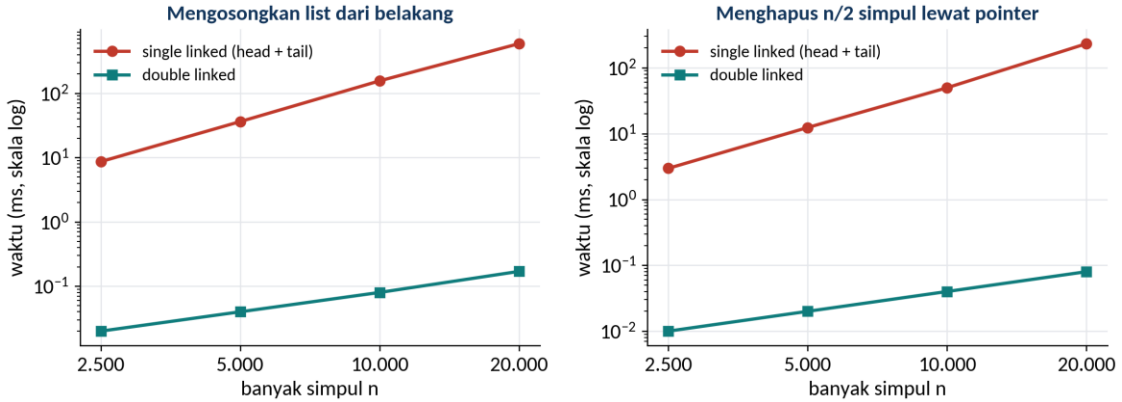
Mengosongkan *single linked list* dari belakang menempuh Persamaan (9.6) langkah; *double linked list* tidak menempuh satu langkah pun.

$$\sum_{m=2}^{n}(m-2) = 0 + 1 + \cdots + (n-2) = \frac{(n-1)(n-2)}{2}, \tag{9.6}$$

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

Setiap penggandaan n mengalikan waktu single linked sekitar 4 kali (kuadratik) dan waktu double linked sekitar 2 kali (linear): pada 20.000 simpul kira-kira 3.500 kali lebih cepat. Buku: Subbab 9.10.2 dan Persamaan (9.7).

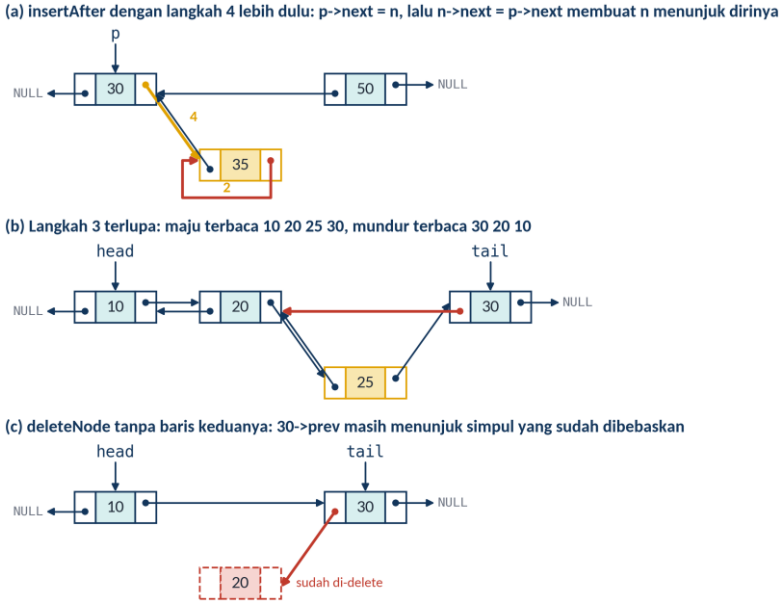
Hasil Pengukuran, Diplot



Gambar 9.18 Hasil pengukuran pada sumbu logaritmik: waktu *single linked list* tumbuh empat kali per penggandaan n (kuadratik), waktu *double linked list* dua kali (linear).

***Gambar 9.18**: waktunya bergantung pada mesin; bentuknya tidak. Bila posisi tidak dipegang, pencarian yang mendominasi dan larik biasanya menang. Buku: Subbab 9.10.2 dan Gambar 9.18.*

Tiga Bug Tautan Dua Arah



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.

***Gambar 9.19**: setiap bug memperbarui satu arah dan melupakan arah lainnya. Buku: Subbab 9.11.1 dan Gambar 9.19.*

Apa Kata Alat-Alatnya

`halflink`: maju 10 20 25 30, mundur 30 20 10, dan *tidak ada laporan*: setiap akses mengenai memori yang hidup. Pemeriksa memori menemukan kesalahan memori, bukan invarian yang rusak.

`stale`: Valgrind melaporkan *invalid read of size 4* di dalam blok 24 bita yang dilepas oleh `deleteNodeWrong`; AddressSanitizer melaporkan *heap-use-after-free* lalu menghentikan program.

`leak`: satu blok 24 bita *definitely lost*: *sentinel* di *heap* yang terlupa.

Daftar periksa: empat saat sisip, dua saat hapus; setiap tautan maju punya kembaran mundur; pasang kedua pointer luar; lepaskan *sentinel*; cetak ke dua arah.

*Pembacaan tidak sah yang kedua berukuran 8 pada offset 8, medan `prev`, tepat di tempat yang ditunjukkan **Gambar 9.2**. Program yang benar diakhiri "All heap blocks were freed". Buku: Subbab 9.11.2–9.11.3.*

Double Linked List dalam Pustaka Standar

Tabel 9.4 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`

***Tabel 9.4**: `std::list` GNU persis list ber-sentinel dari Subbab 9.7; `LinkedHashMap` Java dalam urutan akses serta `OrderedDict` dan `lru_cache` Python menyediakan pola LRU siap pakai. Buku: Subbab 9.12.*

Profitina dalam Praktik

Menjelajahi riwayat ke dua arah: dasbor analitik melangkah minggu demi minggu dengan `cur = cur->prev` atau `cur->next`, $O(1)$ per klik.

Menyimpan butir-butir terbaru: umpan dan daftar notifikasi adalah `collections.deque` dengan `maxlen`.

Menyimpan hasil yang mahal: *cache* LRU (`functools.lru_cache`, `OrderedDict`) adalah list ber-*sentinel* ditambah *hash map* seperti pada **Gambar 9.17**.

Profitina adalah platform visibilitas AI (GEO) yang servernya berupa aplikasi Python di atas FastAPI. Buku: Subbab 9.13.

Profitina Laundry: Mundur Satu Minggu

Kode 9.6 Dari `main` dalam `laundry_weeks.cpp`: kursor yang bergerak satu minggu mundur atau maju.

```
    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);
    }
```

***Kode 9.6**: setiap klik dan setiap perbandingan dengan minggu sebelumnya adalah satu langkah pointer; pada minggu tertua `prev` bernilai null dan tombol Back tidak berbuat apa-apa. Buku: Subbab 9.13.*

Kesalahan yang Sering Terjadi

Memperbarui satu arah saja, atau **menimpa tautan yang dipakai untuk bernavigasi** dalam `insertAfter`.

Lupa mengosongkan `prev` di depan yang baru (atau `next` di belakang yang baru), dan **lupa pointer luar yang kedua**.

Menguji null pada list sirkular: `while (p != nullptr)` tidak pernah berakhir.

Menghapus, membocorkan, atau menyalin *sentinel*; **mengira penghapusan $O(1)$ berarti penghapusan berdasarkan nilai $O(1)$.**

Memakai *double linked list* padahal larik sudah cukup: elemen kecil dan pembacaan berurutan menguntungkan larik.

Setiap kesalahan dalam buku disertai gambar atau subbab yang memperlihatkannya. Buku: Subbab 9.14.

Ringkasan: Lima Hal untuk Diingat

1. Dua pointer per simpul; setiap tautan maju punya kembaran mundur (Persamaan (9.1)), dan setiap operasi harus menjaganya.

2. Dengan `head` dan `tail`, setiap operasi di ujung bernilai $O(1)$, begitu pula sisip dan hapus pada simpul yang dipegang; penyisipan mengubah empat pointer, penghapusan dua.

3. *Sentinel* pada list sirkular menghapus setiap kasus khusus: tidak ada pointer null, tidak ada pointer luar yang berpindah.

4. *Deque* dan list LRU adalah list ber-*sentinel*; list LRU menambahkan *hash map* dari kunci ke simpul.

5. Harganya 8 bita per simpul dan lokalitas *cache*; cetak ke dua arah, periksa invarian, jalankan alat-alatnya.

Buku: Subbab 9.15.

Pertemuan Berikutnya

Bab 10 — Rekursi. Fungsi yang memanggil dirinya sendiri pada masalah yang lebih kecil: kasus dasar, kasus rekursif, dan *call stack* yang membuatnya bekerja.

Linked list adalah struktur rekursif: list itu kosong, atau sebuah simpul yang diikuti sebuah list.

Sebelum itu: kerjakan Soal Latihan 9.1, 9.3, 9.8, dan 9.10, dalam C++ lalu dalam Java atau Python.

Penyelesaiannya, lengkap dengan program, ada di buku. Buku: Subbab 9.16 dan 9.17.

Kuliah 10 — Rekursi

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 10

Dosen: Irfan Subakti — yifana@gmail.com

Hari ini rekursi berhenti menjadi sesuatu yang muncul di sana-sini dan menjadi alat yang dapat kita percayai, buktikan, dan ukur. Buku: Bab 10, Subbab 10.1–10.16.

Cakupan Pertemuan Ini

1. **Berpikir secara rekursif** — *base case*, *recursive case*, induksi, dan *call stack frame* demi *frame*.
2. **Lima soal klasik** — faktorial, Fibonacci (dan enam cara menjinakkannya), FPB Euklides, pemangkatan cepat, Menara Hanoi.
3. **Rekursi pada larik dan *linked list*** — kupas satu elemen, atau belah dua.
4. **Pohon rekursi** — ongkos sebuah rekursi, aras demi aras, dan tabel rekurens.
5. **Pertanyaan praktis** — *tail call*, batas kedalaman di C++, Java, dan Python, serta mengganti rekursi dengan *stack* eksplisit.

Setiap soal klasik menambah satu gagasan: rekursi paling sederhana, rekursi yang meledak, rekursi yang menyusut sangat cepat, membagi dua alih-alih mengurangi, dan rekursi yang optimal namun eksponensial. Buku: Subbab 10.1.

Titik Tolak

Rekursi sudah lama menunggu di balik layar: pencarian biner (Subbab 2.5.5), *merge sort* dan *quicksort* (Subbab 3.8–3.9).

Bab 5 menyelesaikan Menara Hanoi dengan tiga *stack* dan sebuah gelung, serta memperlihatkan **call stack** dengan tiga *frame* `countdown`.

Bab 7 menghitung panjang *list* dan membaliknya secara rekursif, sambil mengingatkan bahwa setiap simpul memakan satu *frame*.

Bab 9 ditutup dengan `functools.lru_cache`, yang pemakaiannya paling terkenal kita jumpai hari ini.

Bab 11, tentang pohon, akan memakai gagasan hari ini di hampir setiap halaman: pohon adalah akar dengan pohon-pohon yang lebih kecil di bawahnya. Buku: Subbab 10.1.

Pertanyaan yang Diteruskan Sepanjang Antrean

Dalam antrean hari Sabtu di meja peminjaman Pustaka Ceria, Anda bertanya kepada orang di depan: “Ada berapa orang di depan *Anda*?” lalu menambahkan satu.

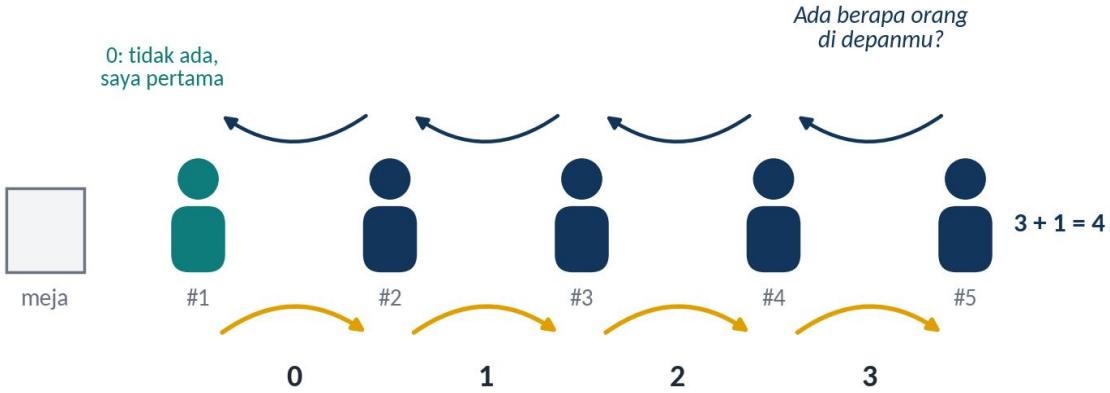
Semua orang menjalankan prosedur yang sama: bila tak ada orang di depan, jawab 0; bila ada, bertanya lalu jawab satu lebih banyak.

Setiap pertanyaan lebih kecil: orang yang ditanya punya lebih sedikit orang di depannya.

Seseorang menjawab tanpa bertanya: orang di meja langsung berkata “nol”.

Selama pertanyaan berjalan, setiap orang yang sudah bertanya berdiri menunggu, ingat bahwa ia harus menambahkan satu. Dalam program, orang-orang yang menunggu itu adalah frame pada stack. Buku: Subbab 10.2.1.

Pertanyaan Maju, Jawaban Kembali



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).

Gambar 10.1: lima orang menunggu pada saat terdalam, lalu jawaban 0, 1, 2, 3, 4 berjalan kembali. Buku: Subbab 10.2.1 dan Gambar 10.1.

Definisi Rekursif

$$f(0)=3, f(n)=2f(n-1)+3 \text{ untuk } n\geq 1. \tag{10.1}$$

DEFINISI. *Definisi rekursif* mendefinisikan suatu objek berdasarkan contoh-contoh yang lebih kecil dari objek itu sendiri. *Base case*-nya memberikan jawaban untuk contoh terkecil secara langsung; *recursive case*-nya memberikan jawaban untuk contoh yang lebih besar berdasarkan satu atau lebih contoh yang **benar-benar lebih kecil**. *Fungsi rekursif* adalah fungsi yang badannya memanggil fungsi itu sendiri.

Menguraikan Persamaan (10.1): $f(0)=3$, $f(1)=9$, $f(2)=21$, $f(3)=45$, $f(4)=93$.

"Benar-benar lebih kecil" itu penting: $g(n)=g(n)+1$ *tidak mendefinisikan apa pun, dan* $h(n)=h(n+1)-1$ *tidak pernah mencapai base case. Buku: Subbab 10.2.2 dan Persamaan (10.1).*

Membuktikan Kebenarannya: Induksi

Nilai 3, 9, 21, 45, 93 mengisyaratkan bentuk tertutup Persamaan (10.2):

$$f(n) = 6 \cdot 2^n - 3 \text{ untuk setiap } n \geq 0. \tag{10.2}$$

Base case. $n = 0$: $6 \cdot 2^0 - 3 = 3 = f(0)$.

Langkah induktif. Bila $f(n-1) = 6 \cdot 2^{n-1} - 3$, maka $f(n) = 2(6 \cdot 2^{n-1} - 3) + 3 = 6 \cdot 2^n - 3$.

> **WAWASAN KUNCI.** Jangan menelusuri setiap panggilan: periksa *base case*, *anggap* panggilan rekursif benar untuk masukan yang lebih kecil, lalu periksa cara menggabungkannya. Itulah bukti induksi.

Rekursi menghitung dengan mereduksi ke kasus yang lebih kecil; induksi membuktikan dengan membangun dari kasus-kasus itu. Pemrogram menyebutnya lompatan iman rekursif; induksilah yang membenarkannya. Buku: Subbab 10.2.3 dan Persamaan (10.2).

Dari Definisi ke Program

Kode 10.1 Fungsi `f` dari `01_recursive_f.cpp`: Persamaan (10.1) dalam C++, dan jejak yang dicetak kembarannya, `fTraced`.

```cpp
long long f(int n) {
    if (n == 0) return 3;     // base case
    return 2 * f(n - 1) + 3;  // recursive case: a smaller n
}
```

```
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
```

Kode 10.1: empat panggilan belum selesai pada saat yang sama; semuanya selesai dengan urutan terbalik dari urutan dimulainya. Program lalu mencocokkan `f` dengan bentuk tertutupnya untuk $n = 0$ sampai 10. Buku: Subbab 10.2.4.

Tiga Aturan, dan Apa yang Rusak

1. *Base case* ada dan diuji lebih dulu. 2. Setiap panggilan membuat kemajuan ke arahnya, dalam ukuran yang tidak dapat turun selamanya. **3. Percayai panggilannya, gabungkan dengan benar.**

Langgar aturan 2, `return n * factRec(n + 1);`, dan *frame* menumpuk sampai *stack* penuh:

C++: *segmentation fault* (Windows: eksepsi *stack overflow*), tanpa pesan. **Java**: `StackOverflowError`. **Python**: `RecursionError` sesudah 1.000 aras.

> **PERINGATAN.** Rekursi yang tidak maju bukanlah program yang lambat; ia *crash* dalam sepersekian detik. Ujilah 0, 1, dan 2 lebih dulu.

Harga rekursi: satu frame per panggilan yang menunggu, beberapa nanodetik pembukuan per panggilan, dan, bila ditulis naif, submasalah yang sama diselesaikan berulang-ulang. Buku: Subbab 10.2.5 dan 10.3.4.

Faktorial: Satu Frame per Panggilan

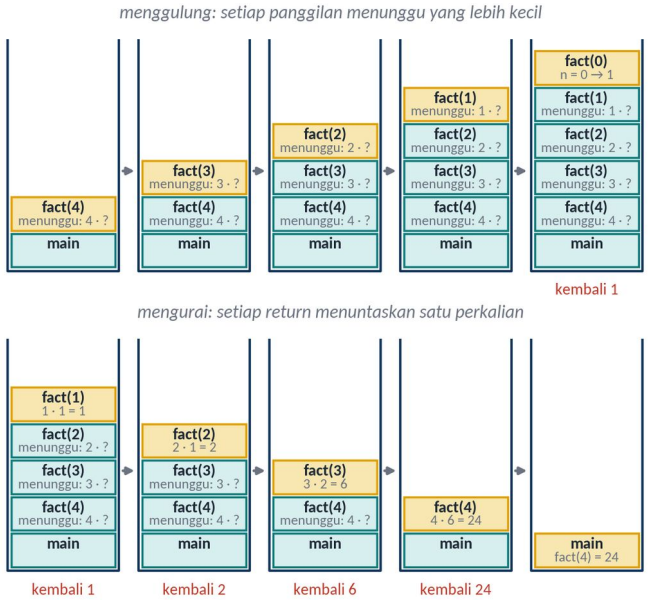
$$n! = \begin{cases} 1 & \text{jika } n = 0, \\ n \cdot (n-1)! & \text{jika } n \geq 1. \end{cases} \tag{10.3}$$

Ketika `fact(3)` memanggil `fact(2)`, program mendorong *frame* baru: fungsi yang sama, *aktivasi* yang berbeda, dengan salinan `n` miliknya sendiri.

Menguraikan Persamaan (10.3): $1! = 1$, $2! = 2$, $3! = 6$, $4! = 4 \cdot 6 = 24$.

Tidak ada yang istimewa dari call stack bagi rekursi, dan justru di situlah intinya: frame-frame itu memisahkan panggilan yang menunggu. Buku: Subbab 10.3.1–10.3.2 dan Persamaan (10.3).

Menumpuk lalu Mengurai



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.

Gambar 10.2: tidak ada perkalian saat turun; keempatnya terjadi saat kembali, dengan urutan terbalik dari panggilan; stack mencapai $n+2$ frame. Buku: Subbab 10.3.2 dan Gambar 10.2.

Faktorial dengan Gelung dan dengan Rekursi

Kode 10.2 `factIter` dan `factRec` dari `02_factorial.cpp`.

```
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)!
}
```

Kode 10.2: gelung mengalikan ke dalam hasil berjalan; rekursi mengikuti Persamaan (10.3) kata demi kata.

Keduanya sepakat sampai $20!$*, faktorial terbesar dalam 64 bit;* $21!$ *diam-diam berputar menjadi bilangan yang tampak masuk akal tetapi salah. Java memerlukan* `BigInteger`*; bilangan bulat Python tidak pernah meluap. Buku: Subbab 10.3.3 dan 10.4.1.*

Ongkos Masing-Masing

$$T(0)=c_0, T(n)=T(n-1)+c\,(n\geq 1). \tag{10.4}$$

Menguraikan Persamaan (10.4): $T(n)=T(n-1)+c=\cdots=c_0+c\,n=\Theta(n)$. Gelungnya juga $\Theta(n)$.

Memorinya berbeda: gelung perlu $O(1)$; rekursi menyimpan $n+1$ *frame*, $\Theta(n)$.

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)
```

Pseudokode 10.1: waktu sebuah rekursi adalah kerja semua panggilannya; memori tambahannya adalah jumlah frame terbanyak di stack pada saat yang sama. Invarian gelung adalah induksi atas banyaknya iterasi. Buku: Subbab 10.4.1–10.4.2 dan Persamaan (10.4).

Kelinci-Kelinci Fibonacci

Liber Abaci, 1202: sepasang kelinci yang baru lahir menjadi dewasa dalam satu bulan, lalu melahirkan sepasang baru setiap bulan; tidak ada kelinci yang mati.

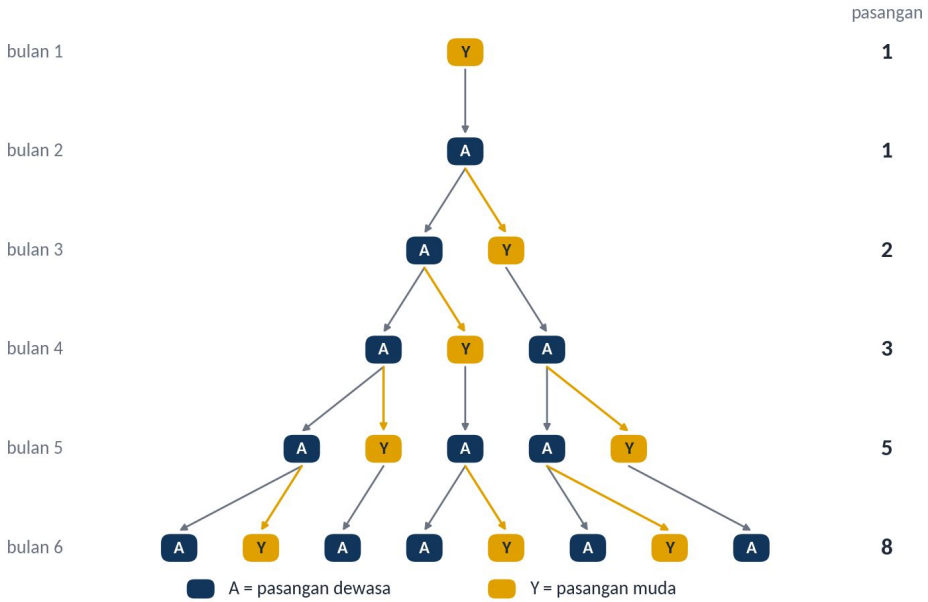
Pasangan pada bulan n = pasangan pada bulan $n-1$ (tidak ada yang mati) + pasangan pada bulan $n-2$ (yang dewasa, masing-masing melahirkan satu), yaitu Persamaan (10.5):

$$F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2)(n\geq 2), \tag{10.5}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144: 144 pasang pada bulan ke-12, jawaban Fibonacci.

Barisan ini dinamai oleh Édouard Lucas, penemu Menara Hanoi; ia muncul lagi dalam algoritma Euklides dan dalam ongkos perhitungan naifnya sendiri. Buku: Subbab 10.5.1 dan Persamaan (10.5).

Kelinci, Bulan demi Bulan



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.

Gambar 10.3: panah abu-abu, pasangan yang terus hidup; panah kuning tua, pasangan dewasa yang melahirkan pasangan muda. Buku: Subbab 10.5.1 dan Gambar 10.3.

Rekursi Naif Meledak

`fibNaive(n)` mengembalikan `fibNaive(n - 1) + fibNaive(n - 2)`: benar, dan salah satu algoritma terburuk dalam buku ini. Panggilannya mengikuti Persamaan (10.6):

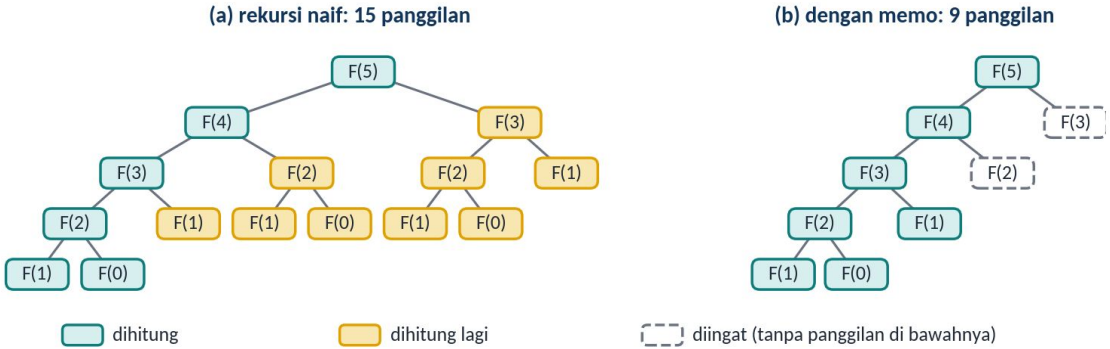
$$C(0)=C(1)=1, C(n)=C(n-1)+C(n-2)+1 (n\geq 2). \tag{10.6}$$

$$C(n)=2F(n+1)-1. \tag{10.7}$$

$$F(n)\geq \varphi^{n-2} \text{ untuk setiap } n\geq 1. \tag{10.8}$$

Menurut Persamaan (10.7) dan Persamaan (10.8), panggilannya tumbuh sebagai $\Theta(\varphi^n)$ dengan $\varphi\approx 1,618$: setiap $+5$ pada n melipatkan kerja sekitar 11 kali. Keduanya induksi dua baris. Buku: Subbab 10.5.2 dan Persamaan (10.6)–(10.8).

Pohon Panggilan $F(5)$



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.

Gambar 10.4: (a) 15 panggilan, $F(3)$ dua kali, $F(2)$ tiga kali; (b) dengan memoisasi, 9 panggilan. Buku: Subbab 10.5.2–10.5.3 dan Gambar 10.4.

Memoisasi: Ingat Apa yang Sudah Dihitung

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]
```

Pseudokode 10.2 menghitung setiap $F(k)$ sekali saja: $2n-1$ panggilan, waktu dan memori $\Theta(n)$. Inilah *dynamic programming top-down*.

> **WAWASAN KUNCI.** Rekursi eksponensial sering kali eksponensial hanya karena menyelesaikan submasalah yang sama berulang-ulang. Bila submasalah yang *berbeda* sedikit, simpan setiap jawaban pada kali pertama.

Dalam Python, `@functools.lru_cache(maxsize=None)` memoisasi fungsi secara otomatis, di dalam list ber-sentinel dari Subbab 9.12.3. Buku: Subbab 10.5.3.

Bottom-Up, dan Kembarannya yang Rekursif Ekor

Kode 10.3 Dari `03_fibonacci.cpp`: dua variabel, dan gelung yang sama ditulis sebagai rekursi ekor.

```
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); }
```

Kode 10.3: setiap langkah mengganti (a, b) dengan $(b, a+b)$. Di `fibAux` panggilan rekursif adalah tindakan paling akhir, sebuah tail call; `fibNaive` dan `factRec` tidak rekursif ekor. Buku: Subbab 10.5.4–10.5.5.

Fast Doubling dan Rumus Binet

$$F(2k)=F(k)\bigl(2F(k+1)-F(k)\bigr),\quad F(2k+1)=F(k)^2+F(k+1)^2. \tag{10.9}$$

Persamaan (10.9) memberi $F(n)$ dari $F(\lfloor n/2\rfloor)$: $\lfloor \lg n \rfloor + 2$ panggilan, 8 untuk $n=90$.

$$F(n)=\frac{\varphi^n-\psi^n}{\sqrt{5}}. \tag{10.10}$$

PERANGKAP. Persamaan (10.10) dalam `double` tepat hanya sampai $F(70)$: pada $n=71$ ia menjawab 308.061.521.170.130, bukan 308.061.521.170.129.

Periksa identitasnya dengan $k=3$: $F(6)=2(2\cdot3-2)=8$, $F(7)=2^2+3^2=13$. Jawaban bilangan bulat yang eksak memerlukan aritmetika bilangan bulat yang eksak. Buku: Subbab 10.5.6–10.5.8 dan Persamaan (9), (10).

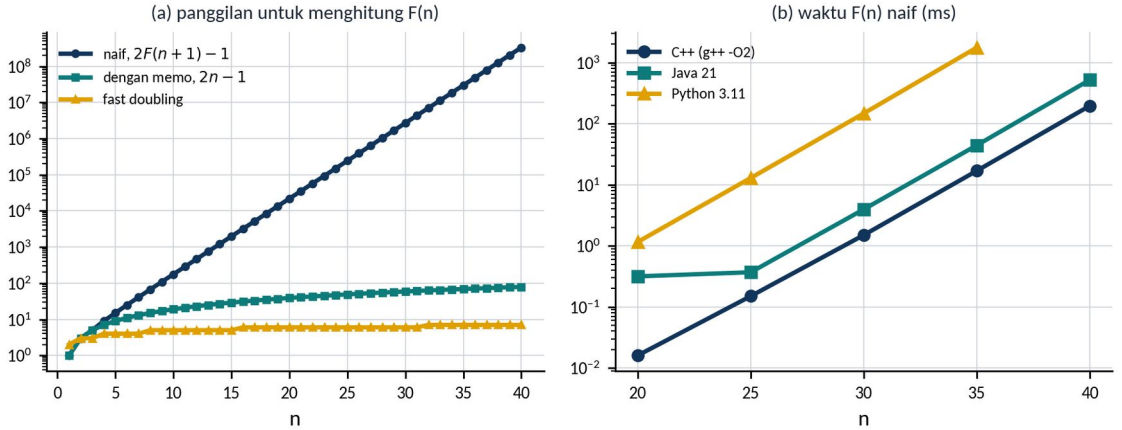
Tujuh Cara Menghitung $F(n)$

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 $F(70)$

Tabel 10.1: semua metode eksak sepakat pada $F(90)=2.880.067.194.370.816.120$; $F(92)$ adalah yang terbesar yang muat dalam bilangan bulat bertanda 64 bit. Buku: Subbab 10.5.9.

Diukur: Panggilan dan Waktu



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.

Gambar 10.5: pertumbuhan eksponensial tampak sebagai garis lurus pada skala logaritmik; setiap bahasa naik sekitar $\varphi^5 \approx 11$ kali tiap lima langkah. Kemiringannya milik algoritma. Buku: Subbab 10.5.9 dan Gambar 10.5.

Faktor Persekutuan Terbesar Euklides

$$\gcd(a,0)=a,\gcd(a,b)=\gcd(b,a \bmod b)(b>0), \tag{10.11}$$

Mengapa benar. Dengan $a=qb+r$: pembagi a dan b membagi $r=a-qb$; pembagi b dan r membagi $a=qb+r$. Pembagi bersamanya sama, maka yang terbesar pun sama.

Mengapa berhenti. Argumen kedua berubah dari b menjadi $a \bmod b<b$ dan tidak pernah negatif.

Persamaan (10.11) berasal dari Elements karya Euklides, sekitar 300 SM; Knuth menyebutnya "kakek dari semua algoritma". Buku: Subbab 10.6.1 dan Persamaan (10.11).

Contoh Terurai: $\gcd(228, 90)$

$$228 = 2 \cdot 90 + 48 \Rightarrow \gcd(228, 90) = \gcd(90, 48)$$

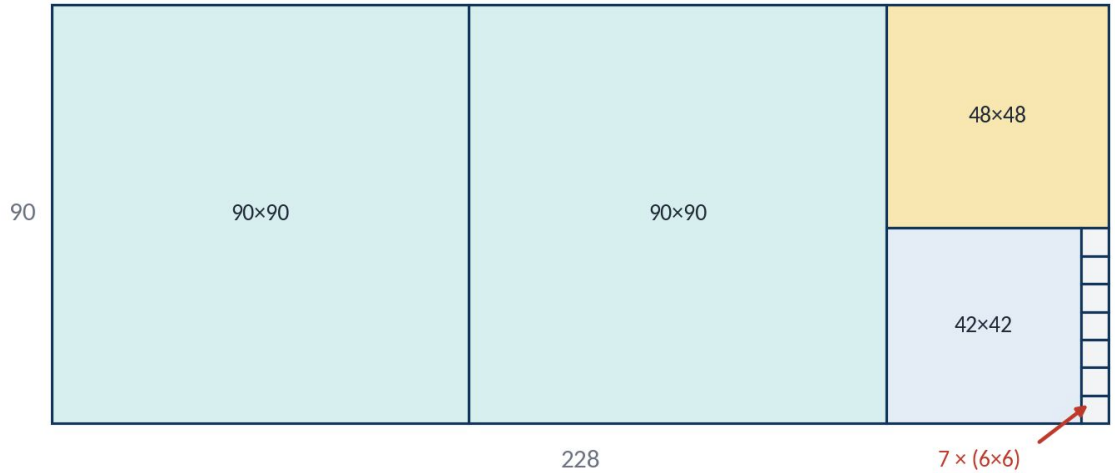
$$90 = 1 \cdot 48 + 42 \Rightarrow \gcd(90, 48) = \gcd(48, 42)$$

$$48 = 1 \cdot 42 + 6 \Rightarrow \gcd(48, 42) = \gcd(42, 6)$$

$$42 = 7 \cdot 6 + 0 \Rightarrow \gcd(42, 6) = \gcd(6, 0) = 6$$

Jawabannya adalah sisa tak nol yang terakhir, 6, sesudah empat pembagian. Bila argumennya tertukar, $\gcd(90, 228)$*, pembagian pertama sekadar menukarnya kembali. Buku: Subbab 10.6.2.*

Euklides sebagai Memotong Persegi



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 $\gcd(228, 90) = 6$.

Gambar 10.6: persegi terkecil, 6×6, mengukur kedua sisi sejumlah bulat kali ($228 = 38 \cdot 6$, $90 = 15 \cdot 6$). Buku: Subbab 10.6.2 dan Gambar 10.6.

Rekursif dan Iteratif, dan Teorema Lamé

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
```

TEOREMA (Lamé, 1844). Bila $a > b \geq 1$ dan algoritma Euklides melakukan k pembagian, maka $a \geq F(k+2)$ dan $b \geq F(k+1)$.

$$k \leq 1 + \log_{\varphi} b \approx 1 + 4.785 \log_{10} b\,. \tag{10.12}$$

Pseudokode 10.3: panggilan rekursifnya adalah tail call, sehingga gelungnya mudah. Menurut Persamaan (10.12) banyaknya pembagian paling banyak sekitar lima kali banyaknya digit desimal b: paling banyak 93 untuk bilangan 64 bit. Kasus terburuk di bawah 100 adalah $\gcd(89, 55)$, 9 pembagian. Buku: Subbab 10.6.3–10.6.4.

Pemangkatan Cepat: Bagi Dua, Jangan Kurangi

$$m^0 = 1, m^n = m \cdot m^{n-1} (n \geq 1) \tag{10.13}$$

$$m^n = \begin{cases} 1 & \text{jika } n = 0, \\ (m^{\lfloor n/2 \rfloor})^2 & \text{jika } n \text{ genap}, \\ (m^{\lfloor n/2 \rfloor})^2 \cdot m & \text{jika } n \text{ ganjil}. \end{cases} \tag{10.14}$$

Persamaan (10.13) memerlukan n perkalian; Persamaan (10.14) membuat **satu** panggilan pada separuh pangkat lalu mengkuadratkan hasilnya.

Memanggilnya dua kali, `power(m, n/2) * power(m, n/2)`, mengembalikan seluruh kerja yang sudah dihemat. Buku: Subbab 10.7.1–10.7.2 dan Persamaan (10.13), (10.14).

Contoh Terurai: 3^{13}

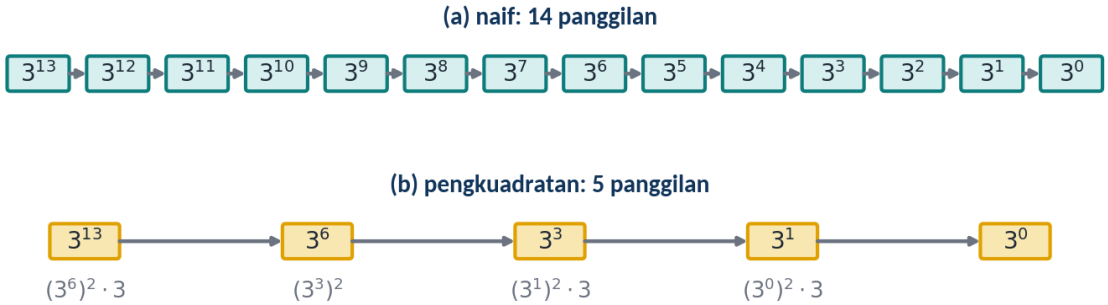
Pseudokode 10.4 Pemangkatan dengan pengkuadratan berulang.

```
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
```

Turun: $13 \to 6 \to 3 \to 1 \to 0$. Naik: $3^1 = 1 \cdot 1 \cdot 3 = 3$, $3^3 = 3 \cdot 3 \cdot 3 = 27$, $3^6 = 27 \cdot 27 = 729$, $3^{13} = 729 \cdot 729 \cdot 3 = 1.594.323$.

Pseudokode 10.4: lima panggilan alih-alih empat belas. Dalam biner $13 = 1101_2$: kuadratkan untuk setiap digit, kalikan dengan m untuk setiap angka 1. Buku: Subbab 10.7.2.

Mengurangi Satu lawan Membagi Dua



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.

Gambar 10.7: di bawah setiap panggilan, cara hasilnya dibangun dari panggilan berikutnya. Buku: Subbab 10.7.2 dan Gambar 10.7.

Panggilan Logaritmik, dan Pangkat Modulo p

$$\text{panggilan}(n) = \lfloor \lg n \rfloor + 2 \,(n \geq 1), \text{perkalian} \leq 2(\lfloor \lg n \rfloor + 1). \tag{10.15}$$

Persamaan (10.15): 5 panggilan untuk $n = 13$, 31 untuk 10^9, 61 untuk 10^{18}.

Reduksi sesudah setiap perkalian, sebab $(x\,y) \bmod p = ((x \bmod p)(y \bmod p)) \bmod p$; dengan $p = 1.000.000.007$ setiap hasil kali muat dalam 64 bit.

Kode 10.4 `powerFast` dari `06_power.cpp`.

```
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;
}
```

Kode 10.4: $7^{10^{18}} \bmod p = 259.616.729$ *dalam 61 panggilan. RSA dalam setiap koneksi TLS menghitung pangkat seperti ini;* `pow(7, n, p)` *milik Python melakukan hal yang sama. Buku: Subbab 10.7.2–10.7.3.*

Menara Hanoi, Secara Rekursif

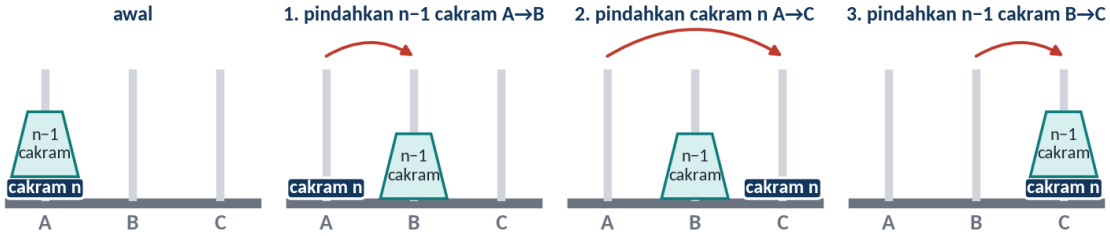
Untuk memindahkan n cakram dari `from` ke `to` dengan penolong `via`:

1. pindahkan $n-1$ cakram yang lebih kecil dari `from` ke `via`; **2.** pindahkan cakram n dari `from` ke `to`; **3.** pindahkan $n-1$ cakram itu dari `via` ke `to`.

Langkah 1 dan 3 adalah masalah yang sedang kita selesaikan, tetapi *lebih kecil*: hanya itu yang dibutuhkan rekursi. Cakram terbesar tidak pernah menghalangi langkah.

Subbab 5.2 sudah menyadarinya: menyelesaikan tiga cakram berarti menyelesaikan dua cakram dua kali, dengan satu langkah di antaranya. Base case-nya $n=0$, tidak ada yang dipindahkan. Buku: Subbab 10.8.1.

Gagasan Rekursifnya, Digambar



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.

Gambar 10.8: $n-1$ cakram yang lebih kecil digambar sebagai satu balok. Buku: Subbab 10.8.1 dan Gambar 10.8.

Hanoi dalam Pseudokode dan C++

Pseudokode 10.5 Menara Hanoi secara rekursif.

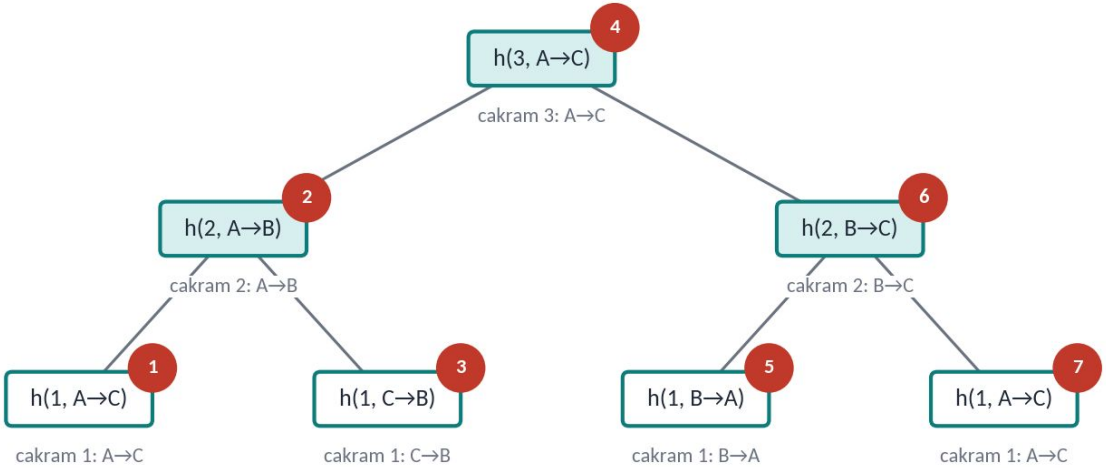
```
HANOI(n, from, to, via)
 1.  if n = 0
 2.      return                          // base case: nothing to move
 3.  HANOI(n − 1, from, via, to)         // clear the n − 1 smaller disks onto via
 4.  move disk n from `from` to `to`
 5.  HANOI(n − 1, via, to, from)         // bring them back on top of disk n
```

Kode 10.5 `hanoi` dari `07_hanoi.cpp`; setiap langkah dimainkan pada tiga tiang `std::stack<int>` yang menolak langkah ilegal.

```cpp
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;
}
```

Pseudokode 10.5 dan Kode 10.5: tidak ada `ILLEGAL MOVE` dalam 2.097.130 langkah untuk 1 sampai 20 cakram, dan kedalamannya hanya n. Java memakai `ArrayDeque<Integer>`, Python sebuah list. Buku: Subbab 10.8.2.

Contoh Terurai: Urutan Langkah



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.

Gambar 10.9: subpohon kiri, langkah sendiri, subpohon kanan (penyusuran inorder, Bab 11): cakram 1 A→C, 2 A→B, 1 C→B, 3 A→C, 1 B→A, 2 B→C, 1 A→C. Buku: Subbab 10.8.3 dan Gambar 10.9.

Tepat $2^n - 1$ Langkah, dan Tidak Kurang

$$M(0) = 0,\; M(n) = 2\,M(n-1) + 1 \quad (n \geq 1). \tag{10.16}$$

Batas atas: $M(n) = 2(2^{n-1} - 1) + 1 = 2^n - 1$ dengan induksi.

Batas bawah: sebelum langkah *pertama* cakram n, semua $n-1$ cakram kecil berada di tiang ketiga; sesudah langkah *terakhir*-nya semuanya harus kembali. Jadi $L(n) \geq 2\,L(n-1) + 1 \geq 2^n - 1$.

Algoritmanya **optimal**: ongkos eksponensialnya milik masalah itu sendiri. Berbeda dengan Fibonacci naif, kedua panggilannya tidak bertumpang tindih, sehingga tidak ada yang perlu diingat.

Persamaan (10.16) untuk 64 cakram memberi 18.446.744.073.709.551.615 langkah, sekitar 585 miliar tahun dengan satu langkah per detik. Buku: Subbab 10.8.4 dan Persamaan (10.16).

Rekursi pada Larik

Kupas satu elemen: $A[0..n-1]$ adalah $A[0..n-2]$ ditambah $A[n-1]$; *base case* larik kosong; kedalaman n.

Belah dua: $A[lo..hi]$ adalah dua belahan di sekitar $mid = lo + \lfloor (hi - lo)/2 \rfloor$; kedalaman $\lceil \lg n \rceil + 1$.

`sumArr` mengupas; `maxArr` membelah; `binSearch` membelah lalu membuang; `reverseArr` menukar kedua ujung lalu berekursi ke dalam; `isPalindrome` memeriksa huruf ujung dan kata di antaranya.

Pada jumlah peminjaman 12, 7, 31, 4, 25, 18 di Pustaka Ceria: jumlah 97, maksimum 31; "racecar", "rotor", "a", dan "" adalah palindrom, "library" bukan. Buku: Subbab 10.9.1.

Linked List adalah Struktur Rekursif

Kode 10.6 Dari `08_arrays_lists.cpp`: mencetak mundur, dan membuang setiap simpul yang memuat sebuah kunci.

```
void printBackward(const Node *p) {  // recurse, then print
    if (p == nullptr) return;
    printBackward(p->next);
    std::printf(" %d", p->data);
}

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;
}
```

Kode 10.6: berekursi dulu, baru mencetak, dan stack memerankan tautan `prev` yang tidak ada. `removeAll` mengembalikan head yang baru, sehingga `head = removeAll(head, 40);` tidak memerlukan kasus khusus. Valgrind: semua blok heap dilepas. Buku: Subbab 10.9.2–10.9.3.

Pohon Rekursi

Satu simpul per panggilan, panggilan rekursifnya sebagai anak, dan di setiap simpul kerja yang dilakukan panggilan itu *sendiri*.

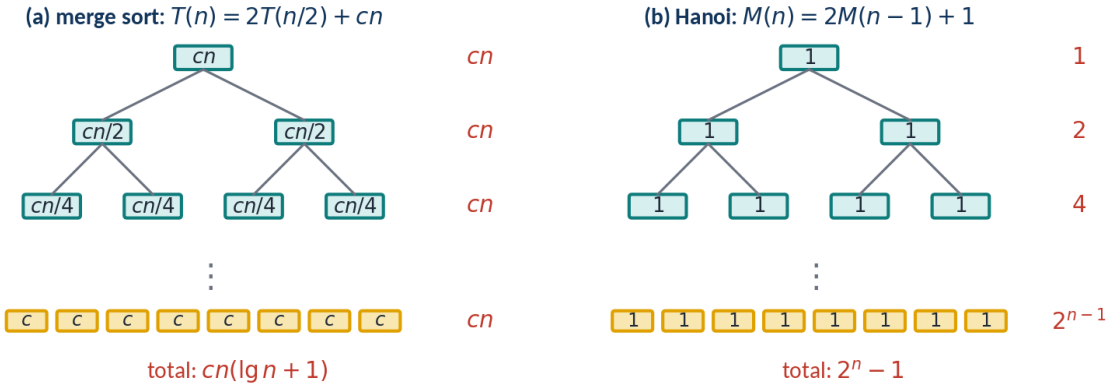
Waktu total = jumlah semua simpul, dihitung aras demi aras; memori *stack* = tinggi pohon.

$$T(n) = 2T(n/2) + cn \Rightarrow T(n) = cn(\lg n + 1) = \Theta(n \lg n). \tag{10.17}$$

Merge sort, Persamaan (10.17): $\lg n + 1$ aras berongkos cn. Hanoi: aras i punya 2^i langkah, totalnya $2^n - 1$.

Membagi dua menghasilkan sedikit aras; mengurangi satu menghasilkan n aras, dengan simpul berlipat dua di setiap aras. Buku: Subbab 10.10.1–10.10.2 dan Persamaan (10.17).

Dua Pohon, Dua Total



Gambar 10.10 Pohon rekursi dengan ongkos setiap aras di sebelah kanan. (a) *Merge sort*: $\lg n + 1$ aras yang masing-masing berongkos $c\,n$. (b) Hanoi: aras i berongkos 2^i, dan jumlah semua aras $2^n - 1$.

Gambar 10.10: ongkos setiap aras di sebelah kanan. Buku: Subbab 10.10.2 dan Gambar 10.10.

Tabel Rekurens

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))
$T(n)=T(n-1)+T(n-$	$\Theta(\varphi^n)$	Fibonacci naif (Persamaan (10.6))

Tabel 10.2: `maxArr` membelah dua tetapi tetap $\Theta(n)$, sebab ia harus melihat setiap elemen; membelah dua menghemat waktu hanya bila salah satu belahan dapat dibuang. Buku: Subbab 10.10.3.

Tail Call

```
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
}

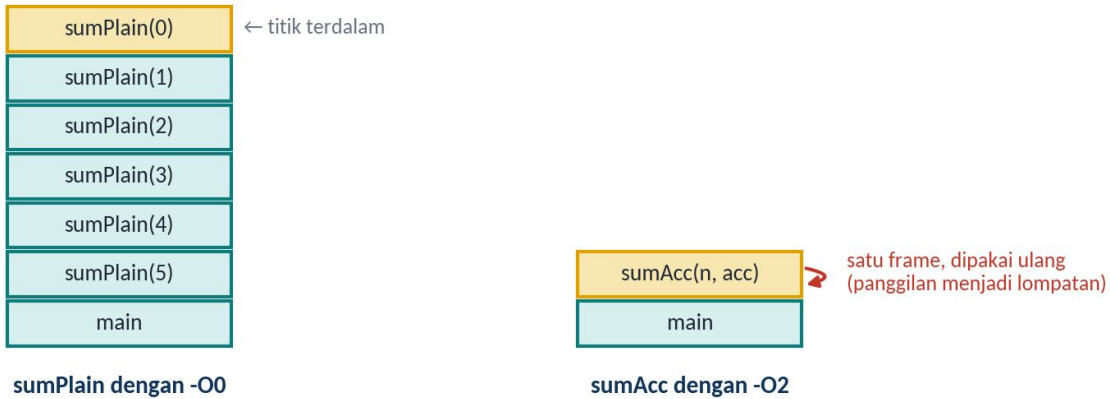
long long sumAcc(long long n, long long acc) {  // acc + 1 + 2 + ... + n
    if (n == 0) return acc;
    return sumAcc(n - 1, acc + n);              // a tail call
}
```

`sumAcc` membawa jumlah sementara dalam sebuah *akumulator*; *frame*-nya tak pernah dibutuhkan lagi, sehingga panggilannya dapat menjadi lompatan: *tail-call elimination*.

GCC pada `-O2`: `sumAcc` menjadi gelung, demikian pula `sumPlain` (GCC sendiri yang memperkenalkan akumulatornya). Pada `-O0` keduanya *crash* pada sepuluh juta.

Opsinya `-foptimize-sibling-calls`, aktif pada `-O2`, `-O3`, `-Os`. Standar C++ tidak menjanjikan apa pun: jangan pernah bergantung padanya. Buku: Subbab 10.11.1–10.11.3.

Satu Frame Alih-Alih Enam



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).

Gambar 10.11: sesudah tail-call elimination, satu frame melayani seluruh rekursi. Buku: Subbab 10.11.2 dan Gambar 10.11.

JVM dan CPython Tidak Menghapus Tail Call

```
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
```

Di Java, `sumPlain` maupun `sumAcc` melempar `StackOverflowError` pada $n = 10^6$; `sumLoop` berhasil.

Pencipta Python menolak *tail-call elimination* karena ia menghapus jejak *stack* yang dipakai pemrogram untuk *debugging*.

Aturannya: di Java dan Python fungsi rekursif ekor sama dalamnya dengan masukannya; bila Anda menginginkan gelung, tulislah gelung. Buku: Subbab 10.11.4.

Seberapa Dalam Rekursi Boleh Berjalan?

$$d_{\max} \approx \frac{S}{s} \tag{10.18}$$

C++: *stack* 8 MB di Linux dan macOS, 1 MB di Windows; *frame* 72 bita pada `-O2`: 116.000 aras berhasil, 117.000 *crash*.

Java: `-Xss`, bawaan 1.024 KB; terukur 10.730 sampai 59.020 aras, berubah-ubah mengikuti JIT.

Python: batas berupa hitungan, 1.000 secara bawaan (`RecursionError` pada kedalaman 997); `sys.setrecursionlimit` dan `threading.stack_size` menaikkannya.

Persamaan (10.18) meramalkan $8.388.608/72 \approx 116.508$ *aras. Batas kedalaman program C++ bahkan bergantung pada tingkat optimisasinya. Buku: Subbab 10.12 dan Persamaan (10.18).*

Rekursi atau Iterasi?

Tabel 10.3 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

Tabel 10.3: ubahlah bila kedalamannya sebanding dengan masukan dan masukannya mungkin besar; pertahankan rekursi bila kedalamannya logaritmik atau kecil dan diketahui. Buku: Subbab 10.13.1.

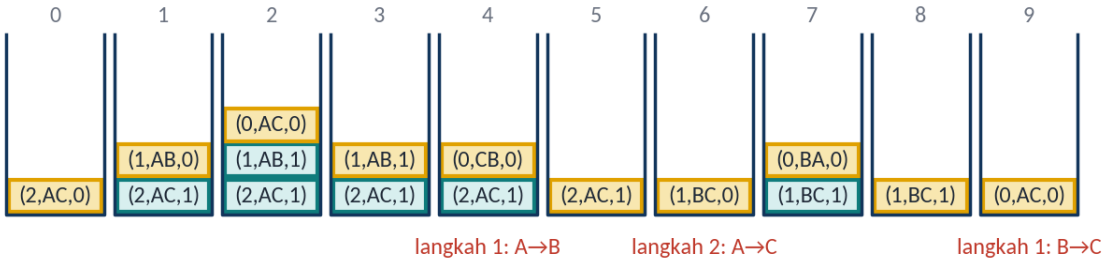
Stack Frame Eksplisit

Kode 10.7 Gelung `hanoiIter` dalam `11_explicit_stack.cpp`: *stack frame* milik kita sendiri, masing-masing dengan sebuah tahap.

```
    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();
    }
```

Kode 10.7: `Frame` menyimpan n, tiang-tiang, dan tahap; panggilan kedua adalah tail call, jadi frame dikeluarkan sebelum panggilan itu dibuat. Paling banyak $n+1$ frame; stack-nya tinggal di heap. Buku: Subbab 10.13.2–10.13.3.

Stack Eksplisit, Putaran demi Putaran



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.

Gambar 10.12: sesudah putaran 9 stack kosong dan ketiga langkah sudah dibuat. List sejuta simpul disusuri mundur dengan cara yang sama, memakai `vector` berisi pointer. Buku: Subbab 10.13.3–10.13.4 dan Gambar 10.12.

Profitina dalam Praktik

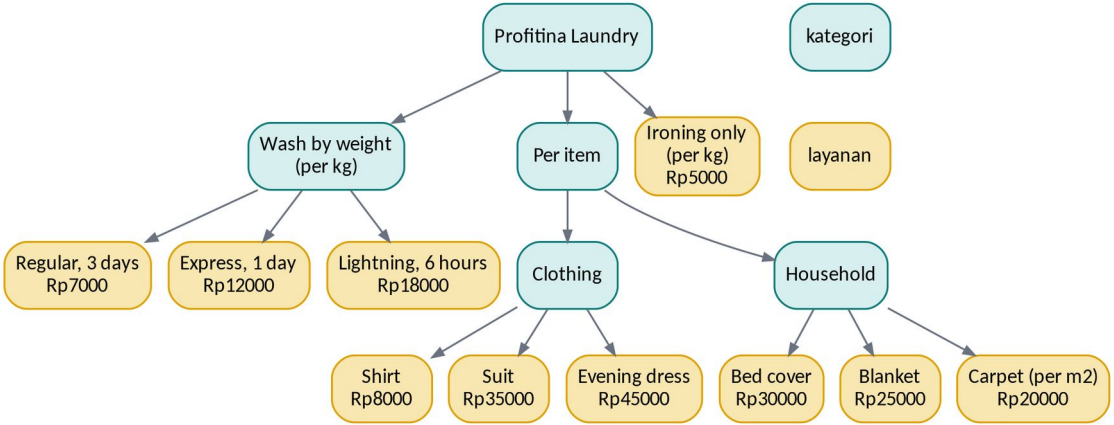
Menjelajahi situs: “kunjungi halaman, lalu setiap halaman yang ditautnya” bersifat rekursif, tetapi rantai tautan bisa ribuan aras dalamnya; *crawler* menyimpan *stack* atau *queue*-nya sendiri, himpunan halaman yang sudah dilihat, dan batas kedalaman.

Data terstruktur bersarang: JSON-LD menyarangkan objek sampai kedalaman berapa pun; memeriksanya adalah penyusuran rekursif, dengan penjaga kedalaman karena masukannya datang dari luar.

Mengingat hasil: memoisasi dengan *cache* LRU berbatas menguntungkan tepat ketika submasalah yang sama benar-benar berulang.

Profitina adalah platform visibilitas AI (GEO) yang dibangun dengan FastAPI; kita melukiskan bentuk setiap masalah, bukan kode produknya. Buku: Subbab 10.14.

Menu Bersarang Profitina Laundry



Gambar 10.13 Menu layanan Profitina Laundry: kategori (hijau kebiruan) memuat layanan (kuning) dan kategori yang lebih kecil, sampai kedalaman berapa pun.

Gambar 10.13: kategori memuat layanan dan kategori yang lebih kecil, sampai kedalaman berapa pun: sebuah pohon, pokok bahasan Bab 11. Buku: Subbab 10.14 dan Gambar 10.13.

Menghitung Layanan

Kode 10.8 `countServices` dan `height` dari `12_laundry_menu.cpp`.

```
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;
}
```

```
services: 10 recursively, 10 with an explicit stack
levels below the top: 3
cheapest service: Ironing only (per kg), Rp5000
```

Kode 10.8: base case "sebuah layanan, tanpa anak", recursive case "sebuah kategori: gabungkan anak-anaknya". `countServicesIter` mendapatkan 10 yang sama dengan stack eksplisit. Buku: Subbab 10.14.

Kesalahan yang Sering Terjadi

Tanpa *base case*, atau *base case* yang terlompati; panggilan yang tidak mengecilkan masalah.

Berekursi atas *list* panjang: satu *frame* per simpul meluapkan setiap *stack* bawaan.

Submasalah yang bertumpang tindih (gunakan memoisasi), dan **memanggil dua kali yang seharusnya sekali** (`power(m, n/2) * power(m, n/2)`).

Memercayai *floating point*, melupakan luapan, bergantung pada *tail-call elimination*, menaikkan batas Python tanpa berpikir.

Membuang nilai kembalian: panggillah `head = removeAll(head, key)`, bukan `removeAll(head, key)`.

Ujilah masukan terkecil lebih dulu: 0, 1, 2, dan nilai negatif apa pun yang mungkin diterima fungsi itu. Buku: Subbab 10.15.

Ringkasan: Lima Hal untuk Diingat

1. *Base case*, *recursive case* yang benar-benar lebih kecil, dan bukti induksi yang memercayai panggilan yang lebih kecil.

2. Waktu adalah kerja semua panggilan; memori adalah tumpukan *frame* terdalam.

3. Submasalah yang bertumpang tindih meledak ($\Theta(\varphi^n)$ untuk Fibonacci); memoisasi atau *bottom-up* memperbaikinya. $2^n - 1$ milik Hanoi itu optimal.

4. Membagi dua memberi kedalaman logaritmik (Euklides, pemangkatan cepat, pencarian biner); mengupas memberi kedalaman n.

5. *Tail call* dihapus hanya oleh *compiler* C++ yang mengoptimasi, tanpa jaminan; rekursi yang dalam menjadi gelung dengan *stack* eksplisit.

Buku: Subbab 10.16.

Pertemuan Berikutnya

Bab 11 — Tree dan Binary Search Tree. Akar dengan pohon-pohon yang lebih kecil di bawahnya: penyusuran, pencarian, penyisipan, dan penghapusan, semuanya ditulis secara rekursif.

Penyusuran *inorder* dari pohon Hanoi kembali, kali ini sebagai alat.

Sebelum itu: kerjakan Soal Latihan 10.3, 10.4, 10.9, dan 10.10, dalam C++ lalu dalam Java atau Python.

Penyelesaiannya, lengkap dengan program, ada di buku. Buku: Subbab 10.17 dan 10.18.

Kuliah 11 — Tree dan Binary Search Tree

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 11

Dosen: Irfan Subakti — yifana@gmail.com

Hari ini kita meninggalkan garis lurus: setiap elemen boleh menunjuk beberapa elemen di bawahnya, dan pencarian dapat membuang seluruh subpohon pada setiap langkah. Buku: Bab 11, Subbab 11.1–11.14.

Cakupan Pertemuan Ini

1. **Pohon** — definisi, kosakata, notasi, dan dua representasi dalam memori.
2. **Pohon biner** — bentuk-bentuknya, berapa banyak simpul yang muat, seberapa rendah pohon dapat dibuat.
3. **Penelusuran** — *preorder*, *inorder*, *postorder*, urutan tingkat, tanpa rekursi, dan pohon ekspresi.
4. ***Binary search tree*** — sisip, cari, minimum dan maksimum, *successor*, penghapusan dalam tiga kasus, dan ongkosnya.
5. **Bentuk dan keseimbangan** — tinggi yang terukur, rotasi, pohon AVL dan *red-black*, pohon umum, dan pustaka baku.

Contoh berjalannya adalah `insert(65, 5, 70, 3, 10)`*: dibangun, dicari, dihitung, dicetak, dan dihapus isinya. Buku: Subbab 11.1.*

Titik Tolak: Dilema Garis Lurus

Setiap struktur sejauh ini berupa **garis**: larik, *stack* dan *queue*, *single* dan *double linked list*.

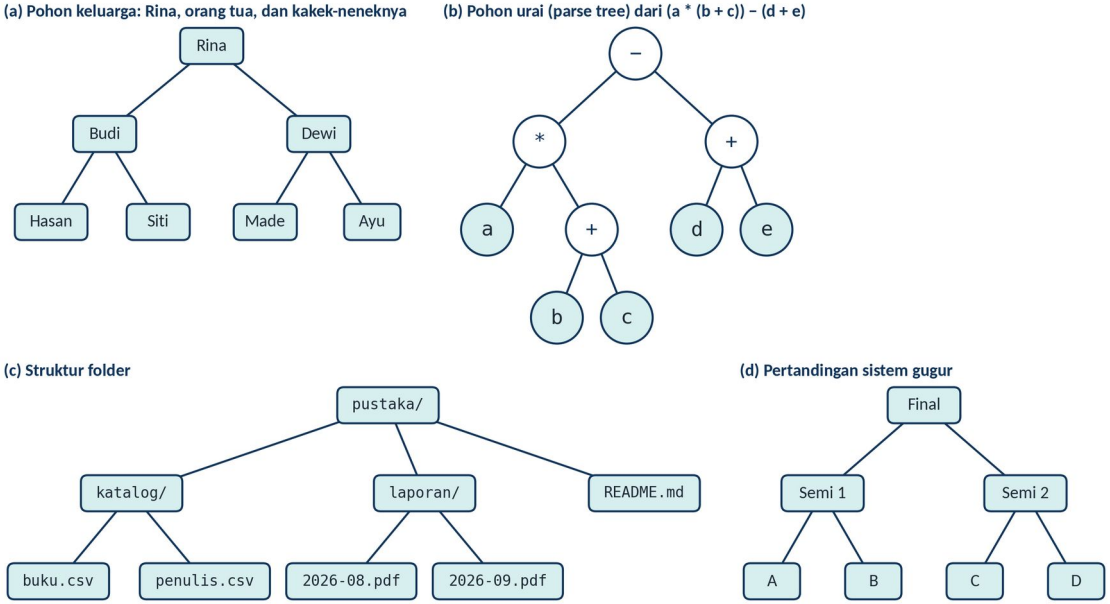
Larik terurut mencari dalam $O(\lg n)$ tetapi menggeser separuh isinya pada setiap penyisipan.

Linked list menyisip dalam $O(1)$ pada posisi yang dipegang, tetapi menemukan posisi itu dalam $\Theta(n)$.

Kita menginginkan keduanya: semudah larik terurut untuk dicari, semudah *list* untuk diubah. Pohon mencapainya dengan meninggalkan garis lurus.

Bentuknya secara alami rekursif, itulah sebabnya Bab 10 didahulukan: pohon adalah akar dengan pohon-pohon yang lebih kecil tergantung di bawahnya. Buku: Subbab 11.1.

Empat Pohon dari Kehidupan Sehari-hari



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.

Gambar 11.1: satu elemen di puncak, setiap elemen lain berada tepat di bawah satu elemen, dan tidak ada yang melingkar. Hanya itulah pohon. Buku: Subbab 11.2.1 dan Gambar 11.1.

Definisinya

DEFINISI. *Pohon* adalah himpunan berhingga T yang berisi satu *simpul* atau lebih sedemikian rupa sehingga (1) ada satu simpul khusus yang disebut *akar*, dan (2) simpul-simpul lainnya terbagi menjadi $m \geq 0$ himpunan saling lepas $T_1, \ldots, T_m$, yang masing-masing juga sebuah pohon: *subpohon* dari akar.

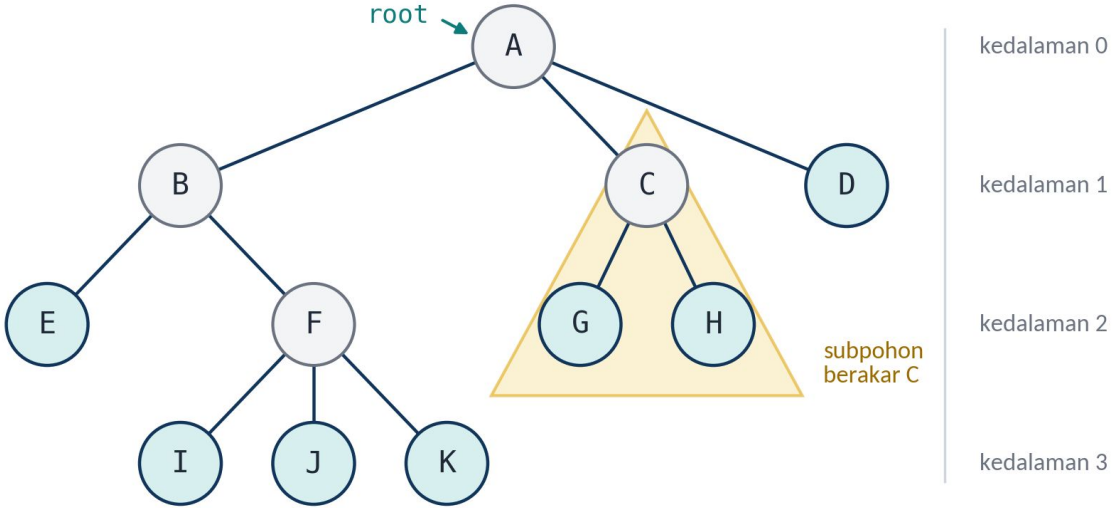
Definisinya **rekursif**, tetapi tidak berputar-putar: setiap subpohon punya lebih sedikit simpul. Dari akar ke setiap simpul ada **tepat satu lintasan**.

Pohon dengan n simpul punya Persamaan (11.1) sisi: setiap simpul kecuali akar punya tepat satu sisi ke atas, ke induknya.

$$e = n - 1 \tag{11.1}$$

Pemeriksaan cepat untuk setiap pohon yang Anda gambar: hitung simpul dan garisnya; garisnya harus kurang satu. Nol atau lebih pohon yang saling lepas membentuk hutan. Buku: Subbab 11.2.2 dan Persamaan (11.1).

Pohon Sebelas Simpul



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.

Gambar 11.2 mendefinisikan kosakatanya: akar, induk, anak, saudara, daun, simpul internal, lintasan, leluhur, keturunan, subpohon, kedalaman, tingkat, tinggi, derajat, ukuran. Buku: Subbab 11.2.3 dan Gambar 11.2.

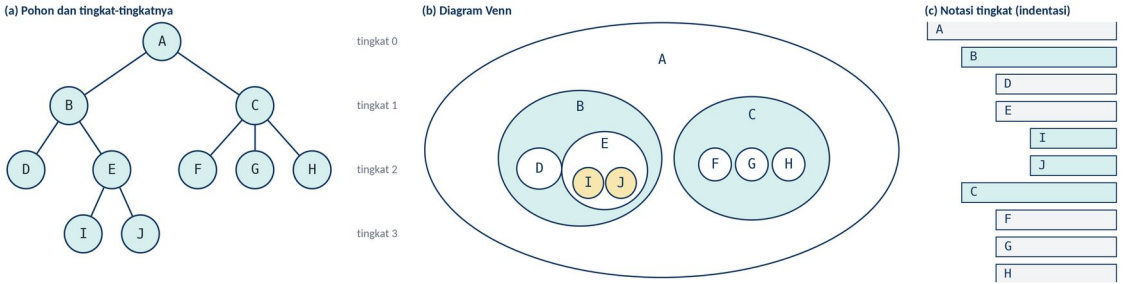
Kosakata, Dievaluasi

Tabel 11.1 Kosakata pohon, dievaluasi pada pohon Gambar 11.2.

Istilah	Nilai pada Gambar 11.2
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)

Tabel 11.1. Kedalaman diukur ke bawah dari akar; tinggi ke atas dari daun terendah. F berkedalaman 2 dan bertinggi 1; B berkedalaman 1 dan bertinggi 2. Pohon kosong bertinggi -1*. Buku: Subbab 11.2.3–11.2.4.*

Satu Pohon, Empat Notasi



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 11.3; notasi keempat, notasi kurung, menulis pohon yang sama sebagai `(A(B(D,E(I,J)),C(F,G,H)))`*, seperti cara LISP dan JSON menulis pohon. Buku: Subbab 11.3 dan Gambar 11.3.*

Representasi Statis: Larik

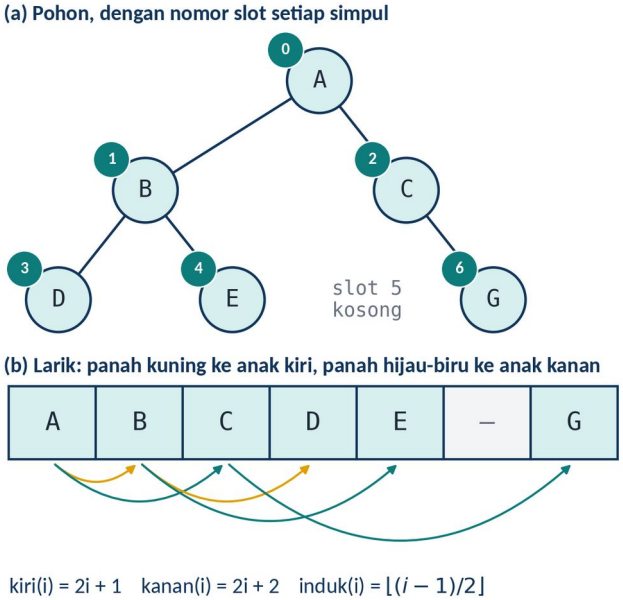
Beri nomor posisi tingkat demi tingkat mulai dari 0 dan simpan simpul i di slot i; anak dan induk diperoleh dari Persamaan (11.2):

$$left(i)=2i+1, right(i)=2i+2, parent(i)=\left\lfloor \frac{i-1}{2} \right\rfloor \tag{11.2}$$

Tanpa pointer, dan induk ditemukan dalam $O(1)$; tetapi setiap posisi yang kosong memboroskan satu slot. Rantai kanan $h+1$ simpul memerlukan $2^{h+1}-1$ slot.

Pohon complete sama sekali tidak punya slot kosong, itulah sebabnya Bab 14 menyimpan heap dengan cara ini. Buku: Subbab 11.4.1 dan Persamaan (11.2).

Pohon dalam Larik



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).

Gambar 11.4: C tidak punya anak kiri, sehingga slot $2 \cdot 2+1=5$ *kosong.* `array_tree.cpp` *menelusuri pohon secara preorder dan inorder hanya dengan aritmetika indeks. Buku: Subbab 11.4.1 dan Gambar 11.4.*

Representasi Dinamis: Simpul Bertaut

```
typedef struct MyNode {
    int data;
    struct MyNode *left;
    struct MyNode *right;
} Node;
```

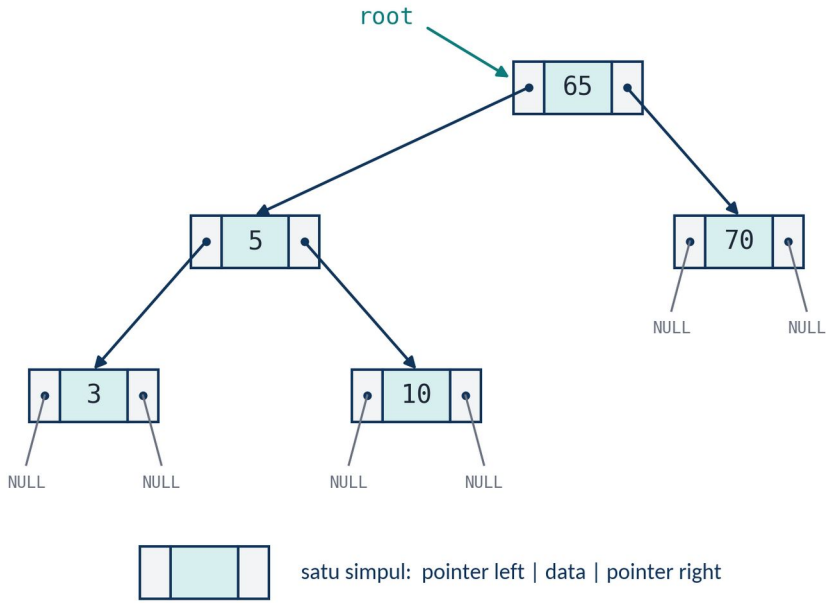
Simpul *double linked* dari Bab 9 dengan makna baru: `left` dan `right` menunjuk *ke bawah* ke dua subpohon; pohon dicapai lewat `Node *root`.

Dari $2n$ medan pointer hanya $n-1$ yang memuat sisi, sehingga Persamaan (11.3) di antaranya null:

$$2n-(n-1)=n+1 \tag{11.3}$$

Untuk banyak anak sembarang: larik induk, daftar anak (DOM halaman web), atau anak pertama dan saudara berikutnya (Subbab 11.10). Buku: Subbab 11.4.2–11.4.3 dan Persamaan (11.3).

Simpul Bertaut, Digambar



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.

Gambar 11.5: lima simpul, enam pointer NULL, sesuai rumusnya. Buku: Subbab 11.4.2 dan Gambar 11.5.

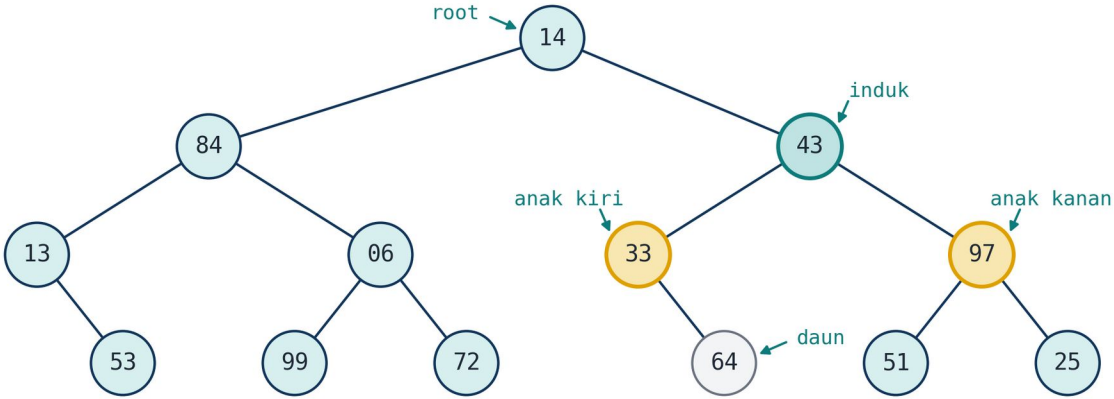
Pohon Biner dan Bentuk-Bentuknya

DEFINISI. *Pohon biner* adalah pohon yang *kosong*, atau sebuah simpul akar bersama dua pohon biner, *subpohon kiri* dan *subpohon kanan*, yang tidak berbagi simpul.

DEFINISI. ***Full***: setiap simpul punya 0 atau 2 anak. ***Perfect***: *full*, semua daun berkedalaman sama. ***Complete***: setiap tingkat penuh kecuali mungkin yang terakhir, yang diisi dari kiri. **Condong**: setiap simpul internal punya tepat satu anak.

Pohon biner boleh kosong, dan kiri serta kanan dibedakan. Teks lama memakai "full" untuk perfect dan "complete" untuk full; buku ini mengikuti Cormen dkk. Buku: Subbab 11.5.1–11.5.2.

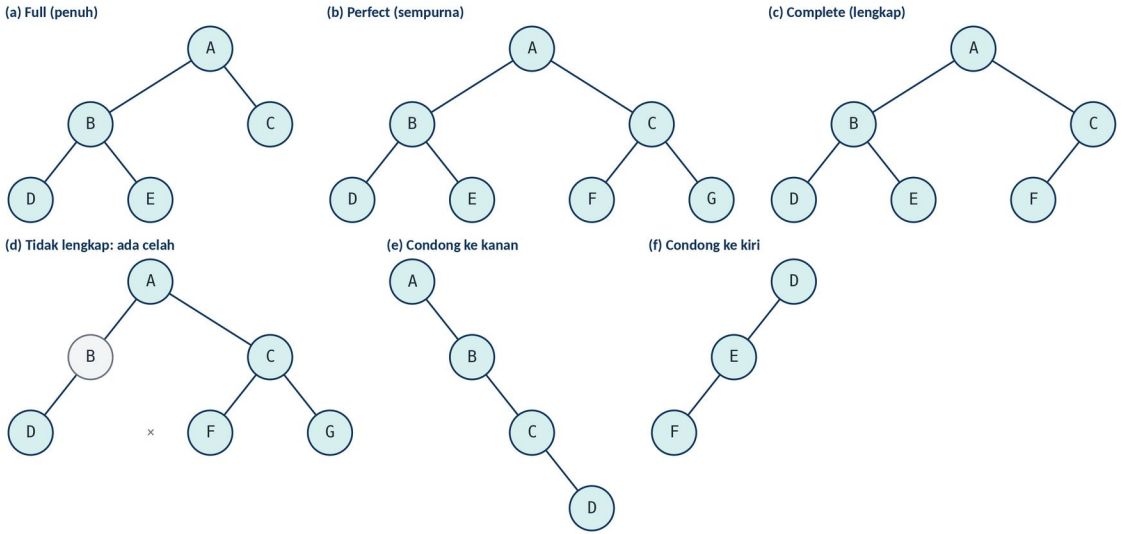
Bagian-Bagian Pohon Biner



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.

Gambar 11.6: dua anak per simpul persis yang dibutuhkan keputusan ya/tidak, dan pohon apa pun dapat disimpan sebagai pohon biner. Buku: Subbab 11.5.1 dan Gambar 11.6.

Enam Bentuk



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.

Gambar 11.7: setiap pohon perfect itu full dan complete; pohon condong n simpul adalah rantai bertinggi $n-1$, linked list yang digambar miring. Buku: Subbab 11.5.2 dan Gambar 11.7.

Berapa Simpul yang Muat, Seberapa Rendah Pohon

$$\text{simpul pada tingkat } \ell \le 2^{\ell} \tag{11.4}$$

$$n \le 1+2+4+\cdots+2^{h} = \sum_{\ell=0}^{h} 2^{\ell} = 2^{h+1}-1 \tag{11.5}$$

$$h \ge \lceil \lg(n+1) \rceil - 1 = \lfloor \lg n \rfloor \tag{11.6}$$

$$L = I + 1 \tag{11.7}$$

Persamaan (11.4) dan Persamaan (11.5) membatasi banyaknya simpul; dibalik, Persamaan (11.6) menyatakan tidak ada pohon biner n simpul yang lebih rendah dari $\lfloor \lg n \rfloor$: 19 untuk sejuta simpul, lawan 999.999 untuk rantai. Menurut Persamaan (11.7), sistem gugur 64 tim memerlukan 63 pertandingan. Buku: Subbab 11.5.3.

Tiga Urutan Telusur-Dalam

Pseudokode 11.1 Tiga penelusuran telusur-dalam; ketiganya hanya berbeda pada baris tempat simpul itu sendiri dikunjungi.

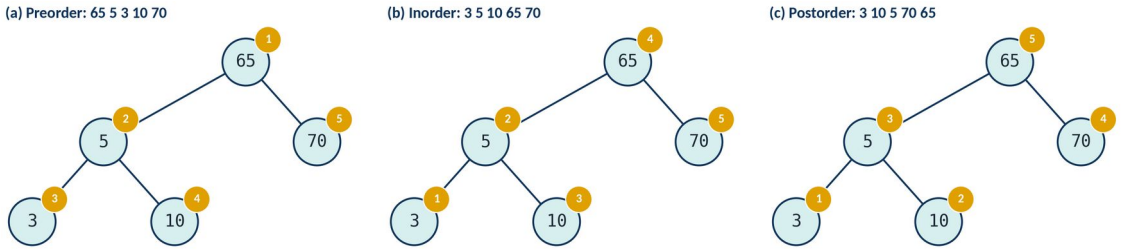
```
PREORDER(x):
 1.  if x ≠ NIL
 2.      visit x
 3.      PREORDER(x.left)
 4.      PREORDER(x.right)

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
```

Pseudokode 11.1: pre, akar sebelum subpohonnya; in, di antaranya; post, sesudahnya. Base case-nya pohon kosong; setiap panggilan menerima pohon yang benar-benar lebih kecil. Buku: Subbab 11.6.1.

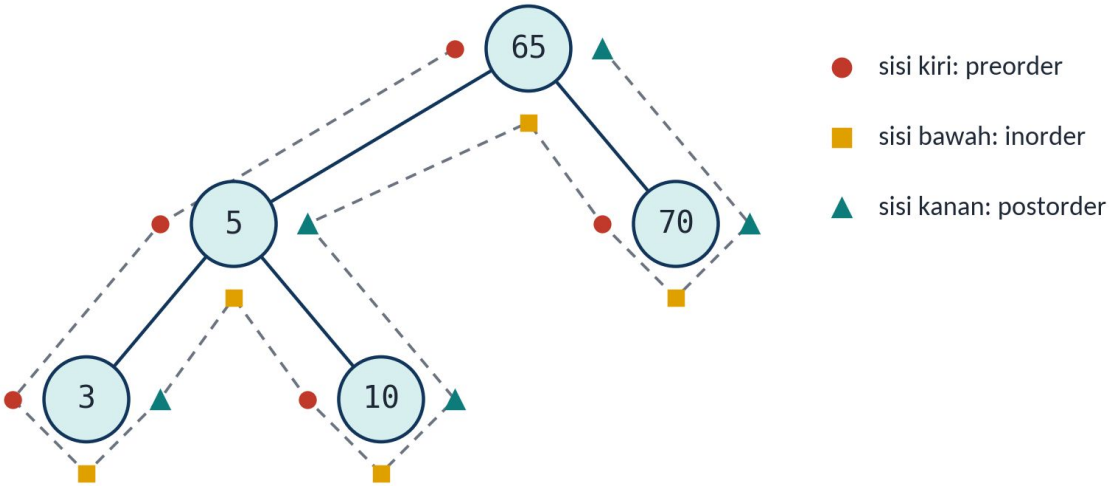
Contoh Terurai: Ketiga Urutan



Gambar 11.8 Urutan setiap penelusuran mengunjungi simpul-simpul pohon yang sama; lencana kuning pada sebuah simpul adalah posisinya dalam barisan kunjungan.

Gambar 11.8: preorder 65, 5, 3, 10, 70; inorder 3, 5, 10, 65, 70 (terurut!); postorder 3, 10, 5, 70, 65. Preorder untuk menyalin dan menyimpan, inorder untuk mendaftar terurut, postorder untuk menghapus dan menghitung. Buku: Subbab 11.6.1 dan Gambar 11.8.

Berjalan Mengelilingi Pohon



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*.

Gambar 11.9: tur Euler melewati setiap simpul di sisi kirinya (preorder), di bawahnya (inorder), dan di sisi kanannya (postorder). Rekursinya adalah perjalanan ini. Buku: Subbab 11.6.2 dan Gambar 11.9.

Urutan Tingkat dengan Queue

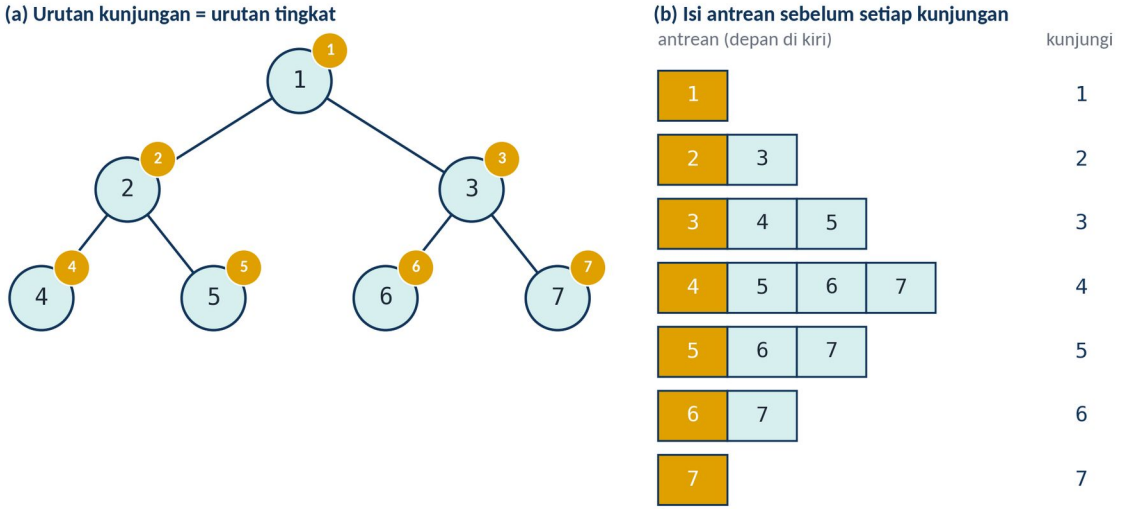
Pseudokode 11.2 `LEVEL-ORDER`: penelusuran telusur-lebar; anak-anak masuk di belakang *queue*, sehingga satu tingkat utuh dikunjungi sebelum simpul mana pun dari tingkat berikutnya.

```
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)
```

$$T(0)=c,T(n)=T(k)+T(n-k-1)+d(n\geq 1) \tag{11.8}$$

Pseudokode 11.2 bersifat telusur-lebar: anak-anak bergabung di belakang queue, sehingga satu tingkat utuh dikunjungi sebelum tingkat berikutnya. Menurut Persamaan (11.8) setiap penelusuran bernilai $\Theta(n)$ apa pun bentuknya; ruangnya $O(h)$ untuk telusur-dalam dan selebar tingkat terlebar untuk urutan tingkat. Buku: Subbab 11.6.3 dan 11.6.5.

Urutan Tingkat, Ditelusuri



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.

Gambar 11.10: [1], [2, 3], [3, 4, 5], [4, 5, 6, 7], … Queue menampung paling banyak dua tingkat berurutan; Bab 13 memperumum ini menjadi breadth-first search. Buku: Subbab 11.6.3 dan Gambar 11.10.

Inorder Tanpa Rekursi

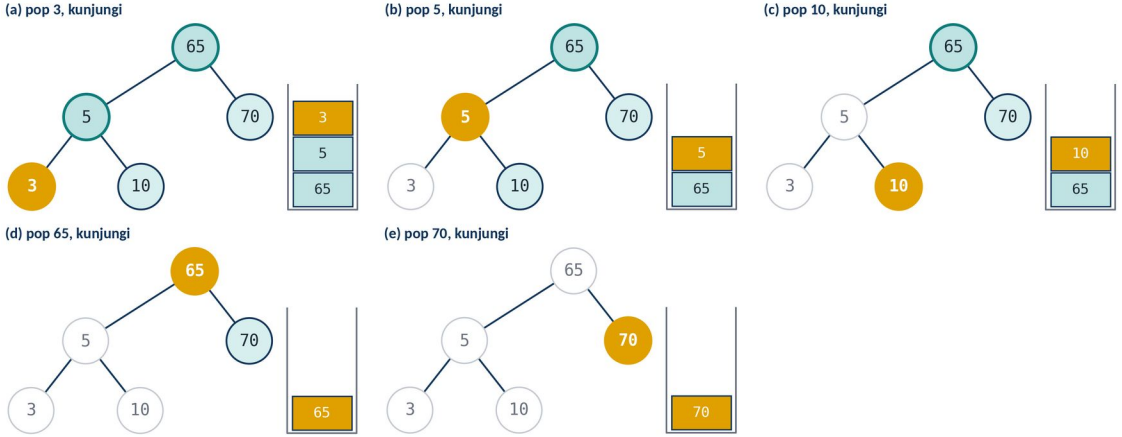
Pseudokode 11.3 Penelusuran *inorder* dengan *stack* eksplisit: turun ke kiri sambil mendorong setiap simpul, ambil dan kunjungi, lalu lanjutkan dengan subpohon kanan.

```
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
```

Turun ke kiri sambil mendorong setiap simpul; keluarkan dan kunjungi; lanjutkan ke subpohon kanan. *Preorder*: keluarkan, kunjungi, dorong kanan lalu kiri. *Postorder*: dua *stack*.

Pseudokode 11.3: stack eksplisit di heap tidak dapat meluapkan call stack pada pohon merosot, dan penelusuran iteratif dapat dijeda, begitulah cara kerja iterator `std::map`. Buku: Subbab 11.6.4.

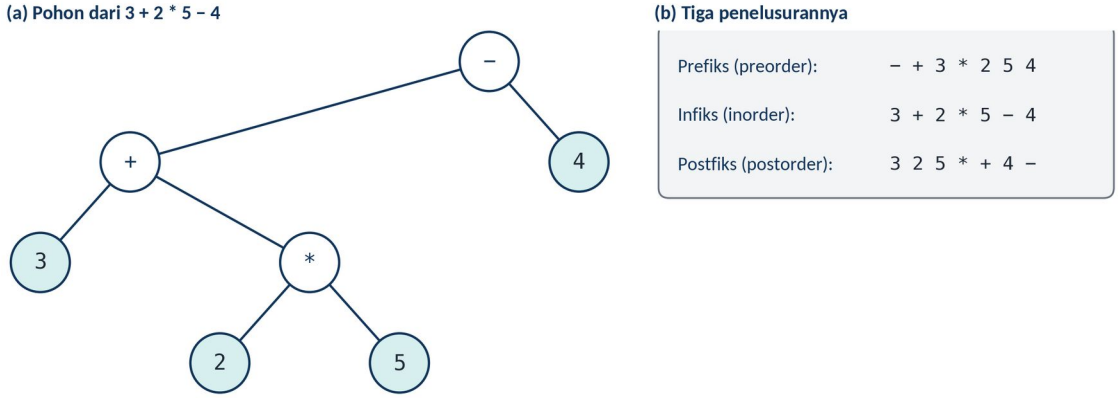
Stack pada Setiap Kunjungan



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.

Gambar 11.11: dorong 65, 5, 3; kunjungi 3; kunjungi 5; dorong dan kunjungi 10; kunjungi 65; dorong dan kunjungi 70. Buku: Subbab 11.6.4 dan Gambar 11.11.

Pohon Ekspresi



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.

Gambar 11.12: preorder `- + 3 * 2 5 4` *(prefiks), inorder* `3 + 2 * 5 - 4` *(infiks, perlu kurung), postorder* `3 2 5 * + 4 -` *(postfiks). Mengevaluasinya adalah penelusuran postorder. Buku: Subbab 11.6.6 dan Gambar 11.12.*

Membangun Pohon dari Postfiks

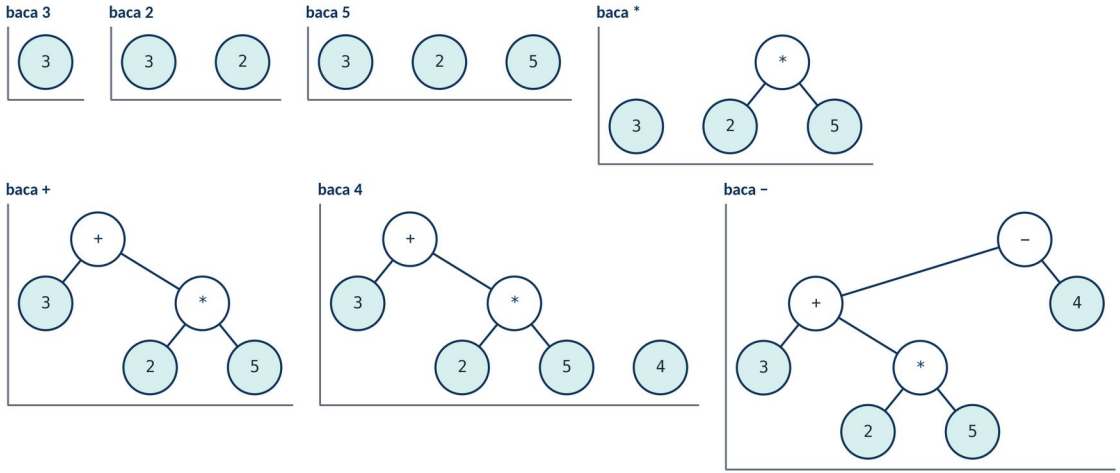
Pseudokode 11.4 Membangun pohon ekspresi dari notasi postfiks dengan *stack* berisi pohon.

```
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
```

Algoritma *stack* dari Subbab 5.9 dengan pohon di atas *stack*: operan mendorong pohon satu simpul; operator mengeluarkan operan kanan, lalu kiri, dan mendorong pohon gabungannya.

Pseudokode 11.4: `expr_tree.cpp` *mengevaluasi* `3 2 5 * + 4 -` *menjadi 9:* $2 \cdot 5 = 10$, $3 + 10 = 13$, $13 - 4 = 9$. *Compiler melakukan hal ini dalam skala besar. Buku: Subbab 11.6.6.*

Stack Berisi Pohon



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.

Gambar 11.13: di akhir tersisa tepat satu pohon, yaitu pohon ekspresinya. Buku: Subbab 11.6.6 dan Gambar 11.13.

Binary Search Tree

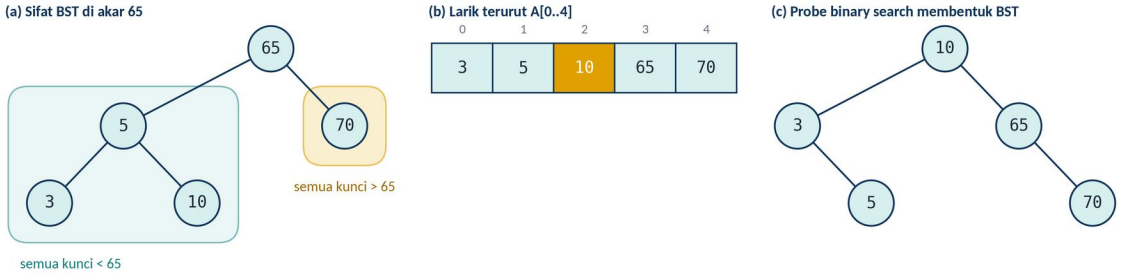
DEFINISI. *Binary search tree* (BST) adalah pohon biner yang, untuk setiap simpul x, setiap kunci di subpohon kiri x lebih kecil daripada $x.key$ dan setiap kunci di subpohon kanannya lebih besar.

Setiap kunci di subpohon, bukan hanya anaknya; dan pada *setiap* simpul, bukan hanya akar.

INORDER ITU TERURUT. Dengan induksi: subpohon kiri menaik, lalu akar, lalu subpohon kanan menaik, dan sifat BST mengurutkan ketiga bagian itu.

BST adalah kamus yang tetap terurut: ia mendaftar kunci secara berurutan, melaporkan minimum dan maksimum, dan menemukan tetangga, hal yang tidak dapat dilakukan tabel hash. Buku: Subbab 11.7.1.

Pencarian Biner, Dikeluarkan dari Larik



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*.

Gambar 11.14: gambarlah setiap urutan penyelidikan pencarian biner dan Anda mendapatkan BST. Windley, Booth dan Colin, serta Hibbard menemukannya secara terpisah sekitar 1960. Buku: Subbab 11.7.1 dan Gambar 11.14.

Penyisipan

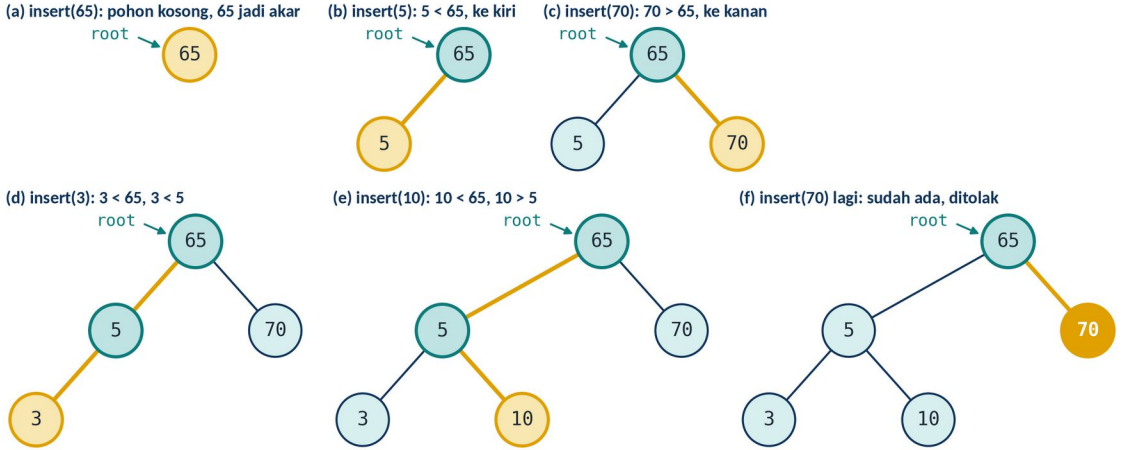
Pseudokode 11.5 `TREE-INSERT`: turun seperti pada pencarian dan gantungkan simpul baru pada subpohon kosong tempat perjalanan berakhir.

```
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"
```

Turun seperti mencari; gantungkan simpul baru pada subpohon kosong tempat perjalanan berakhir. Baris 2 harus mengubah *pointer* yang tadinya NIL.

Pseudokode 11.5: C memakai `Node **root`*, C++ boleh memakai* `Node *&root`*, sedangkan Java dan Python mengembalikan akar subpohon yang baru:* `t.left = insert(t.left, z)`*. Ongkosnya* $O(h)$*. Buku: Subbab 11.7.2.*

Contoh Terurai: `insert(65, 5, 70, 3, 10)`



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.

Gambar 11.15: 65 menjadi akar; 5 ke kiri; 70 ke kanan; 3 kiri, kiri; 10 kiri, kanan; 70 kedua ditemukan dan ditolak. Bentuknya merekam riwayat penyisipan. Buku: Subbab 11.7.2 dan Gambar 11.15.

Penyisipan dalam C++

Kode 11.1 `insert` dari `bst.cpp`: alamat sebuah pointer, sehingga panggilan dapat mengubah pointer milik pemanggil.

```
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);
    }
}
```

Kode 11.1: `&(*root)->left` adalah alamat medan kiri simpul yang sedang dikunjungi. `bst.py` dan `Bst.java` sebagai gantinya mengembalikan akar subpohon yang baru: `t.left = insert(t.left, new_data)`. Buku: Subbab 11.7.7–11.7.8.

Pencarian

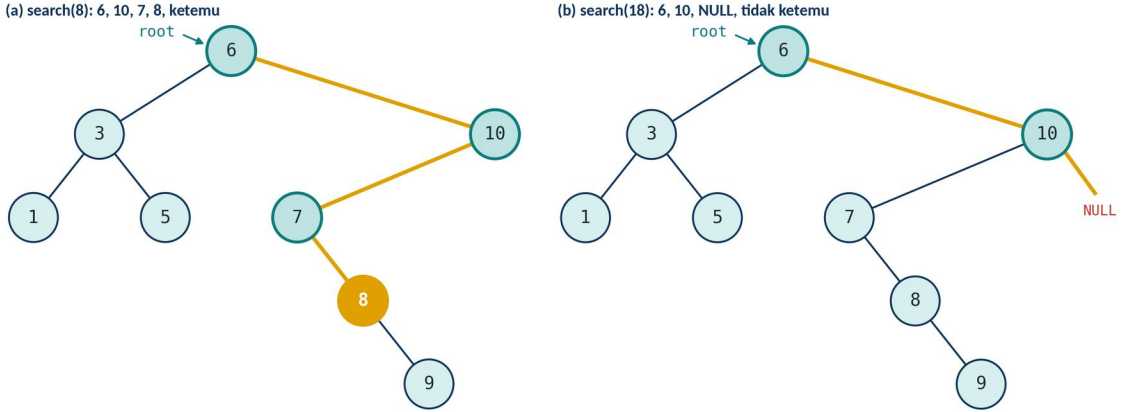
Pseudokode 11.6 Pencarian pada *binary search tree*, secara rekursif dan iteratif; keduanya mengembalikan simpul yang memuat k, atau NIL.

```
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
```

Pseudokode 11.6: sama, ketemu; lebih kecil, ke kiri; lebih besar, ke kanan; NIL, tidak ada. Bentuk iteratifnya tidak memerlukan stack, sebab ia tidak pernah kembali. Ongkosnya $O(h)$. Buku: Subbab 11.7.3.

Contoh Terurai: Dua Pencarian



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.

Gambar 11.16: `search(8)` membandingkan dengan 6, 10, 7, 8: empat perbandingan dalam delapan simpul; `search(18)` membuktikan ketiadaan sesudah dua. Buku: Subbab 11.7.3 dan Gambar 11.16.

Minimum, Maksimum, Tinggi

Pseudokode 11.7 Minimum adalah simpul paling kiri dan maksimum adalah simpul paling kanan dari pohon yang tidak kosong.

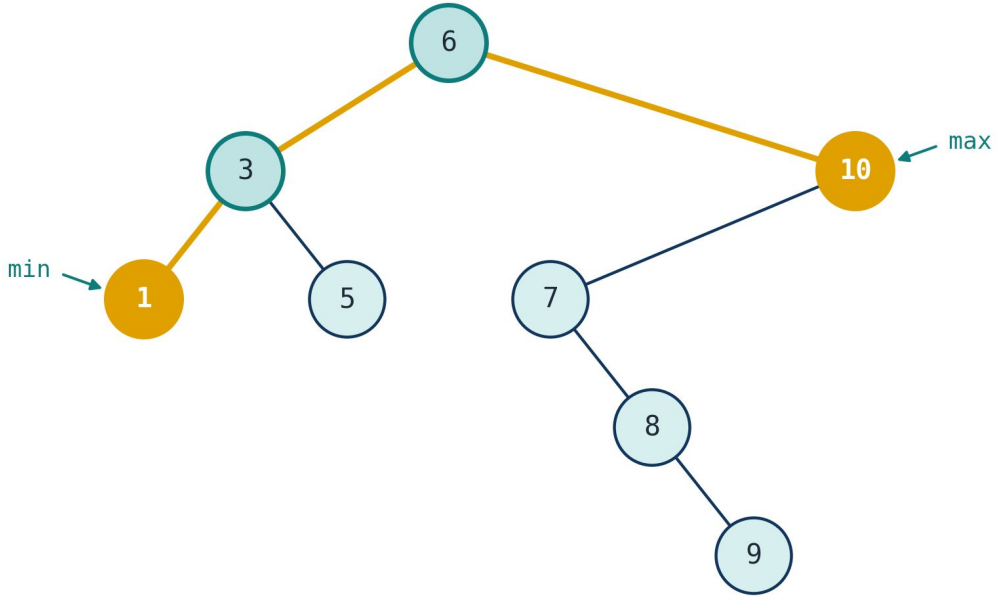
```
TREE-MINIMUM(x):
 1.  while x.left ≠ NIL
 2.      x ← x.left
 3.  return x
```

```
TREE-MAXIMUM(x):
 1.  while x.right ≠ NIL
 2.      x ← x.right
 3.  return x
```

$$height(\text{NIL}) = -1, height(x) = 1 + \max(height(x.left), height(x.right)) \tag{11.9}$$

Pseudokode 11.7: simpul paling kiri dan paling kanan, $O(h)$. Persamaan (11.9) memberi tinggi 2 untuk contoh berjalan; menghitung simpul dan daun berpola postorder yang sama, $\Theta(n)$. Mencetak menyamping adalah inorder terbalik. Buku: Subbab 11.7.4–11.7.6.

Paling Kiri dan Paling Kanan



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.

Gambar 11.17: maksimumnya, 10, bukan daun; ia hanya tidak punya anak kanan. Buku: Subbab 11.7.4 dan Gambar 11.17.

Successor: Dua Kasus

Subpohon kanan ada: *successor*-nya adalah minimum subpohon itu.

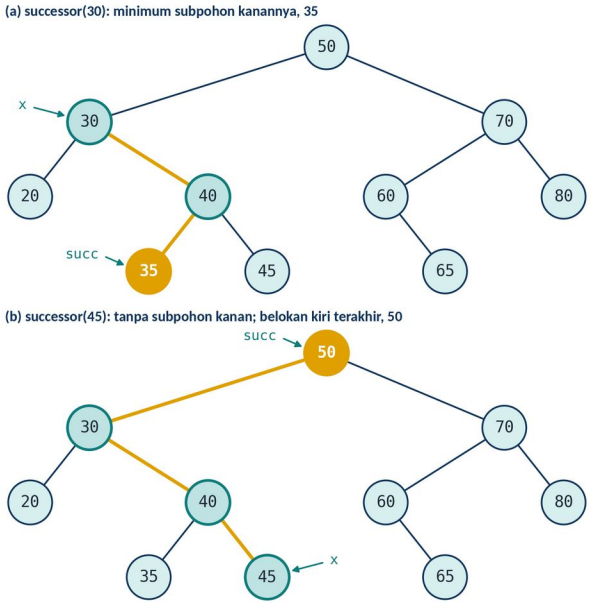
Tanpa subpohon kanan: *successor*-nya adalah simpul terakhir tempat lintasan pencarian dari akar *berbelok ke kiri*.

Pseudokode 11.8 `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)
```

Pseudokode 11.8 turun sambil mengingat belokan kiri terakhir, sebab simpul kita tidak punya pointer induk. Predecessor adalah cerminnya. Menyusuri seluruh pohon lewat successor berongkos $\Theta(n)$: itulah `++it` pada `std::map`. Buku: Subbab 11.7.9.

Kedua Kasus, Digambar



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.

Gambar 11.18: inorder 20, 30, 35, 40, 45, 50, 60, 65, 70, 80; successor(30) = 35, successor(45) = 50. Buku: Subbab 11.7.9 dan Gambar 11.18.

Penghapusan: Tiga Kasus

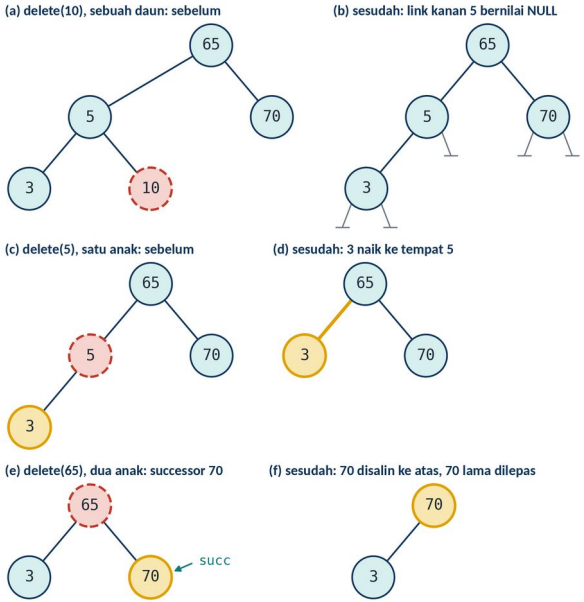
Kasus 1, daun: jadikan pointer induknya NIL dan lepaskan simpulnya.

Kasus 2, satu anak: sambung lewati simpul itu; anaknya naik bersama seluruh subpohonnya.

Kasus 3, dua anak (Hibbard): salin kunci *successor* ke simpul itu, lalu hapus *successor* dari subpohon kanan; ia tidak punya anak kiri, jadi itu kasus 1 atau 2.

Sifat BST bertahan di setiap kasus; barisan inorder hanya kehilangan kunci yang dihapus. Buku: Subbab 11.7.10.

Contoh Terurai: Menghapus 10, 5, 65



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).

Gambar 11.19: kasus 1, 2, dan 3 berturut-turut; untuk 65 kunci 70 disalin ke atas. Buku: Subbab 11.7.10 dan Gambar 11.19.

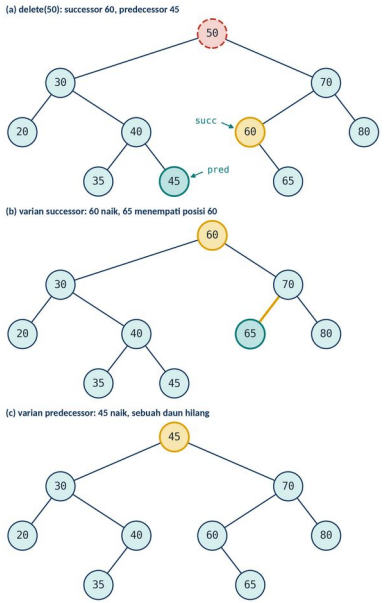
TREE-DELETE

Pseudokode 11.9 `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
```

Pseudokode 11.9 mengembalikan akar subpohon sesudah penghapusan, dan pemanggil menyimpannya: tidak ada kasus khusus untuk akar. Pada kasus 3 yang berpindah adalah kunci, bukan simpul; pustaka menaut ulang simpulnya. Buku: Subbab 11.7.10 dan 11.7.12.

Successor atau Predecessor?



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).

Gambar 11.20: kedua hasilnya benar; bergantianlah di antara keduanya. Buku: Subbab 11.7.11 dan Gambar 11.20.

Setiap Operasi Berongkos $O(h)$

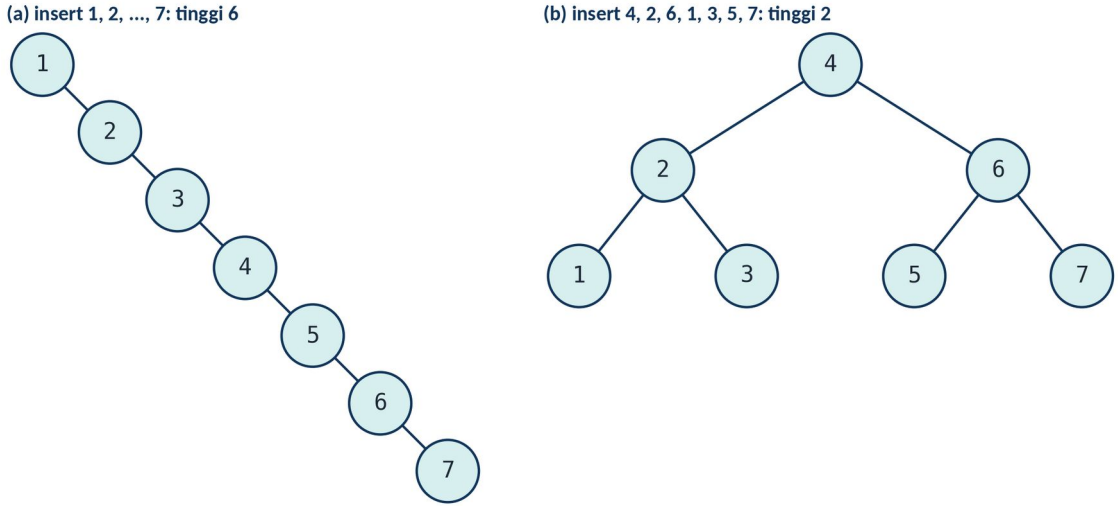
$$T_{\text{cari}}, T_{\text{sisip}}, T_{\text{hapus}}, T_{\min}, T_{\max}, T_{\text{succ}}, T_{\text{pred}} = O(h) \tag{11.10}$$

Tabel 11.2 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 11.9)	$O(\lg n)$	$O(n)$
minimum, maksimum	$O(h)$ (Pseudokode 11.7)	$O(\lg n)$	$O(n)$
successor, *predecessor*	$O(h)$ (Pseudokode 11.8)	$O(\lg n)$	$O(n)$
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	$O(n \lg n)$	$\Theta(n^2)$

Persamaan (11.10) dan Tabel 11.2: BST sebaik pencarian biner bila tingginya logaritmik, dan tidak lebih baik daripada linked list bila tingginya linear. Buku: Subbab 11.7.13.

Urutan Penyisipan Menentukan Bentuk



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.

Gambar 11.21: masukan terurut membangun rantai bertinggi 6; 4, 2, 6, 1, 3, 5, 7 membangun pohon perfect bertinggi 2. Menurut Persamaan (11.11), 64 dari 5.040 urutan tujuh kunci menghasilkan rantai. Buku: Subbab 11.8.1 dan Gambar 11.21.

Tinggi yang Terukur

$$\#\{\text{urutan bertinggi } n-1\} = 2^{n-1} \tag{11.11}$$

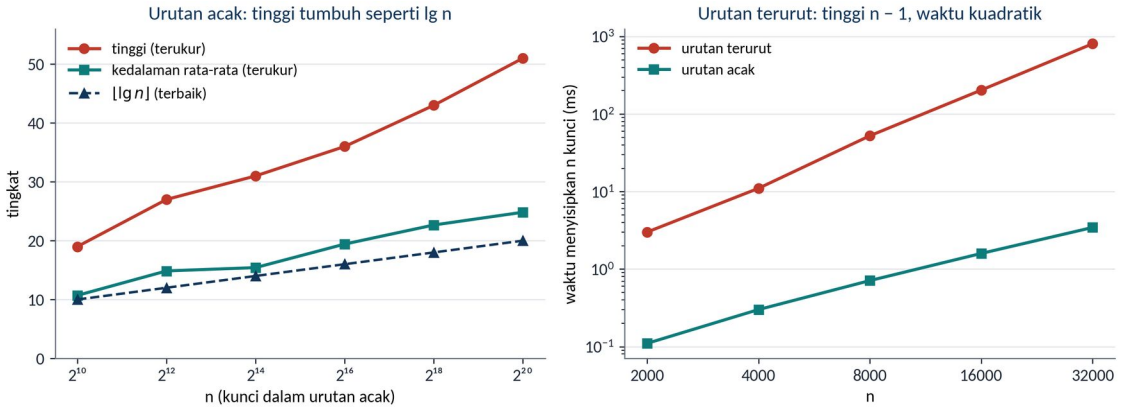
$$\acute{d}(n) = 2\left(1+\frac{1}{n}\right)H_n - 4 \approx 2\ln n \approx 1.39 \lg n \tag{11.12}$$

Urutan acak: sejuta kunci menghasilkan tinggi 51 dan kedalaman rata-rata 24,83, sesuai Persamaan (11.12).

Urutan terurut: tinggi tepat $n-1$, dan waktu pembangunan naik empat kali setiap penggandaan: 806 ms untuk 32.000 kunci, lawan 3,44 ms secara acak.

Masukan terurut adalah kasus terburuk dan, dalam praktik, sering terjadi: rekaman datang menurut urutan pembuatannya. Dalam Python pembangunan terurut 32.000 kunci memakan sekitar 16 detik. Buku: Subbab 11.8.2 dan Persamaan (11.12).

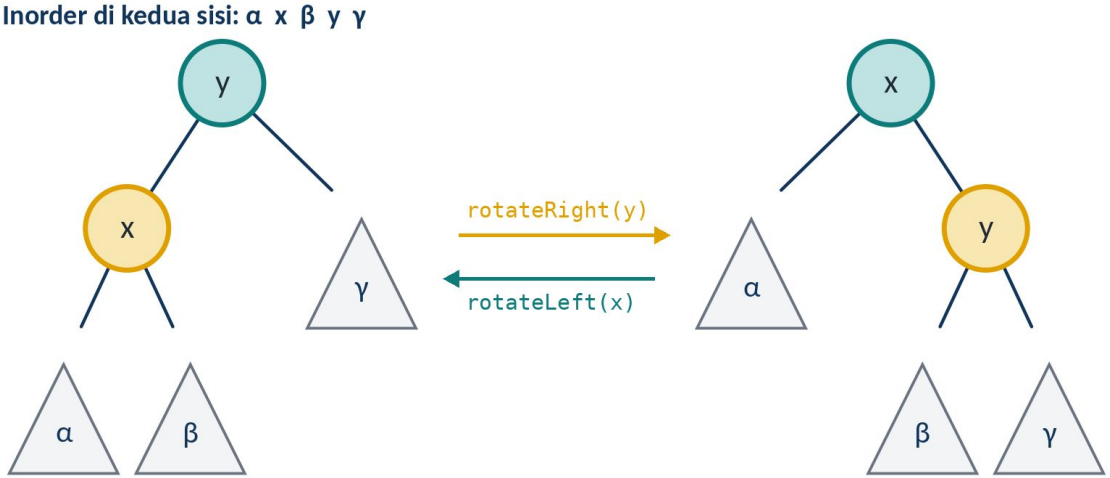
Acak lawan Terurut, Diplot



Gambar 11.22 Hasil pengukuran: dalam urutan acak tinggi dan kedalaman rata-rata tumbuh seiring $\lg n$ (kiri); dalam urutan terurut pohonnya berupa rantai, dan waktu pembangunannya naik empat kali setiap penggandaan n (kanan, skala logaritmik).

Gambar 11.22: tingginya ditentukan oleh seed; hanya waktunya yang bergantung pada mesin. Buku: Subbab 11.8.2 dan Gambar 11.22.

Rotasi



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.

Gambar 11.23: tiga pointer berubah, waktunya $O(1)$, dan barisan inorder $\alpha, x, \beta, y, \gamma$ tidak berubah, sehingga sifat BST bertahan. Buku: Subbab 11.9.1 dan Gambar 11.23.

Pohon AVL

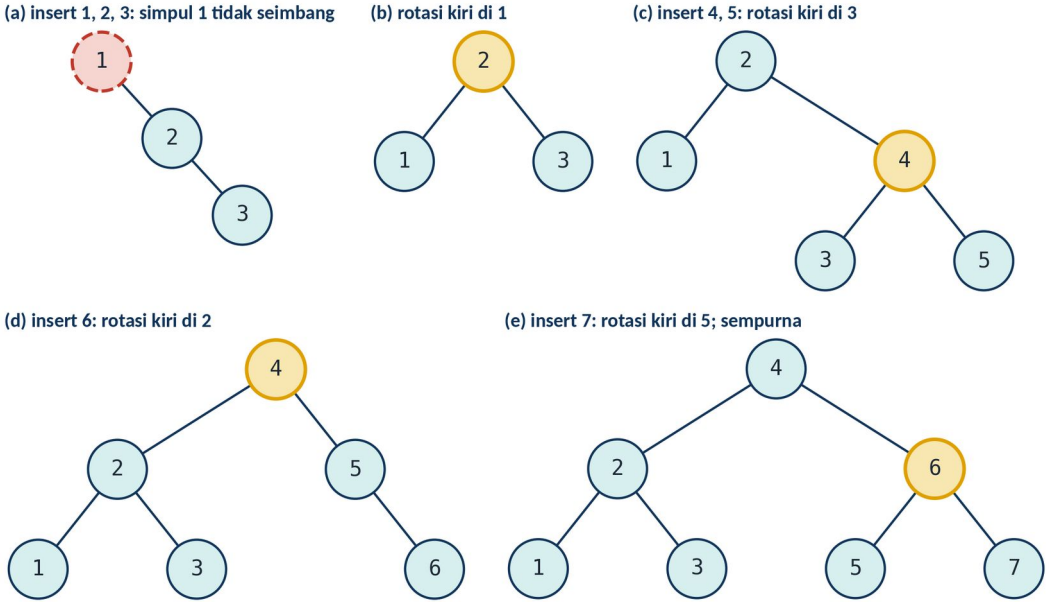
DEFINISI. Pohon biner *seimbang-tinggi* bila, pada setiap simpul, tinggi kedua subpohonnya berselisih paling banyak 1: *faktor keseimbangan*-nya -1, 0, atau $+1$ di mana-mana.

AVL (1962) menjaga setiap simpul seimbang-tinggi, memperbaiki faktor ± 2 dengan satu rotasi, atau dua bila bagian yang dalam berada di sisi dalam.

$$h < 1.4405 \lg(n+2) - 0.3277 \tag{11.13}$$

Pohon AVL paling jarang bertinggi h *punya* $N(h) = N(h-1) + N(h-2) + 1$ *simpul, rekurens Fibonacci dari Bab 10, sehingga berlaku Persamaan (11.13): tidak pernah lebih dari 44% lebih tinggi daripada pohon terbaik. Sejuta kunci terurut: tinggi 19. Buku: Subbab 11.8.3 dan 11.9.2.*

Kunci Terurut ke dalam Pohon AVL



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*.

Gambar 11.24: empat rotasi kiri, dan 1 sampai 7 berakhir sebagai pohon perfect; `avl_first_look.cpp` *mencetak rotasi yang sama. Buku: Subbab 11.9.2 dan Gambar 11.24.*

Pohon Red-Black

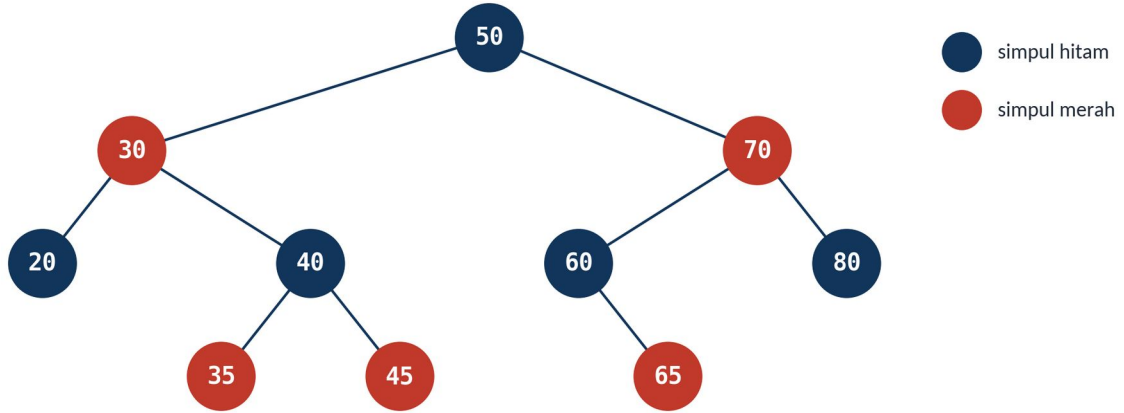
1. Setiap simpul merah atau hitam. **2.** Akar hitam. **3.** Setiap daun NIL hitam. **4.** Simpul merah punya dua anak hitam. **5.** Setiap lintasan dari sebuah simpul ke NIL memuat banyak simpul hitam yang sama.

$$h \leq 2 \lg (n+1) \tag{11.14}$$

Lebih lemah daripada AVL, tetapi paling banyak 2 rotasi per penyisipan dan 3 per penghapusan: dipakai `std::map`, `std::set`, `TreeMap`, `TreeSet`, dan kernel Linux.

Aturan 4 dan 5 membuat lintasan terpanjang paling banyak dua kali yang terpendek, sehingga berlaku Persamaan (11.14). Pohon AVL, yang lebih rendah, cocok untuk kerja yang didominasi pencarian. Buku: Subbab 11.9.3 dan Persamaan (11.14).

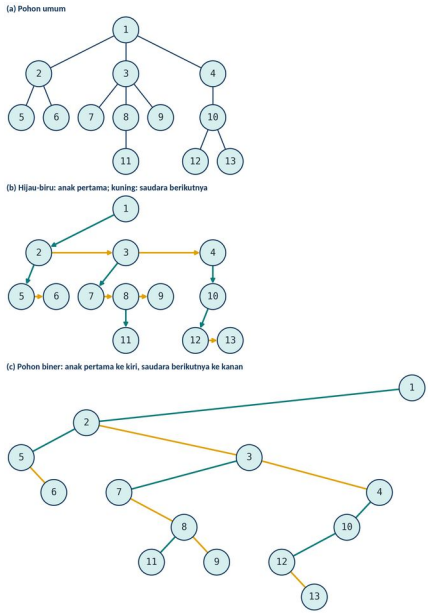
Pewarnaan Red-Black



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.

Gambar 11.25: setiap lintasan akar-ke-NIL melewati tepat dua simpul hitam, dan tidak ada simpul merah beranak merah. B-tree, splay tree, dan treap melengkapi keluarganya. Buku: Subbab 11.9.3–11.9.4 dan Gambar 11.25.

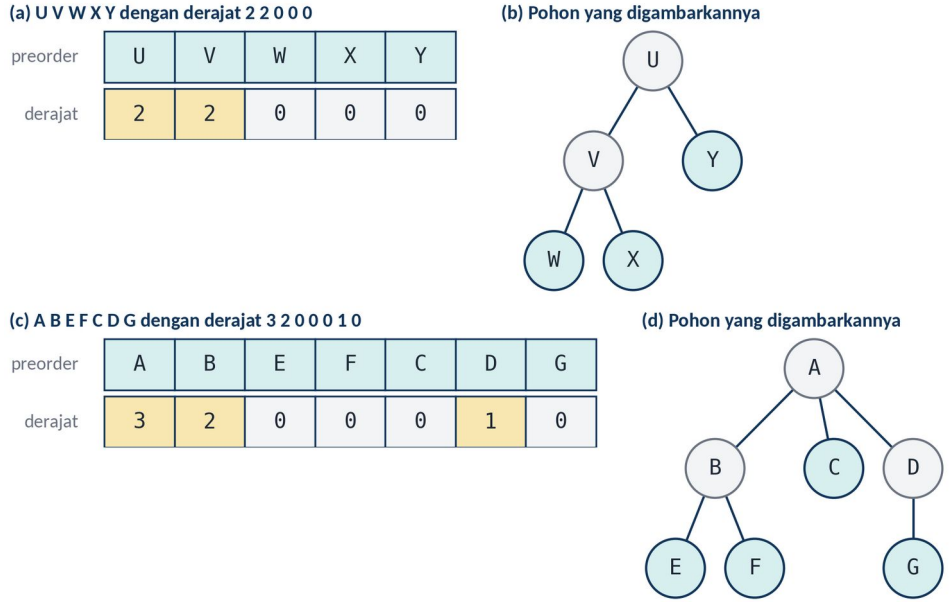
Anak Pertama, Saudara Berikutnya



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.

Gambar 11.26: postorder umum = inorder biner. Buku: Subbab 11.10.1 dan Gambar 11.26.

Contoh Terurai: Preorder dan Derajat



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).

Gambar 11.27: pindai dari kanan ke kiri; simpul berderajat d mengambil d subpohon di kanannya. U V W X Y dengan 2 2 0 0 0 menghasilkan U(V(W,X),Y); jumlah derajatnya selalu $n-1$. Buku: Subbab 11.10.2–11.10.3 dan Gambar 11.27.

Pohon Pencarian dalam Pustaka Baku

Tabel 11.3 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`

Tabel 11.3: mengiterasi `std::map` adalah perjalanan inorder; `lower_bound` dan `subMap` menjawab kueri rentang. Python tidak punya pohon pencarian bawaan: list terurut dengan `bisect`, atau `sortedcontainers`. Buku: Subbab 11.11.

Profitina dalam Praktik

Halaman web adalah pohon (DOM); urutan dokumen adalah *preorder*-nya. **Judul-judul membentuk kerangka**; memeriksanya adalah perjalanan dengan *stack* judul yang terbuka.

Data terstruktur adalah pohon (JSON-LD). **Situs adalah pohon tautan**: urutan tingkat memberi setiap halaman kedalaman *crawl*-nya.

Data terurut tinggal di *B-tree*: "pemindaian antara Juli dan September" adalah kueri rentang pada indeks *B-tree* PostgreSQL.

Kode 11.2 Kueri rentang terpangkas dari `laundry_tree.cpp`, papan pengambilan Profitina Laundry yang berkunci waktu siap.

```
// 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);
}
```

Kode 11.2 turun ke kiri hanya bila `lo` lebih kecil daripada waktu simpul dan ke kanan hanya bila waktunya lebih kecil daripada `hi`. Buku: Subbab 11.12.

Papan Pengambilan



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.

Gambar 11.28: kueri itu mengunjungi 5 dari 12 simpul. Tingginya 7, bukan 3: waktu siap datang hampir terurut, jadi sistem sungguhan akan memakai `std::map` *atau* `TreeMap`*. Buku: Subbab 11.12 dan Gambar 11.28.*

Kesalahan yang Sering Terjadi

Mencampuradukkan kedalaman dan tinggi; **memeriksa sifat BST hanya antara induk dan anak** (10(5(2, 12), 15) bukan BST).

Kehilangan perubahan pointer: `void insert(Node *root, int x)` mengubah salinan; lupa `root = insert(root, x)` di Java atau Python.

Menyambung-lewati simpul beranak dua; **mendereferensi NIL**; **melepas dengan urutan yang salah** (hapus anak dulu: *postorder*).

Menganggap $O(\lg n)$ untuk BST biasa, dan **berekursi pada pohon yang mungkin merosot**.

Juga: memakai penelusuran yang salah untuk tugasnya, dan mencampuradukkan nama-nama bentuk. Buku: Subbab 11.13.

Ringkasan: Lima Hal untuk Diingat

1. Pohon adalah akar dengan subpohon yang saling lepas; n simpul, $n-1$ sisi; pohon biner bertinggi paling sedikit $\lfloor \lg n \rfloor$.

2. *Preorder*, *inorder*, *postorder*, dan urutan tingkat semuanya $\Theta(n)$; postfiks membangun pohon ekspresi dengan *stack*.

3. Pada BST, *inorder* itu terurut; sisip, cari, minimum, *successor*, dan hapus semuanya berongkos $O(h)$.

4. Penghapusan: lepas daun, sambung-lewati satu anak, pinjam kunci *successor* atau *predecessor* untuk dua anak.

5. Masukan terurut membuat rantai; rotasi menjaga pohon AVL dan *red-black* tetap logaritmik, dan pustaka baku memakai *red-black*.

Buku: Subbab 11.14.

Pertemuan Berikutnya

Bab 12 — Evaluasi dan Pembahasan III. Rekursi bekerja pada *list* dan pohon: soal-soal diselesaikan dari halaman kosong, dalam C++, Java, dan Python.

Pohon yang dibangun persis seperti hari ini, diperiksa oleh program.

Sebelum itu: kerjakan Soal Latihan 11.1, 11.5, 11.8, dan 11.10.

Penyelesaiannya, lengkap dengan program, ada di buku. Buku: Subbab 11.15 dan 11.16.

Kuliah 12 — Evaluasi dan Pembahasan III: Rekursi Bekerja pada List dan Pohon

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 12

Dosen: Irfan Subakti — yifana@gmail.com

Tidak ada struktur baru hari ini: enam belas soal tentang list dan pohon, masing-masing diselesaikan dari halaman kosong, diperiksa oleh sebuah program, dan disandingkan dengan penyelesaian keliru yang menggoda. Buku: Bab 12, Subbab 12.1–12.20.

Cakupan Pertemuan Ini

1. **Sebuah metode** — tujuh langkah untuk soal rekursif pada struktur bertaut, dan templat fungsi pohon.
2. **Soal 12.1–12.6** — urutan *depth first*, satu tingkat, jumlah, pelipattigaan, cermin, pemeriksaan keseimbangan.
3. **Soal 12.7–12.10** — penghapusan, induk, anak-anak, dan pohon yang diratakan menjadi *double linked list*.
4. **Soal 12.11–12.16** — membalik *double linked list*, lingkaran Josephus, memilih buku, memeriksa BST, leluhur bersama terendah, mencetak tingkat demi tingkat.

Setiap soal menempuh jalur yang sama: pernyataannya, dengan kata-kata kita sendiri, dengan tangan, penyelesaian C++ (Java dan Python ada di buku), penyelesaian keliru yang menggoda, dan pertanyaan uji diri. Buku: Subbab 12.1.

Titik Tolak Kita: Tiga Alat

Bab 9: tautan kedua per simpul. Menyusuri ke dua arah, membuang simpul yang dipegang; setiap panah maju yang berubah memerlukan kembaran mundurnya.

Bab 10: rekursi. *Base case* dijawab langsung, *recursive case* memercayai panggilan yang lebih kecil; *call stack*; rekurens.

Bab 11: pohon dan *binary search tree*. Sisip, telusuri, cari, hitung, tinggi, hapus; hampir semuanya rekursif.

> **SATU GAGASAN.** Sebuah pohon itu kosong, atau berupa akar dengan pohon kiri dan pohon kanan: jawablah pohon kosong secara langsung, ajukan pertanyaan yang sama kepada kedua pohon yang lebih kecil, lalu gabungkan kedua jawabannya dengan akar.

Bila sebuah program hari ini keliru, salah satu dari tiga bagian itu rusak: pohon kosong dijawab salah, pertanyaannya berubah di jalan turun, atau jawaban pohon yang lebih kecil dibuang alih-alih digabungkan. Buku: Subbab 12.1.

Metode Tujuh Langkah

Tabel 12.1 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?

Tabel 12.1: langkah 1 dan 2 memastikan jawabannya, langkah 3 sampai 5 merancang rekursinya, langkah 6 dan 7 memeriksanya. Setiap program C++ juga dijalankan di bawah AddressSanitizer dan Valgrind. Buku: Subbab 12.2.

Templat Fungsi Pohon

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.

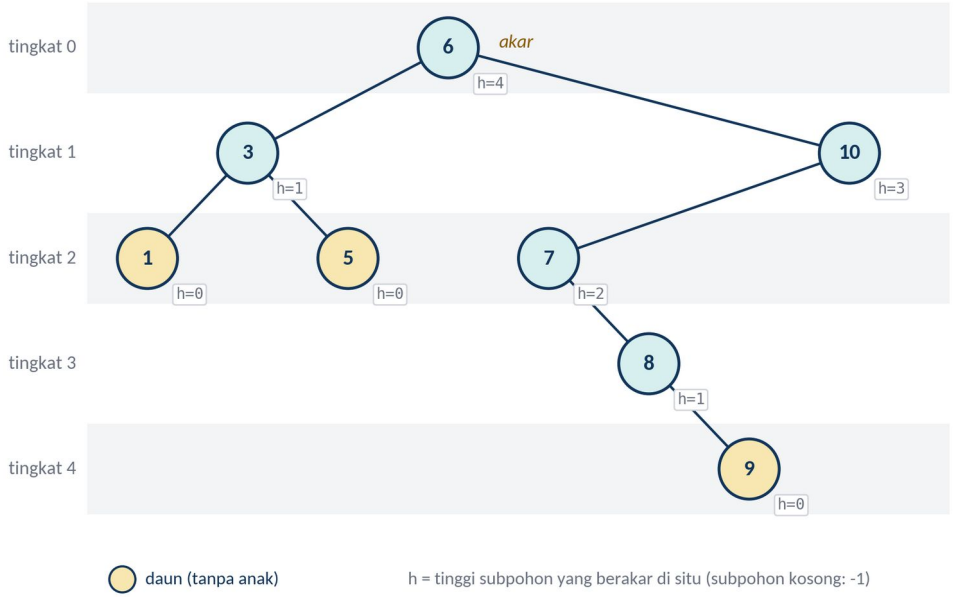
```
TREE-FUNCTION(T)
1  if T = NIL
2      return ANSWER-FOR-THE-EMPTY-TREE          // Langkah 2: base case
3  a = TREE-FUNCTION(T.left)                     // percayai pemanggilan kecil
4  b = TREE-FUNCTION(T.right)                    // percayai pemanggilan kecil
5  return COMBINE(T.key, a, b)                   // Langkah 4: akar, kiri, kanan
```

Pohon n simpul punya n panggilan pada pohon tak kosong dan $n+1$ pada anak kosong, sehingga berlaku Persamaan (12.1):

$$T(n)=cn+c_0(n+1)=\Theta(n). \tag{12.1}$$

Pseudokode 12.1: isi bagian kosongnya dengan 0 dan $T.key+a+b$ untuk jumlah, dengan -1 dan $1+\max(a,b)$ untuk tinggi. Pengecualian ongkos linear: penelusuran yang berhenti lebih awal, satu lintasan BST, penelusuran di dalam penelusuran. Buku: Subbab 12.2.

Pohon yang Kita Kerjakan



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.

Gambar 12.1: sisipkan 6, 3, 10, 1, 5, 7, 8, 9. Tingkat 0 sampai 4, tinggi 4, daun 1, 5, dan 9. Soal 12.1 sampai 12.10 semuanya menyusuri gambar ini. Buku: Subbab 12.3.1 dan Gambar 12.1.

Tiga Fakta yang Terus Dipakai

$$h(\varnothing) = -1,\; h(T) = 1 + \max\big(h(T.left), h(T.right)\big). \tag{12.2}$$

$$\text{untuk setiap simpul } x\text{: } y < x.key \text{ untuk setiap } y \text{ di } x.left,\; y > x.key \text{ untuk setiap } y \text{ di } x.right. \tag{12.3}$$

$$n \le 2^{h+1} - 1 \Rightarrow h \ge \lceil \lg(n+1) \rceil - 1, \text{dan selalu } h \le n - 1. \tag{12.4}$$

Persamaan (12.2) memberi daun tinggi $1 + \max(-1, -1) = 0$. *Persamaan (12.3) menyangkut setiap keturunan: 5 adalah anak kanan 3 tetapi harus lebih kecil daripada 6. Menurut Persamaan (12.4) 8 simpul kita bertinggi antara 3 dan 7. Buku: Subbab 12.3.1.*

Soal 12.1 — Depth First, Cabang Kiri Lebih Dulu

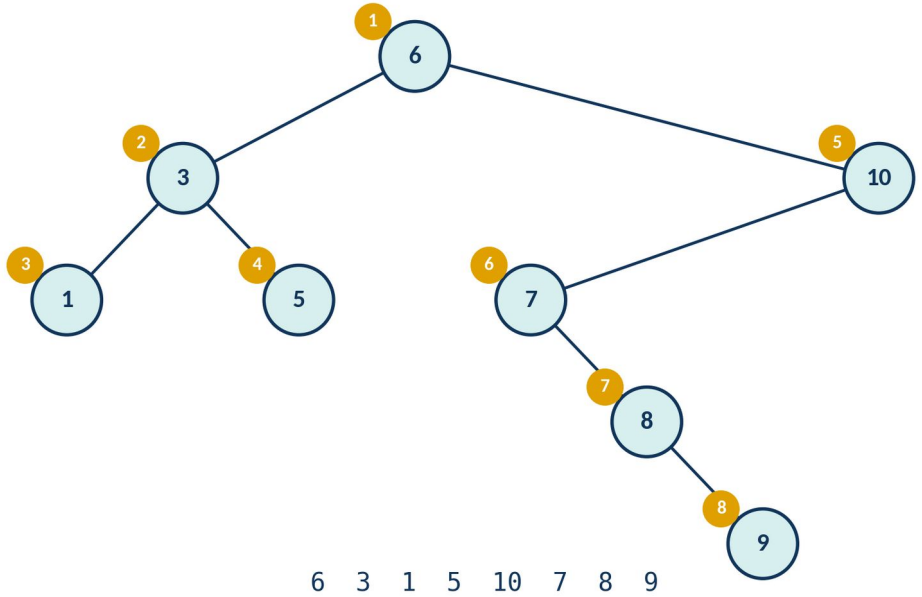
SOAL. Write a recursive function to print the depth-first traversal of a tree, i.e., choosing left branches first. `void printDepthFirst(Node *root)`

Kode 12.1 `printDepthFirst` dari `p1201_depth_first.cpp`: simpulnya, lalu seluruh subpohon kirinya, lalu seluruh subpohon kanannya.

```
// 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
}
```

Kode 12.1 adalah preorder: `6 3 1 5 10 7 8 9`*. Buku juga memberi versi dengan* `std::stack` *eksplisit, yang mendorong anak kanan sebelum anak kiri. Waktu* $\Theta(n)$*, stack* $O(h)$*. Buku: Subbab 12.4.1–12.4.4.*

Urutan Pencetakan



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.

Gambar 12.2: ketika 9 dicetak, panggilan pada 6, 10, 7, 8, dan 9 semuanya menunggu: satu lebih banyak daripada tingginya. Buku: Subbab 12.4.3 dan Gambar 12.2.

Keliru: Tanpa Base Case

```
void printDepthFirst(Node *root) {
    printf("%d ", root->data);          // WRONG: no base case at all
    printDepthFirst(root->left);
    printDepthFirst(root->right);
}
```

GCC memperingatkan `infinite recursion detected`; programnya mencetak 6, 3, 1 lalu membaca `root->data` lewat anak kiri 1, sebuah `nullptr`: *SEGV on unknown address 0x000000000000*.

Bacalah laporan *sanitizer* dari atas: *apa* yang terjadi, *di mana* (baris 7), dan *bagaimana kita sampai di sana* (baris 8).

Setiap pohon, sebesar apa pun, berakhir pada $n+1$ anak `nullptr`, sehingga setiap fungsi pohon rekursif harus menyatakan apa yang terjadi pada `nullptr`. Buku: Subbab 12.4.6.

Soal 12.2 — Mencetak Satu Tingkat

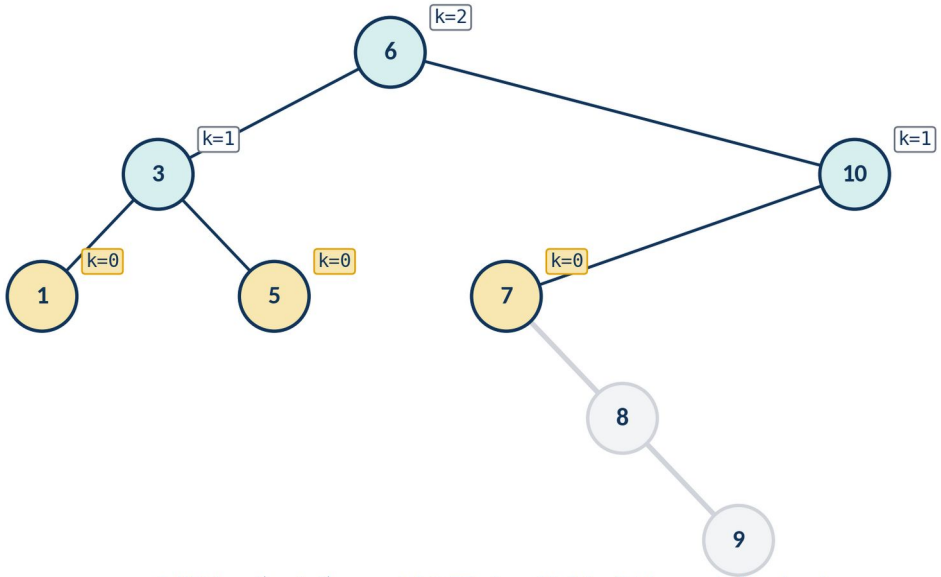
SOAL. Write a recursive function to print all nodes at the k-th level in the binary tree. `void printKthLevel(Node *root, int k)`

Kode 12.2 `printKthLevel` dari `p1202_kth_level.cpp`: pertanyaannya berubah di jalan turun, dari k menjadi $k-1$.

```
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);
}
```

Kode 12.2 menguji `root == nullptr` *sebelum* `k == 0`*. Tingkat 0 sampai 5 mencetak* `6`*,* `3 10`*,* `1 5 7`*,* `8`*,* `9`*, dan baris kosong. Mencetak semua tingkat dengan cara ini berongkos sampai* $\Theta(n^2)$ *pada rantai. Buku: Subbab 12.5.1–12.5.4.*

Nilai k yang Diterima Setiap Panggilan



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.

Gambar 12.3: untuk tingkat 2, panggilan dengan $k = 0$ mencetak 1, 5, dan 7 lalu berhenti, sehingga 8 dan 9 tidak pernah dikunjungi. Buku: Subbab 12.5.3 dan Gambar 12.3.

Keliru: `--k` Mengubah k Itu Sendiri

```
    printKthLevel(root->left, --k);      // WRONG: --k changes k itself,
    printKthLevel(root->right, --k);     // so the right side gets k - 2
```

```
level 0: 6
level 1: 3
level 2: 1 10
level 3: 5 7
```

Panggilan pertama menerima $k-1$, tetapi juga telah mengubah k lokal, sehingga panggilan kedua menerima $k-2$: sisi kanan muncul satu baris lebih awal. `k - 1` menghitung nilai dan membiarkan `k` utuh. Langkah 3 metode ini langsung memperlihatkannya. Buku: Subbab 12.5.6.

Soal 12.3 — Jumlah Semua Simpul

SOAL. Write a recursive function to calculate the sum of all nodes in the binary tree. `int sumOfNodes(Node *root)`

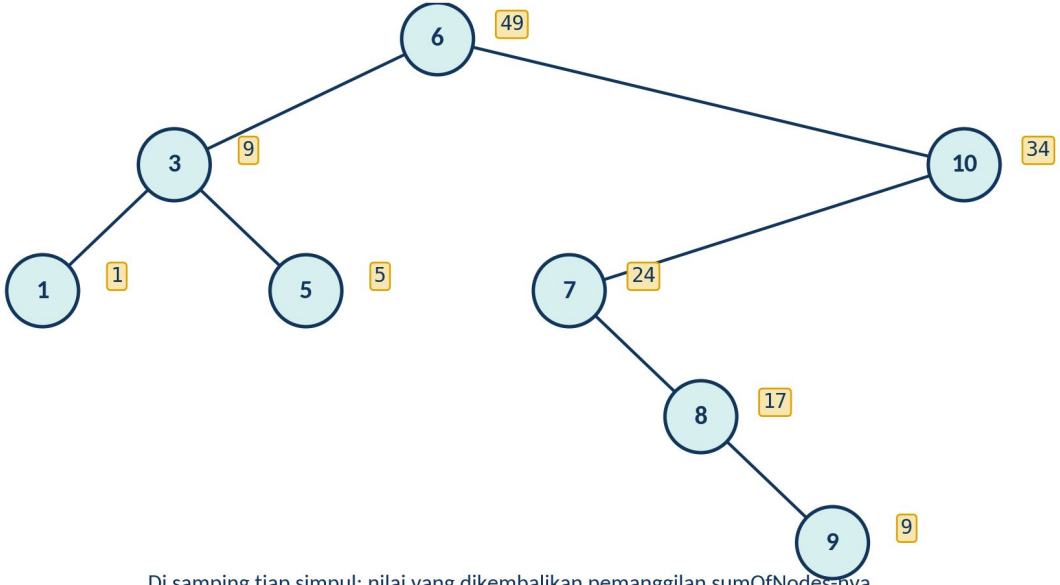
$$S(\varnothing)=0, S(T)=T.key+S(T.left)+S(T.right). \tag{12.5}$$

Kode 12.3 `sumOfNodes` dari `p1203_sum.cpp`: satu baris rekursi.

```
int sumOfNodes(Node *root) {
    if (root == nullptr) return 0;       // the empty tree adds nothing
    return root->data + sumOfNodes(root->left) + sumOfNodes(root->right);
}
```

Persamaan (12.5) dan Kode 12.3: base case-nya 0 karena 0 tidak mengubah apa pun bila ditambahkan. Seluruh pohon memberi 49, kedua subpohonnya 9 dan 34, dan $6+9+34=49$. *Buku: Subbab 12.6.1–12.6.4.*

Nilai yang Dikembalikan Panggilan



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.

Gambar 12.4: bacalah dari bawah ke atas, urutan selesainya panggilan: 8 mengembalikan 17, 7 mengembalikan 24, 10 mengembalikan 34, 3 mengembalikan 9, akar 49. Buku: Subbab 12.6.3 dan Gambar 12.4.

Keliru: Total Berjalan di Luar Fungsi

```
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;
}
```

```
sum of the whole tree  : 49
sum again (same answer): 98
left subtree of 6      : 107
```

Panggilan pertama benar, dan itulah yang membuat kesalahan ini menggoda; `total` mengingat setiap panggilan sebelumnya. Fungsi rekursif menggabungkan apa yang dikembalikan panggilannya; ia tidak berkomunikasi lewat variabel bersama. Buku: Subbab 12.6.6.

Soal 12.4 — Melipattigakan Setiap Simpul

SOAL. Write a recursive function which multiplies by three every node of the tree. `void triple(Node *root)`

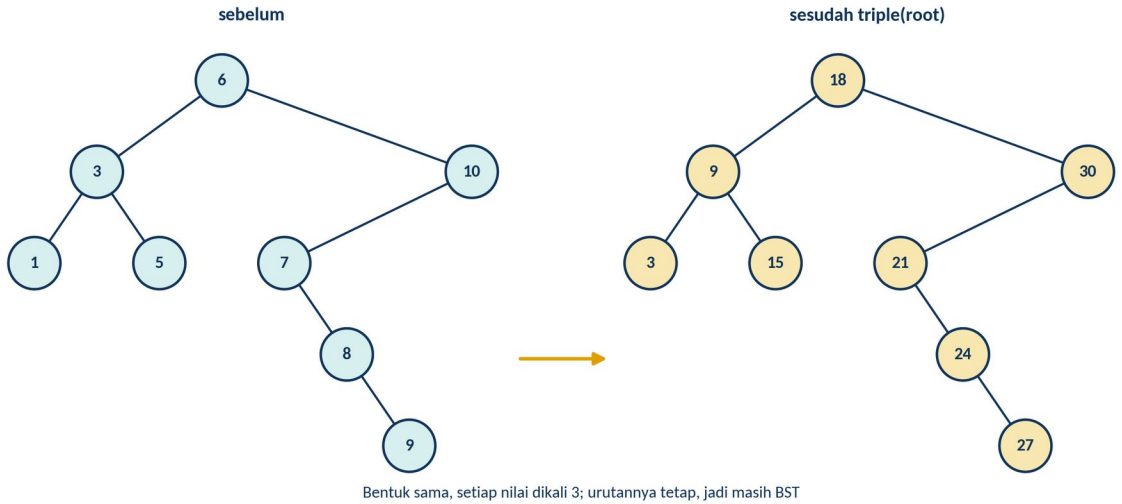
Kode 12.4 `triple` dari `p1204_triple.cpp`: simpul diubah lewat pointer.

```
void triple(Node *root) {
    if (root == nullptr) return;
    root->data *= 3;  // change the node itself, through the pointer
    triple(root->left);
    triple(root->right);
}
```

$$a<b \Rightarrow 3a<3b\,. \tag{12.6}$$

Kode 12.4; menurut Persamaan (12.6) hasilnya tetap BST, dan `search` menemukan 24. Versi keliru `void triple(Node node)` hanya melipattigakan salinan: tidak ada yang berubah. Buku: Subbab 12.7.1–12.7.5.

Sebelum dan Sesudah Dilipattigakan



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*.

Gambar 12.5: preorder `6 3 1 5 10 7 8 9` *menjadi* `18 9 3 15 30 21 24 27`*; inorder-nya tetap menaik. Buku: Subbab 12.7.2 dan Gambar 12.5.*

Soal 12.5 — Bayangan Cermin

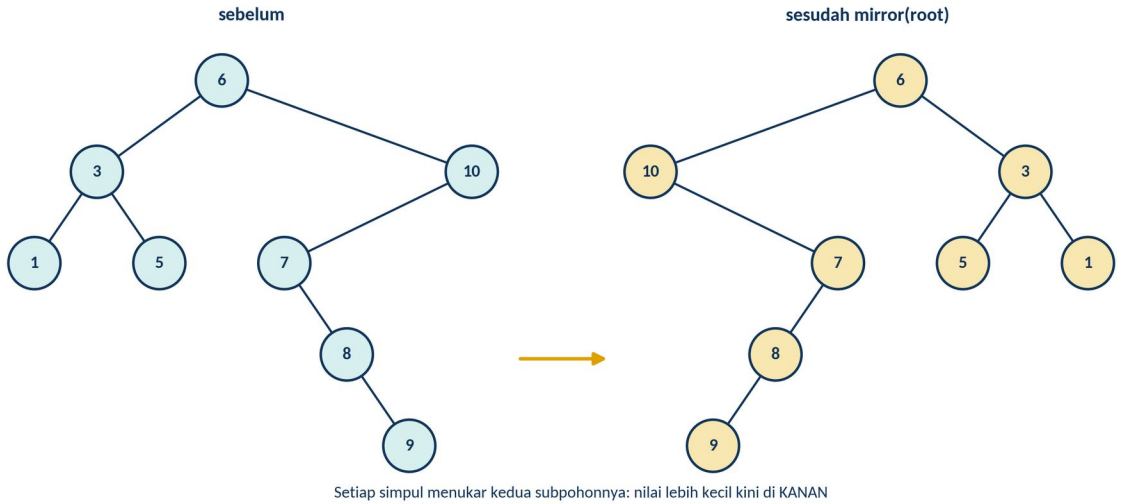
SOAL. Write a recursive function to create a mirror image of the binary tree. `void mirror(Node *root)`

Kode 12.5 `mirror` dari `p1205_mirror.cpp`: tukar kedua subpohon dengan pointer yang disimpan, lalu cerminkan masing-masing.

```
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);
}
```

Kode 12.5: preorder `6 10 7 8 9 3 5 1`*, inorder* `10 9 8 7 6 5 3 1`*.* `search(3)` *kini gagal: cerminnya bukan BST. Mencerminkan dua kali mengembalikan pohon asal. Buku: Subbab 12.8.1–12.8.4.*

Pohon di dalam Cermin



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*.

Gambar 12.6: setiap simpul menukar kedua subpohonnya; nilai-nilai yang lebih kecil kini tergantung di kanan. Buku: Subbab 12.8.3 dan Gambar 12.6.

Keliru: Menukar Tanpa Menyimpan

```
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 subpohon kanan; baris kedua menyalinnya kembali. “Pohon” yang tercetak adalah `6 10 10`.

Enam simpul tak terjangkau (bocor), dan `clear` melepas simpul 10 dua kali: AddressSanitizer melaporkan *heap-use-after-free*.

Aturan Bab 8 di tempat baru: sebelum menimpa sebuah pointer, simpan apa yang ditunjuknya. Penugasan tupel Python menyimpankannya untuk Anda. Buku: Subbab 12.8.6.

Soal 12.6 — Seimbang-Tinggikah Pohon Ini?

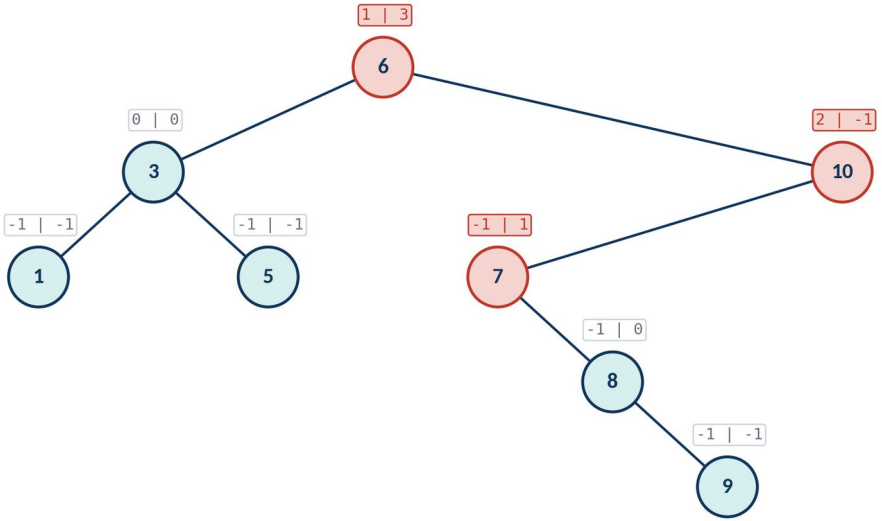
DEFINISI. Pohon biner T *seimbang-tinggi* bila T kosong, atau bila $|h(T.left) - h(T.right)| \leq 1$ dan kedua subpohonnya seimbang-tinggi.

Kode 12.6 `isBalancedHelper` dari `p1206_balanced.cpp`: simpulnya, lalu kedua subpohonnya.

```
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);
}
```

Kode 12.6: setiap simpul harus seimbang, bukan hanya akar; pohon kosong itu seimbang. Pohon 5 3 8 1 4 7 9 0 10 11 menyembunyikan simpul 8 yang tak seimbang di bawah akar yang seimbang. Buku: Subbab 12.9.1–12.9.4.

Tinggi di Atas Setiap Simpul



Label di atas tiap simpul: height(kiri) | height(kanan). Merah: selisihnya lebih dari 1

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.

Gambar 12.7: 7 bertinggi -1 *dan 1, 10 bertinggi 2 dan* -1*, akar 1 dan 3: jawabannya 0. Buku: Subbab 12.9.3 dan Gambar 12.7.*

Satu Kali Jalan, Bukan Penelusuran di dalam Penelusuran

Kode 12.7 `checkedHeight` dari `p1206_balanced.cpp`: tingginya, atau `UNBALANCED` begitu ada simpul yang melanggar aturan.

```
// 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;
}
```

$$C = \sum_{x \text{ diperiksa}} size(x) = \sum_{x \text{ diperiksa}} (d(x)+1) \le n(h+1), \tag{12.7}$$

Kode 12.7 bernilai $\Theta(n)$*; versi dengan fungsi bantu berongkos Persamaan (12.7), dalam praktik* $O(n \log n)$*.* `UNBALANCED` *bernilai* -2*, sebab* -1 *sudah berarti "pohon kosong". Buku: Subbab 12.9.4.*

Keliru: “Kosong Berarti Tidak Seimbang”

```
      if (root == nullptr) return 0;       // WRONG: "empty tree: not balanced"
```

```
sample tree            isBalanced = 0, one-pass = 0
6 3 9 1 5 8 10         isBalanced = 0, one-pass = 1
empty tree             isBalanced = 0, one-pass = 1
```

Setiap pohon berakhir pada subpohon kosong, sehingga angka 0 dibawa naik oleh `&&`: pohon perfect pun “tidak seimbang”. Ia lolos contoh soalnya sendiri. Base case harus tidak mengubah apa pun di bawah `&&`: true. Buku: Subbab 12.9.6.

Soal 12.7 — Menghapus Sebuah Nilai

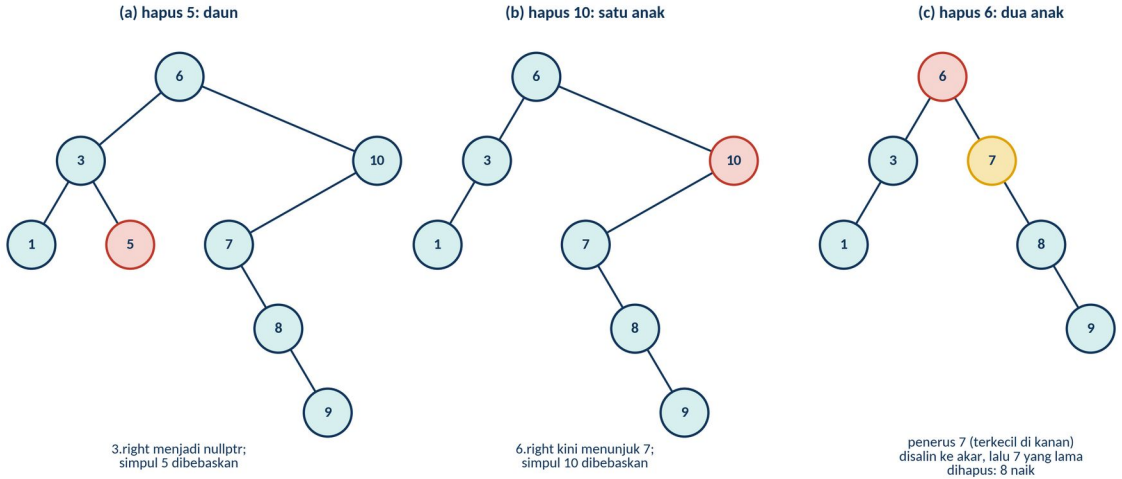
SOAL. Write a recursive function to delete a particular data. `Node *delNode(Node *root, int delData)`, with the helper `Node *minValueNode(Node *node)`.

Tabel 12.2 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

Tabel 12.2: fungsi mengembalikan akar sesudah penghapusan, sebab akar itu sendiri mungkin hilang. Kasus 3 adalah skema Hibbard dari Subbab 11.7.10. Buku: Subbab 12.10.1–12.10.2.

Ketiga Kasus pada Pohon Kita



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.

Gambar 12.8: hapus 5 (daun), lalu 10 (satu anak, 7 naik), lalu 6 (dua anak: 7 disalin ke akar). Menghapus 42 tidak mengubah apa pun. Buku: Subbab 12.10.3 dan Gambar 12.8.

`delNode` dalam C++

Kode 12.8 `delNode` dari `p1207_delete.cpp`: setiap pemanggil menautkan kembali apa yang dikembalikan panggilannya.

```
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 {                                          // 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;
}
```

Kode 12.8, tanpa cabang cermin untuk anak kanan yang tidak ada: kasus 1 dan 2 adalah tiga baris yang sama, sebab pada kasus 1 satu-satunya anak adalah `nullptr`*. Preorder sesudah 5, 10, 6:* `7 3 1 8 9`*. Ongkos* $O(h)$*. Buku: Subbab 12.10.4.*

Keliru: Membuang Hasilnya

```
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
```

Panggilan pada 5 melepas simpulnya dan mengembalikan `nullptr`, tetapi 3 mengabaikannya: `3->right` menjadi *pointer menggantung*.

Penelusuran berikutnya membaca memori yang sudah dilepas: *heap-use-after-free* di `preOrder`, dilepas di `delNode`, dialokasikan di `insert`.

Perbaikannya satu penugasan per cabang: `root->left = delNode(root->left, delData)`. *Mencampur* `free` *dengan* `new` *adalah bug kedua yang lebih senyap (alloc-dealloc-mismatch). Buku: Subbab 12.10.6.*

Soal 12.8 — Mencari Induk

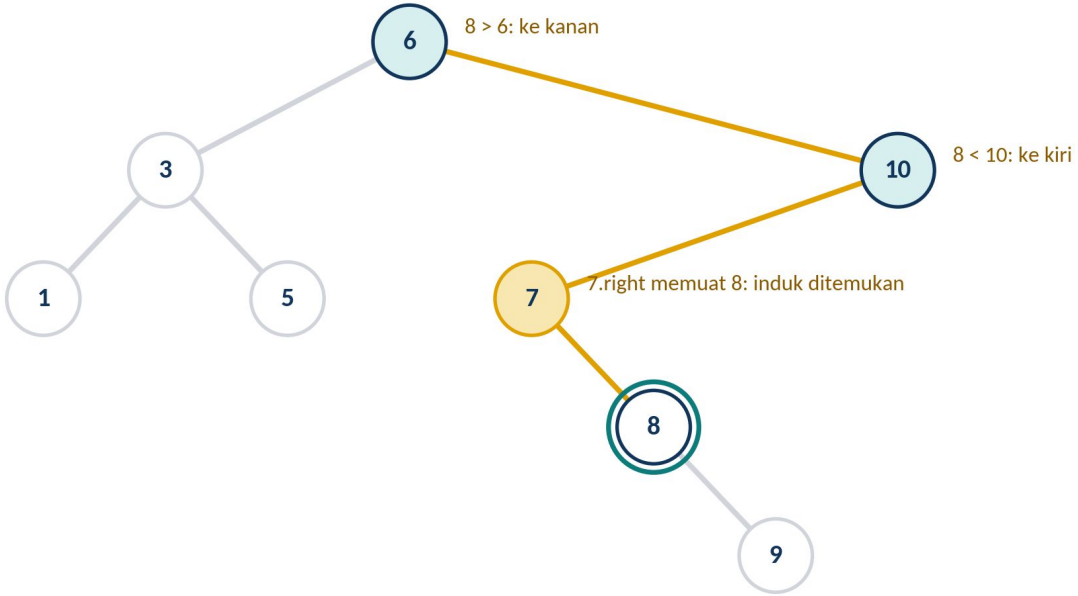
SOAL. Write a recursive function to find the parent node of a given child data. `Node *findParent(Node *root, int *childData)`

Kode 12.9 `findParent` dari `p1208_find_parent.cpp`: lihat satu langkah ke depan pada lintasan pencarian.

```
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
}
```

Kode 12.9 menerima nilainya sebagai `int`, karena hanya membacanya. Induk: 8→7, 10→6, 9→8, 1→3; 6 adalah akar dan 4 tidak ada, keduanya `nullptr`. Versi untuk pohon sembarang melihat ke dua sisi, $\Theta(n)$. Buku: Subbab 12.11.1–12.11.5.

Satu Langkah ke Depan pada Lintasan



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.

Gambar 12.9: di 7 simpul berikutnya memuat 8, jadi 7 adalah induknya. Satu baris keliru `root->left->data == childData || …` *crash di 10, yang tidak punya anak kanan. Buku: Subbab 12.11.3 dan 12.11.6 serta Gambar 12.9.*

Soal 12.9 — Melaporkan Anak-Anak

Kode 12.10 `findChild` dari `p1209_find_child.cpp`: satu panggilan `search` dan dua pemeriksaan yang cermat.

```
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;
}
```

Kode 12.10: 3 punya 1 dan 5; 7 hanya punya 8; 9 adalah daun; 4 tidak ada. Memercayai `search` begitu saja crash pada 4 di alamat `0x8`: medan `left` dari pointer null, 8 bita ke dalam. Buku: Subbab 12.12.

Soal 12.10 — Pohon Menjadi List

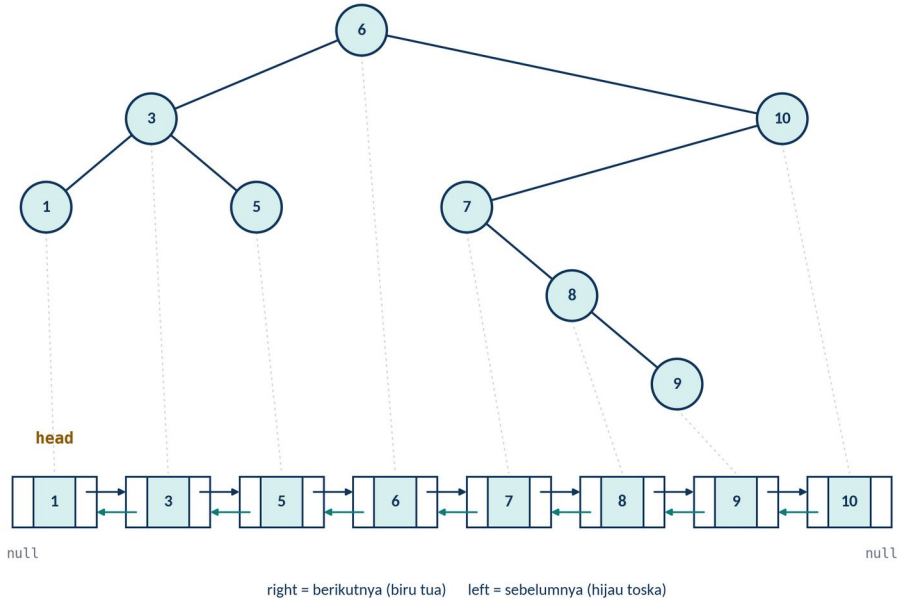
> **SOAL.** Write a recursive function to convert the binary tree to a doubly linked list in place. The left pointer acts as the previous pointer and the right pointer as the next pointer. `Node *convertToDLL(Node *root)`

Penelusuran **inorder** mengunjungi simpul dalam urutan *list*; tambahkan setiap simpul yang dikunjungi sesudah `prev` dengan dua panah: `prev->right = root`, `root->left = prev`.

`head` dan `prev` harus *dibagi* oleh semua panggilan, sehingga fungsi bantunya menerima alamat keduanya, `Node **`.

Menimpa itu aman: `root->left` baru berubah sesudah subpohon kiri selesai, dan `root->right` sudah dibaca serta diteruskan ke bawah. Buku: Subbab 12.13.1–12.13.3.

Delapan Simpul yang Sama, Ditaut Ulang



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.

Gambar 12.10: `right` kini menunjuk nilai berikutnya yang lebih besar, `left` nilai berikutnya yang lebih kecil. Buku: Subbab 12.13.2 dan Gambar 12.10.

Penambahan secara Inorder

Kode 12.11 Fungsi bantu dan `convertToDLL` dari `p1210_to_dll.cpp`.

```
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;
}
```

Kode 12.11: maju `1 3 5 6 7 8 9 10`, mundur `10 9 8 7 6 5 3 1`. Simpul-simpulnya kini sebuah list: lepaskan dengan penyusuran sepanjang `right`, jangan dengan `clear` milik pohon. Java dan Python menyimpan `head` dan `prev` dalam sebuah objek. Buku: Subbab 12.13.4–12.13.5.

Keliru: `prev` Dikirim sebagai Nilai

```
void convertToDLLHelper(Node *root, Node **head, Node *prev) {  // WRONG: a copy
    ...
    prev = root;                                    // changes the copy only
    convertToDLLHelper(root->right, head, prev);
}
```

```
forward : 6 10
backward: 10 6
SUMMARY: AddressSanitizer: 144 byte(s) leaked in 6 allocation(s).
```

Salinan membawa informasi turun, tidak pernah naik kembali: 3 dan 6 masing-masing mengira dirinya yang pertama dan memutus semua simpul sebelumnya. Variabel yang harus dibagi oleh panggilan-panggilan dikirim lewat pointer atau disimpan dalam objek bersama. Buku: Subbab 12.13.6.

Soal 12.11 — Membalik Double Linked List

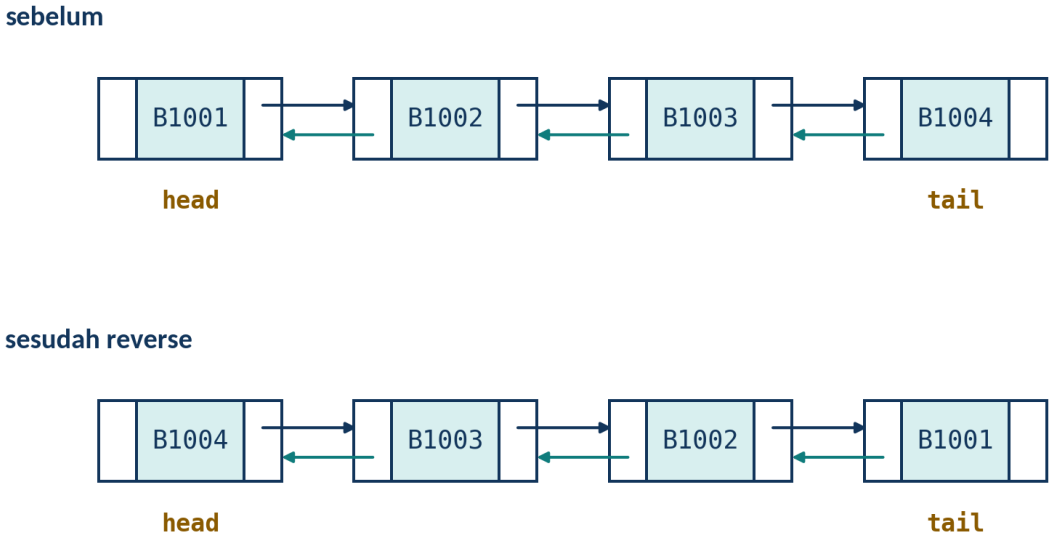
> **SOAL.** Baliklah riwayat baca seorang anggota, sebuah *double linked list* dengan `head` dan `tail`, di tempat, tanpa membuat simpul dan tanpa memindahkan pengenal buku mana pun.

Kode 12.12 `reverse` dari `p1211_reverse_dll.cpp`: tukar kedua panah setiap simpul, lalu tukar kedua ujungnya.

```
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 12.12: bergeraklah ke `next` lama yang disimpan. `cur = cur->next` yang keliru mengikuti panah yang baru saja ditukar, berhenti sesudah satu simpul, dan membocorkan tiga. Waktu $\Theta(n)$, memori $O(1)$. Buku: Subbab 12.14.1.

Hanya Panahnya yang Berubah



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.

Gambar 12.11: setiap simpul tetap di tempatnya bersama bukunya; membalik dua kali mengembalikan list asal. Buku: Subbab 12.14.1 dan Gambar 12.11.

Soal 12.12 — Lingkaran Josephus

n anggota duduk melingkar; setiap anggota ke-k yang terhitung keluar; yang terakhir tersisa mendapat eksemplar awal.

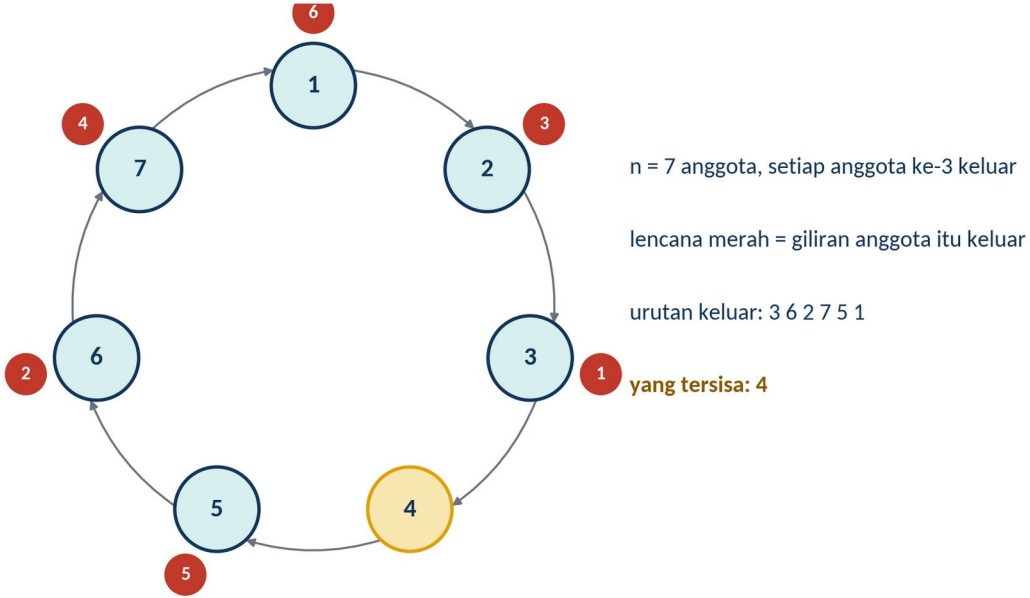
List sirkular memodelkannya: berhitung mengelilingi lingkaran berarti mengikuti `next` tanpa henti.

$$J(1)=0,\ J(n)=\big(J(n-1)+k\big)\bmod n\ (n\geq 2). \tag{12.8}$$

Untuk $n=7$, $k=3$: $J=0,1,1,0,3,0,3$; posisi 3 adalah anggota **4**.

Persamaan (12.8): sesudah anggota pertama keluar, lingkaran yang lebih kecil adalah soal yang sama yang dinomori ulang mulai posisi k. Rumus ini mengubah simulasi $O(nk)$ menjadi aritmetika $O(n)$. Buku: Subbab 12.14.2 dan Persamaan (12.8).

Contoh Terurai: $n=7$, $k=3$



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.

Gambar 12.12: anggota keluar dengan urutan 3, 6, 2, 7, 5, 1; anggota 4 tidak pernah terhitung ketiga. Buku: Subbab 12.14.2 dan Gambar 12.12.

Lingkaran dan Rumusnya

Kode 12.13 `josephusList` dari `p1212_josephus.cpp`; `josephusFormula` di sampingnya menghitung Persamaan (12.8).

```
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;
}
```

Kode 12.13: `prev` dimulai pada simpul terakhir, simpul sebelum anggota 1. Memulai dari anggota 1 membuat setiap hitungan terlambat satu, dan hanya rumus yang independen yang mengungkapnya. Buku: Subbab 12.14.2.

Soal 12.13 — Memilih Buku

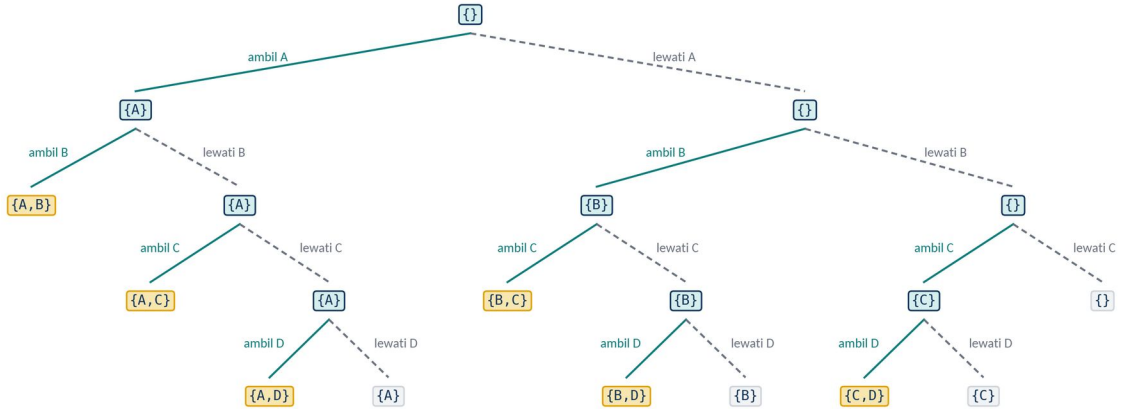
Cetak setiap pilihan tiga dari lima buku `B1001` sampai `B1005`. Setiap pilihan **mengambil** buku berikutnya dan memerlukan $k-1$ lagi, atau **melewatinya** dan memerlukan k lagi:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \quad \binom{n}{0} = \binom{n}{n} = 1. \tag{12.9}$$

Base case: $k = 0$, cetak pilihannya; buku yang tersisa kurang dari k, menyerah (*pemangkasan*).

Persamaan (12.9) adalah segitiga Pascal. `binom(20, 10)` *yang rekursif membuat 369.511 panggilan, mengulang submasalah seperti Fibonacci naif; sebuah tabel memangkasnya menjadi sekitar* $n\,k$*. Buku: Subbab 12.15.1 dan Persamaan (12.9).*

Ambil atau Lewati



kuning: pilihan lengkap (k = 0), dicetak; kelabu: buku tersisa terlalu sedikit, ditinggalkan

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.

Gambar 12.13: dua dari A, B, C, D: enam belas panggilan, enam pilihan kuning tua, tiga jalan buntu. Buku: Subbab 12.15.1 dan Gambar 12.13.

Backtracking dalam C++

Kode 12.14 `choose` dari `p1213_choose_books.cpp`: tambahkan, jelajahi, buang.

```
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]
}
```

Kode 12.14: sepuluh pilihan dalam urutan katalog, dan $\binom{5}{3}=10$*. Lupakan* `pop_back` *dan setiap baris makin panjang. Apa pun yang diterima sebuah panggilan sebagai nilai dibatalkan dengan sendirinya; keadaan bersama harus dibatalkannya sendiri. Buku: Subbab 12.15.1.*

Soal 12.14 — Benarkah Ini BST?

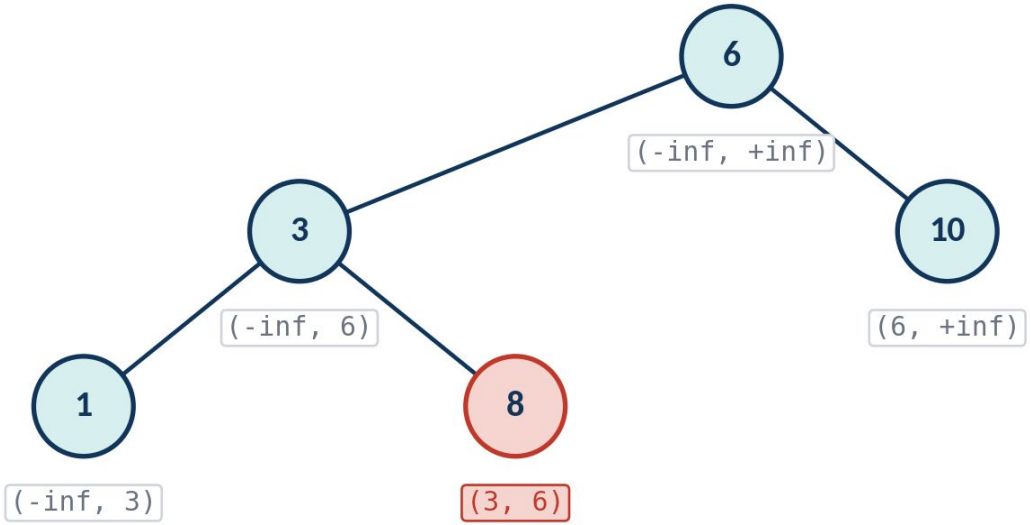
Kode 12.15 `inRange` dan `isBST` dari `p1214_is_bst.cpp`: setiap simpul diperiksa terhadap selang yang diwarisi dari leluhurnya.

```
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
}
```

Kode 12.15: ke kiri nilainya menjadi `high`, ke kanan menjadi `low`; selang terbuka menolak nilai yang sama. Membandingkan setiap simpul hanya dengan anak-anaknya keliru menerima pohon licik. Buku: Subbab 12.16.1.

Pohon yang Licik



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*.

Gambar 12.14: setiap pasangan induk-anak berurutan, tetapi 8 berada di luar $(3, 6)$*;* `search(8)` *akan melewatkannya. Inorder-nya* `1 3 8 6 10` *tidak menaik. Buku: Subbab 12.16.1 dan Gambar 12.14.*

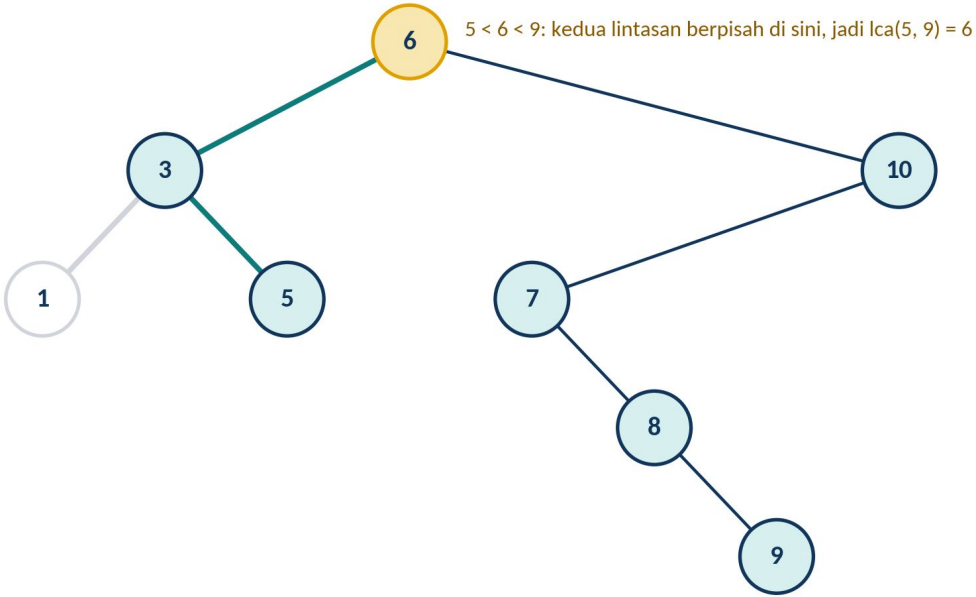
Soal 12.15 — Leluhur Bersama Terendah

Kode 12.16 `lca` dari `p1215_lca.cpp`: turun selama kedua nilai berada di sisi yang sama.

```
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
}
```

Kode 12.16: lca(1, 5) = 3, lca(5, 9) = 6, lca(8, 9) = 8, lca(9, 9) = 9. Berkemudi dengan satu nilai saja hanya berjalan menuju a. Satu lintasan, $O(h)$. Buku: Subbab 12.16.2.

Tempat Lintasan Berpisah



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.

Gambar 12.15: karena $5<6<9$, kedua lintasan berpisah di akar. Buku: Subbab 12.16.2 dan Gambar 12.15.

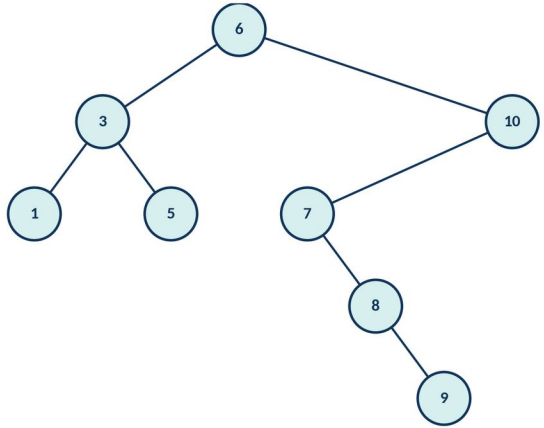
Soal 12.16 — Tingkat demi Tingkat

Kode 12.17 `printLevels` dari `p1216_levels.cpp`: ukuran *queue*, dibaca sekali, adalah lebar tingkat itu.

```
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;
}
```

Kode 12.17: $\Theta(n)$ *alih-alih* $\Theta(n^2)$ *milik Soal 12.2. Membaca* `q.size()` *di dalam kondisi gelung membuat tingkat-tingkat menyatu:* `level 0: 6 3 10`*. Buku: Subbab 12.16.3.*

Queue, Satu Tingkat Setiap Kali



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.

Gambar 12.16: lebar 1, 2, 3, 1, 1; pohon perfect bertinggi 3 memberi 1, 2, 4, 8. Buku: Subbab 12.16.3 dan Gambar 12.16.

Soal-Soal Dibandingkan

Setiap simpul sekali bernilai $\Theta(n)$ menurut Persamaan (12.1): Soal 12.1, 12.3–12.5, 12.10, 12.14, 12.16.

Satu lintasan dari akar bernilai $O(h)$: Soal 12.7–12.9 dan 12.15, itulah sebabnya pohon seimbang penting.

Penelusuran di dalam penelusuran adalah pola yang harus diwaspadai: pemeriksaan keseimbangan dengan fungsi bantu, semua tingkat lewat Soal 12.2. Biarkan penelusuran luar *mengembalikan* apa yang dihitung ulang oleh penelusuran dalam.

Beberapa penyelesaian keliru *lolos contoh soal*: pemeriksaan keseimbangan, jumlah kedua, urutan Josephus. Langkah 6 dan 7 menangkapnya.

Tabel perbandingan di buku mencantumkan, untuk keenam belas soal, strukturnya, ongkosnya, dan masukan tempat penyelesaian kelirunya gagal. Buku: Subbab 12.17.

Profitina dalam Praktik

Diskusi berutas adalah pohon: urutan baca adalah Soal 12.1, lekukan adalah tingkat Soal 12.2, jumlah balasan adalah jumlah Soal 12.3; menghapus kiriman yang punya balasan memunculkan pertanyaan Soal 12.7.

Situs disusuri tingkat demi tingkat: pemindai berkedalaman tetap memakai *queue*, seperti Soal 12.16.

Rekursi punya batas kedalaman: perender Python yang bersarang seribu balasan memunculkan `RecursionError`; gunakan *stack* eksplisit atau batasi sarangnya.

Profitina adalah platform visibilitas AI (GEO); kita melukiskan bentuk soal-soal ini dalam produk seperti itu, bukan implementasinya. Buku: Subbab 12.18.

Kesalahan yang Sering Terjadi

Tanpa *base case*, atau *base case* yang salah: 0 untuk `+`, *true* untuk `&&`, `INT_MIN` untuk max, -1 untuk tinggi. **Menguji `k == 0` sebelum `nullptr`.**

Berkomunikasi lewat variabel global; mengirim simpul sebagai nilai; menukar tanpa menyimpan.

Membuang subpohon yang dikembalikan; memakai fungsi BST pada pohon yang bukan BST; memeriksa hanya induk dan anak.

Mengikuti pointer yang baru ditimpa; lupa membatalkan dalam *backtracking*; rekursi terlalu dalam; hanya menguji contoh soal.

Setiap kesalahan adalah penyelesaian keliru yang menggoda dari salah satu soal hari ini. Buku: Subbab 12.19.

Ringkasan: Lima Hal untuk Diingat

1. Tujuh langkah: contoh, jawaban kosong, argumen digambar, akar-kiri-kanan, tautkan kembali, kasus tepi, uji di bawah pemeriksa memori.

2. *Base case* adalah nilai yang tidak mengubah apa pun di bawah langkah penggabungan.

3. Fungsi yang mungkin mengganti simpulnya mengembalikan penggantinya, dan setiap pemanggil menautkannya kembali.

4. Bagilah keadaan hanya dengan sengaja (lewat pointer atau objek), simpan sebelum menimpa, batalkan apa yang Anda ubah.

5. Setiap simpul sekali bernilai $\Theta(n)$, satu lintasan $O(h)$; waspadai penelusuran di dalam penelusuran.

Buku: Subbab 12.20.

Pertemuan Berikutnya

Bab 13 — Graf. Pohon yang tautannya boleh membentuk siklus: simpul, sisi, ketetanggaan, *breadth-first* dan *depth-first search*.

Queue dari Soal 12.16 dan *stack* dari Soal 12.1 kembali sebagai BFS dan DFS.

Sebelum itu: kerjakan Soal Latihan 12.1, 12.4, 12.7, dan 12.12, dalam C++ lalu dalam Java atau Python.

Penyelesaiannya, lengkap dengan program, ada di buku. Buku: Subbab 12.21 dan 12.22.

Kuliah 13 — Graf

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 13

Dosen: Irfan Subakti — yifana@gmail.com

Hari ini batasan terakhir hilang: benda-benda dan hubungan antarpasangan benda, dengan banyak hubungan sembarang dan putaran yang diperbolehkan. List adalah graf, pohon adalah graf, dan begitu pula hampir semua hal lain. Buku: Bab 13, Subbab 13.1–13.15.

Cakupan Pertemuan Ini

1. **Kosakata** — simpul, sisi, graf berarah dan berbobot, derajat dan lema jabat tangan, lintasan, siklus, komponen, DAG.
2. **Representasi** — matriks ketetanggaan, *list* ketetanggaan, *list* sisi, dan ongkos masing-masing.
3. **Penelusuran melebar dan mendalam** — *queue* dan *stack* yang bekerja, lengkap dengan bukti dan jejaknya.
4. **Tiga aplikasi** — komponen terhubung, pengurutan topologis, dan lintasan terpendek Dijkstra.
5. **Pada skala besar** — jaringan jalan dan graf sosial; audit situs web Profitina Laundry.

Contoh berjalannya adalah peta kampus Pustaka Ceria: sepuluh gedung, sebelas jalan setapak. Buku: Subbab 13.1.

Titik Tolak

Larik dan *list* menaruh elemen dalam **garis**; *stack* dan *queue* membatasi tempat kita menyentuhnya; pohon mengizinkan banyak anak tetapi **satu induk**.

Jalan setapak kampus, pertemanan, penerbangan, tautan web, dan ketergantungan tugas tidak menaati aturan itu, dan semuanya membentuk putaran.

Queue dari Bab 5 menggerakkan penelusuran melebar; ***stack***, atau rekursi, menggerakkan penelusuran mendalam; ***priority queue*** dari Bab 14 menggerakkan Dijkstra.

Setiap subbab menjawab satu pertanyaan praktis: dapatkah saya sampai ke sana, dengan berapa langkah, berapa meter, apa yang terputus, dalam urutan apa tugas harus berjalan? Buku: Subbab 13.1.

Jembatan-Jembatan Königsberg

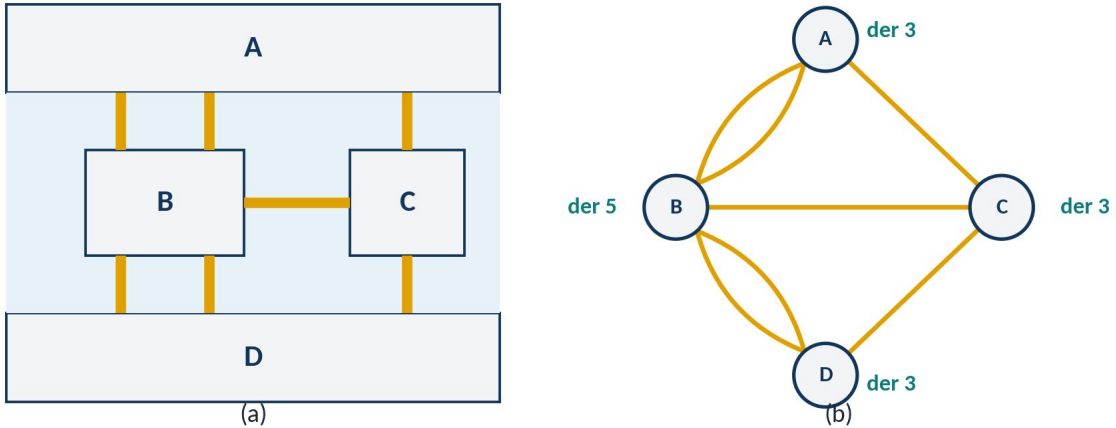
Tujuh jembatan menghubungkan empat daratan. Adakah jalan-jalan yang melintasi setiap jembatan tepat sekali?

Euler, 1736, hanya menyimpan daratan **mana** yang ada dan **pasangan mana** yang dihubungkan, serta berapa kali.

Jalan-jalan yang *melewati* sebuah daratan memakai jembatannya dua-dua, sehingga semua daratan kecuali titik awal dan akhir memerlukan jumlah genap. Di sini keempatnya ganjil (3, 5, 3, 3): mustahil.

Titik dan hubungan, satu hitungan per titik (derajat), dan penalaran tentang jalan: itulah teori graf dalam bentuk mini, berusia 290 tahun. Buku: Subbab 13.2.1.

Königsberg sebagai Graf



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.

Gambar 13.1: argumen ini adalah bukti, bukan kegagalan mencari dengan cukup keras. Buku: Subbab 13.2.1 dan Gambar 13.1.

Simpul dan Sisi

DEFINISI. *Graf* G terdiri atas himpunan berhingga V berisi *simpul* dan himpunan E berisi *sisi*, dengan setiap sisi menghubungkan sepasang simpul, seperti pada Persamaan (13.1).

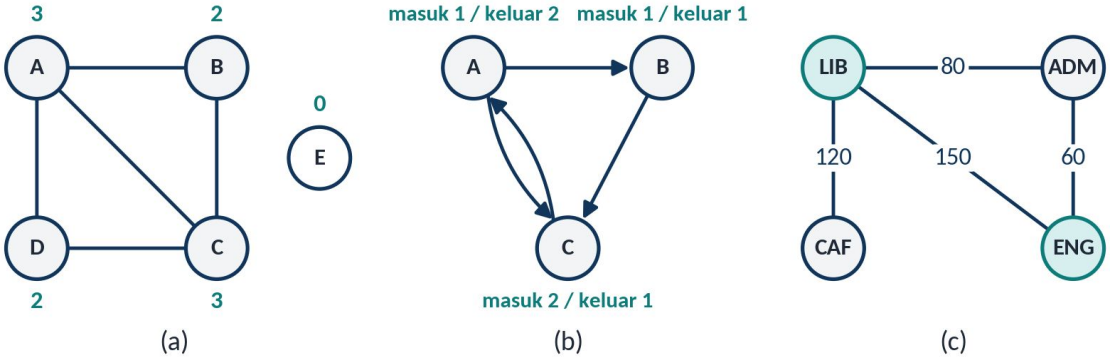
$$G=(V,E), E\subseteq\{\{u,v\}:u,v\in V, u\neq v\} \tag{13.1}$$

Simpul yang terhubung disebut *bertetangga*. $u\neq v$ melarang putaran-diri; E sebagai himpunan melarang sisi ganda: grafnya *sederhana*.

Königsberg, dengan dua jembatan antara A dan B, adalah *multigraf*. Kita menulis V dan E untuk $|V|$ dan $|E|$ dalam rumus waktu jalan.

Tanda kurung kurawal di Persamaan (13.1) berarti sisi tak berarah: $\{u,v\}$ *dan* $\{v,u\}$ *adalah sisi yang sama. Buku: Subbab 13.3.1 dan Persamaan (13.1).*

Tak Berarah, Berarah, Berbobot



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.

Gambar 13.2: (a) E adalah simpul terisolasi; (b) panah adalah pasangan berurut (u, v)*; (c)* $w(\text{LIB}, \text{ADM}) = 80$ *meter. Sisi tak berarah disimpan sebagai dua panah. Buku: Subbab 13.3.1–13.3.3 dan Gambar 13.2.*

Derajat dan Lema Jabat Tangan

$$\sum_{v \in V} \deg(v) = 2|E| \tag{13.2}$$

Bukti. Hitung pasangan (simpul, sisi yang bersisian) dengan dua cara: per simpul, $\sum \deg(v)$; per sisi, dua untuk setiap sisi.

Jadi banyaknya simpul berderajat ganjil itu genap: derajat 3, 3, 3, 3, 3 mustahil. Pada graf berarah:

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E| \tag{13.3}$$

Persamaan (13.2) adalah uji gratis bagi program yang membangun graf; Persamaan (13.3) menghitung derajat-masuk, dasar algoritma Kahn. Pada Gambar 13.2(a): $3+2+3+2+0 = 10 = 2 \cdot 5$*. Buku: Subbab 13.3.4.*

Lintasan, Siklus, Komponen, DAG

Lintasan adalah barisan simpul yang dihubungkan sisi; panjangnya banyaknya sisi; v **terjangkau** dari u bila lintasan itu ada.

Siklus kembali ke titik awalnya. Pohon persis graf tak berarah yang terhubung dan asiklik, jadi algoritma graf harus mengingat apa yang sudah dilihatnya.

> **DEFINISI.** *Komponen terhubung* graf tak berarah adalah himpunan simpul maksimal yang setiap simpulnya terjangkau dari setiap simpul lainnya.

DAG adalah graf berarah tanpa siklus berarah: ketergantungan.

"Terjangkau dari" adalah relasi ekuivalensi pada graf tak berarah, dan komponen-komponennya adalah kelas-kelasnya. Buku: Subbab 13.3.5–13.3.7.

Jarang dan Padat

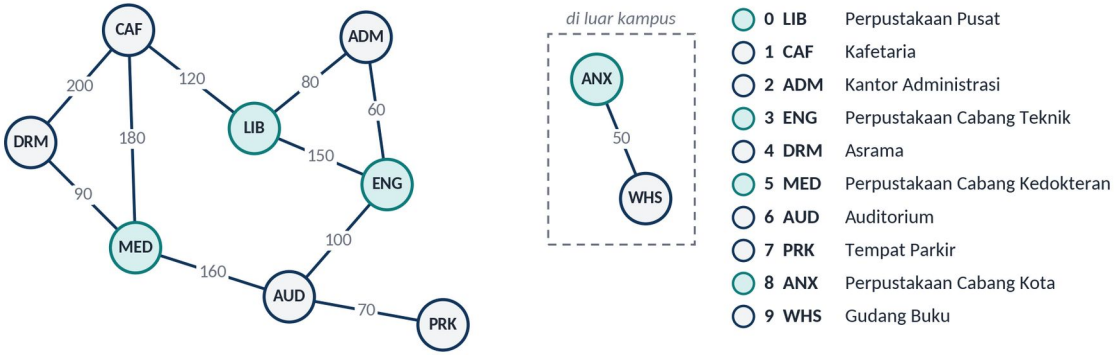
$$|E| \leq \binom{|V|}{2} = \frac{|V|(|V|-1)}{2} \tag{13.4}$$

$$\rho = \frac{|E|}{|V|(|V|-1)/2} \tag{13.5}$$

Padat: mendekati Persamaan (13.4); jarang: konstanta kecil kali $|V|$. Persamaan (13.5) bernilai 1 untuk graf lengkap.

Pada jaringan jalan hampir setiap persimpangan punya paling banyak empat atau lima jalan, sehingga $|E|$ tumbuh seperti $|V|$: graf yang paling penting sangat jarang. Buku: Subbab 13.3.8.

Peta Kampus Pustaka Ceria



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.

Gambar 13.3: tak berarah dan berbobot, penuh siklus, dan tidak terhubung (ANX dan WHS). Contoh berjalan mulai sekarang. Buku: Subbab 13.4 dan Gambar 13.3.

Contoh Terurai: Peta dalam Angka

$|V| = 10$, $|E| = 11$. Derajat 3, 3, 2, 3, 2, 3, 3, 1, 1, 1: jumlahnya 22 $= 2 \times 11$.

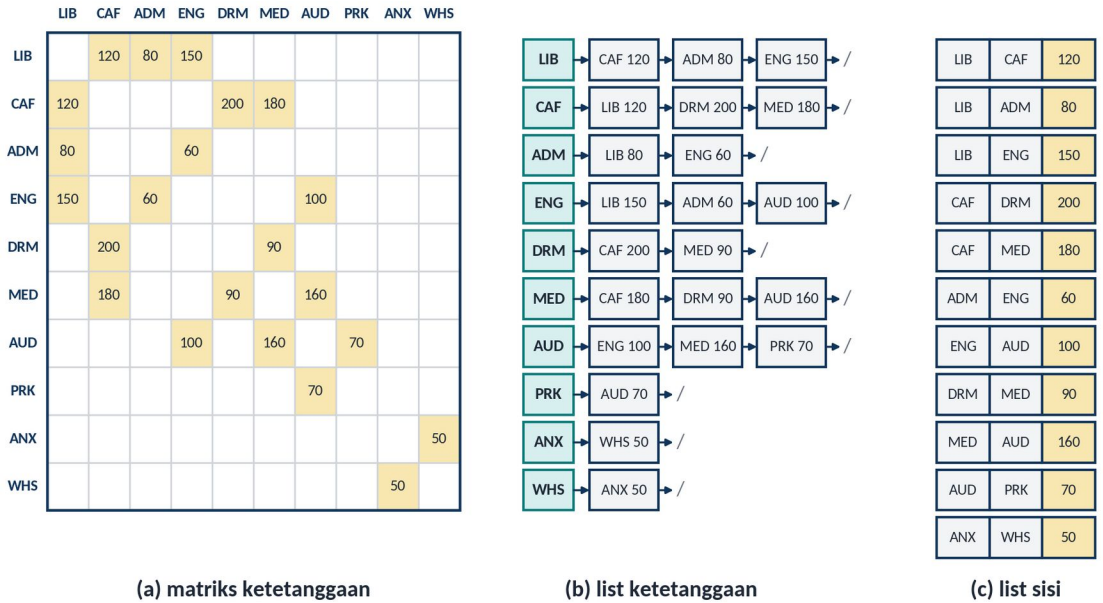
Delapan simpul berderajat ganjil: jumlah yang genap, sesuai tuntutan lema.

Paling banyak $10 \cdot 9/2 = 45$ sisi, sehingga kepadatannya $11/45 \approx 0,24$.

Langkah dan meter tidak sepakat: LIB, ENG, AUD, PRK adalah 3 jalan setapak dan 320 m; LIB, ADM, ENG, AUD, PRK adalah 4 jalan setapak dan 310 m.

BFS akan menemukan rute pertama, Dijkstra yang kedua. Buku: Subbab 13.4.

Tiga Representasi



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.

Gambar 13.4: matriksnya simetris dengan diagonal kosong dan 22 dari 100 sel terisi; setiap list memuat entri (tetangga, panjang); list sisi punya satu rekaman per jalan setapak. Buku: Subbab 13.5 dan Gambar 13.4.

Matriks atau List?

Matriks: "adakah sisi?" dalam $O(1)$; mendaftar tetangga memindai satu baris utuh, $\Theta(V)$.

List: tetangga dalam $\Theta(\deg(u))$; uji sisi memindai `adj[u]`. Sisi tak berarah muncul dua kali.

$$S_{\text{matriks}} = |V|^2 \,\text{sel}, S_{\text{list}} = |V| + 2|E| \,\text{entri} \tag{13.6}$$

> **PEGANGAN PRAKTIS.** Pakailah *list* ketetanggaan kecuali grafnya padat atau algoritmanya terutama bertanya "adakah sisi?". Simpan *list* sisi bila sisi datang dan diproses satu per satu.

Persamaan (13.6): 100 sel lawan 32 entri pada peta kampus. Dalam Python, `[[0] * n] * n` *membuat satu baris yang dipakai bersama* n *kali: pakailah* `[[0] * n for _ in range(n)]`*. Buku: Subbab 13.5.1–13.5.6 dan Persamaan (13.6).*

Ongkos Berdampingan

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

Tabel 13.1: baris penentunya adalah "kunjungi setiap sisi sekali": $\Theta(V+E)$ dengan list lawan $\Theta(V^2)$ dengan matriks. Buku: Subbab 13.5.4.

Penelusuran Melebar: Riak di Kolam

Dari sumber s: mula-mula s, lalu setiap simpul berjarak satu sisi, lalu dua, lalu tiga: cincin-cincin dengan **jarak langkah** yang sama.

***Queue* FIFO** menjaga urutan cincin: cincin 1 masuk sebelum simpul mana pun dari cincin 2 ditemukan.

Untuk setiap simpul BFS menyimpan $d[v]$ (jarak langkah, -1 = belum ditemukan) dan $\pi[v]$ (orang tua yang menemukannya).

Moore (1959) memakainya untuk labirin, Lee (1961) untuk merutekan kabel pada papan sirkuit. Buku: Subbab 13.6.1.

Prosedurnya

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)
```

*Pseudokode 13.1: simpul **ditandai saat masuk** queue, sehingga masuk paling banyak sekali; d merangkap tanda sudah dikunjungi; orang tua membentuk pohon BFS. Buku: Subbab 13.6.2.*

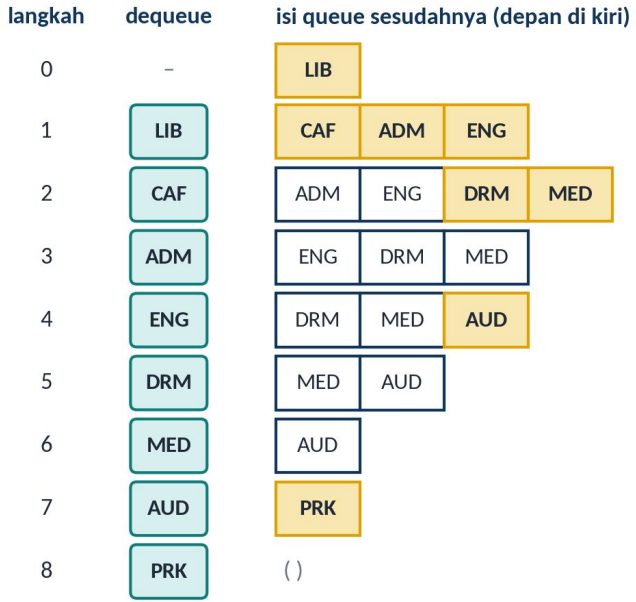
Contoh Terurai: BFS dari LIB

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)

Tabel 13.2: pada langkah 3 ENG ditolak meskipun masih di dalam queue; pada langkah 6 MED membiarkan AUD. ANX dan WHS tidak pernah masuk. Buku: Subbab 13.6.3.

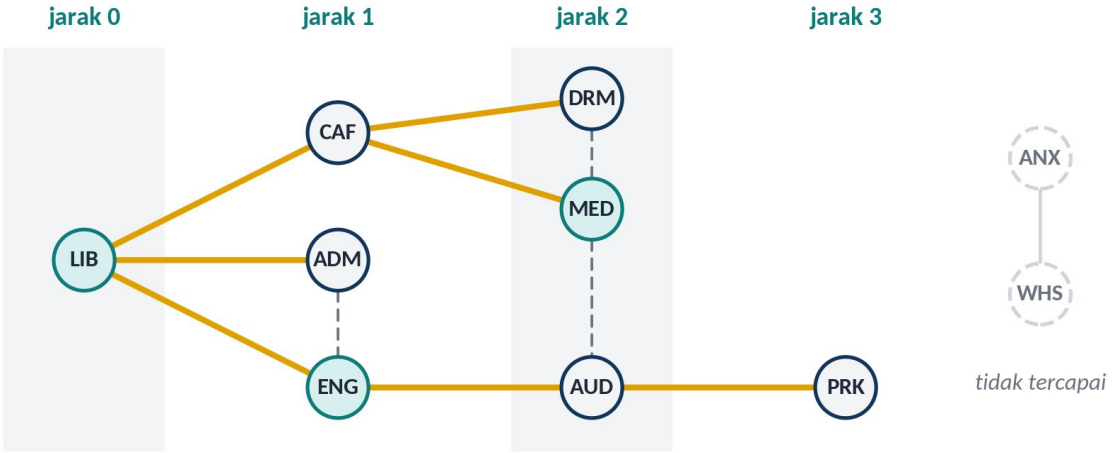
Queue Sesudah Setiap Langkah



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 13.5: queue selalu memuat satu cincin, atau ujung satu cincin yang diikuti awal cincin berikutnya. Buku: Subbab 13.6.3 dan Gambar 13.5.

Peta, Cincin demi Cincin



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.

Gambar 13.6: sisi kuning tua membentuk pohon BFS; sisi putus-putus menghubungkan simpul dari cincin yang sama atau cincin yang bersebelahan. Tidak ada sisi yang melompati cincin. Buku: Subbab 13.6.3 dan Gambar 13.6.

Mengapa BFS Menemukan Rute Terpendek dalam Langkah

Langkah 1: $d[v]$ tidak pernah terlalu kecil: rantai orang tua adalah lintasan nyata sepanjang $d[v]$ sisi.

Langkah 2: *queue* terurut menurut d dan memuat paling banyak dua nilai bersebelahan, sehingga simpul keluar dengan d tidak menurun.

Langkah 3: $d[v]$ tidak pernah terlalu besar: contoh penyangkal terkecil bertentangan dengan Langkah 2. Jadi untuk setiap sisi berlaku Persamaan (13.7):

$$d[v] \le d[u] + 1 \tag{13.7}$$

$$T_{\text{BFS}} = \Theta(V) + \sum_{u \in V} O(1 + \deg(u)) = O(V + E) \tag{13.8}$$

Persamaan (13.8): setiap simpul masuk dan keluar sekali dan setiap list dipindai sekali: waktu linear untuk graf. Dengan matriks akan menjadi $\Theta(V^2)$. *Buku: Subbab 13.6.4–13.6.5.*

BFS dalam C++

Kode 13.1 Inti `bfs` dalam `bfs.cpp`: `dist` merangkap tanda sudah dikunjungi, dipasang ketika simpul masuk *queue*.

```
    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()) {
        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];
            }
        }
```

Kode 13.1: rute dengan langkah paling sedikit ke PRK adalah LIB, ENG, AUD, PRK; WHS tetap -1 *dan program menyatakannya. Java memakai* `ArrayDeque`*, Python* `collections.deque`*;* `list.pop(0)` *di Python akan membuat BFS kuadratik. Buku: Subbab 13.6.6–13.6.7.*

Penelusuran Mendalam: Segulung Benang

Theseus di Labirin: ambil lorong yang belum dicoba dan masuk lebih dalam; bila setiap lorong menuju ruangan yang sudah dilihat, ikuti benang kembali satu ruangan.

Benang itu adalah ***stack***; rekursi menyimpankannya untuk kita.

Warna: **putih** belum ditemukan, **abu-abu** di atas benang, **hitam** selesai. Dua cap waktu: penemuan $v.d$ dan selesai $v.f$.

BFS menjelajah dengan hati-hati, cincin demi cincin; DFS dengan berani, sejauh mungkin di satu rute. Bentuk modernnya dari Tarjan (1972). Buku: Subbab 13.7.1–13.7.2.

DFS dan DFS-VISIT

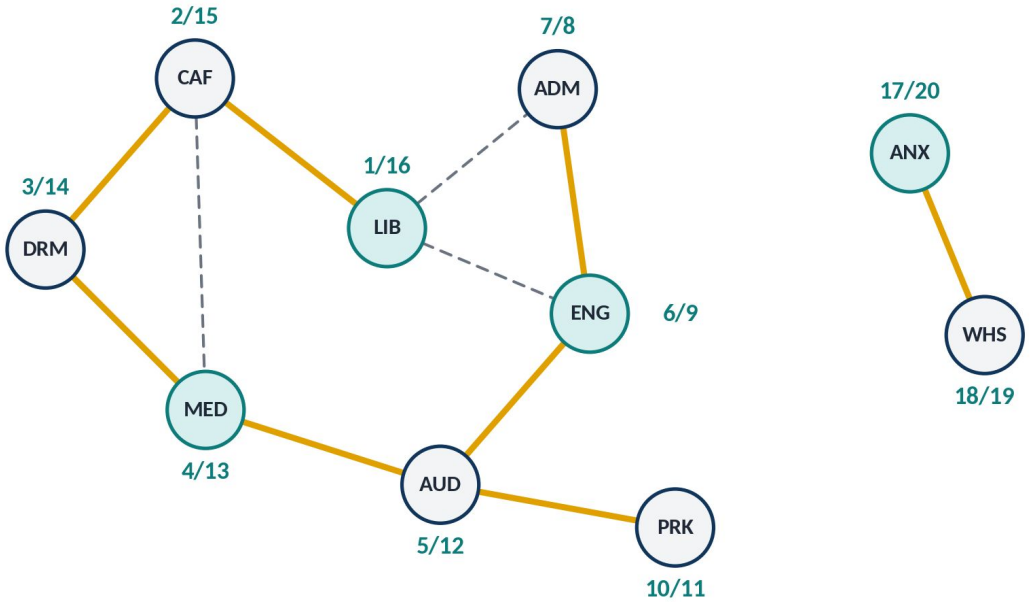
Pseudokode 13.2 DFS dan DFS-VISIT: penelusuran mendalam dengan waktu penemuan dan waktu selesai.

```
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
```

Pseudokode 13.2: satu panggilan per simpul; $\Theta(V+E)$. *Buku: Subbab 13.7.2.*

Contoh Terurai: DFS pada Peta Kampus



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.

Gambar 13.7: LIB, CAF, DRM, MED, AUD, ENG, ADM, PRK, lalu ANX, WHS; enam gedung ke dalam sebelum PRK. Pada graf tak berarah hanya ada sisi pohon dan sisi balik, dan setiap sisi balik menutup siklus. Buku: Subbab 13.7.3 dan Gambar 13.7.

Teorema Tanda Kurung

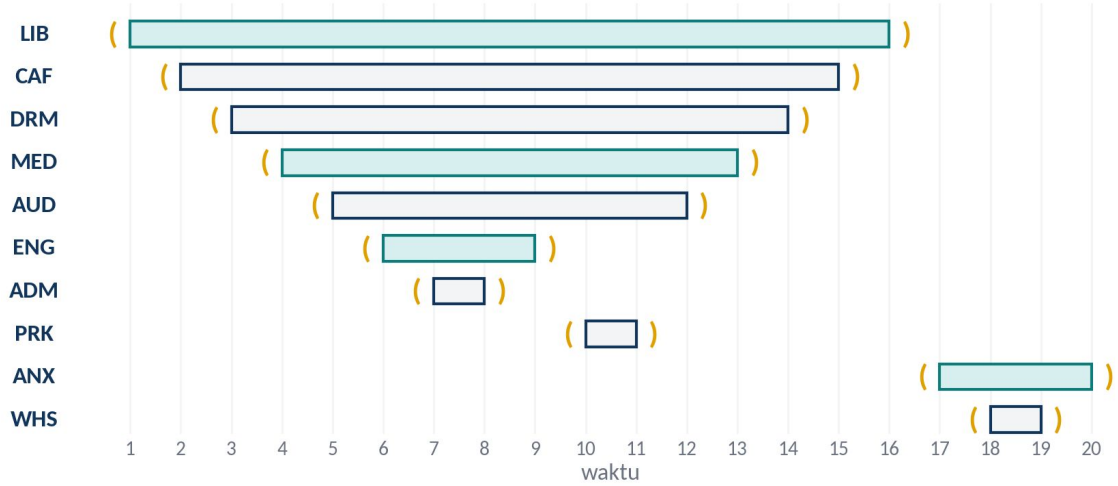
$$[u.d, u.f] \cap [v.d, v.f] = \varnothing \text{ atau } [u.d, u.f] \subset [v.d, v.f] \text{ atau } [v.d, v.f] \subset [u.d, u.f] \tag{13.9}$$

Penemuan adalah “(”, selesai adalah “)”: “(LIB (CAF (DRM (MED (AUD (ENG (ADM ADM) ENG) (PRK PRK) AUD) MED) DRM) CAF) LIB) (ANX (WHS WHS) ANX)”.

Simpul yang ditemukan saat u abu-abu didorong di atas u dan selesai lebih dulu: selangnya bersarang di dalam.

Persamaan (13.9) adalah disiplin stack dari benang itu; waktu selesai adalah kunci pengurutan topologis. Buku: Subbab 13.7.4 dan Persamaan (13.9).

Bersarang atau Saling Lepas



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.

Gambar 13.8: PRK (10 sampai 11) berada di dalam AUD (5 sampai 12) dan lepas dari ENG (6 sampai 9). Buku: Subbab 13.7.4 dan Gambar 13.8.

DFS Tanpa Rekursi

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)
```

Pseudokode 13.3: setiap entri stack adalah (simpul, indeks tetangga berikutnya), persis yang diingat panggilan rekursif, sehingga cap waktunya identik. Versi "sederhana" yang mendorong semua tetangga tidak memberi waktu selesai dan bisa memuat $|E|$ entri. Buku: Subbab 13.7.5.

DFS Rekursif dalam C++

Kode 13.2 `dfsVisit` dari `dfs.cpp`: `d[v] == 0` berarti putih; kedalaman hanya mengindentasi cetakan.

```
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
}
```

Kode 13.2. Pendampingnya, `findCycle`, melaporkan tetangga yang sudah dikunjungi dan bukan simpul asal: siklus MED, DRM, CAF. Python menghentikan rekursi pada sekitar 1.000 tingkat: untuk graf besar, hanya versi iteratif yang aman. Buku: Subbab 13.7.6–13.7.7.

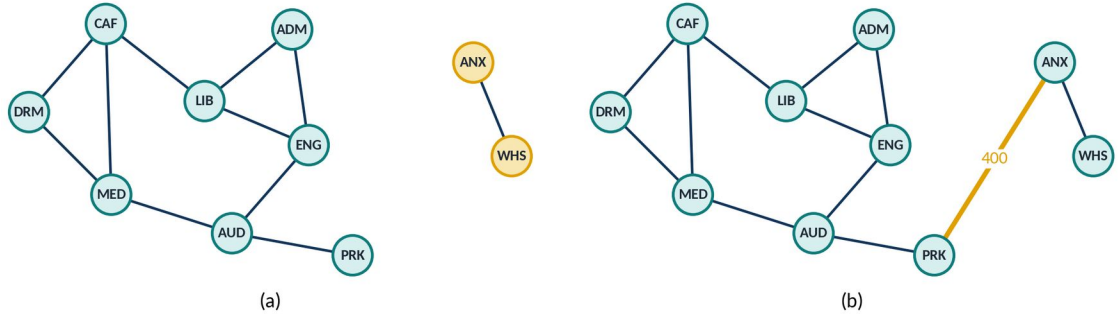
Komponen Terhubung

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
```

Pseudokode 13.4: setiap simpul diberi label dan dimasukkan ke queue sekali dalam seluruh jalannya, sehingga gelung bersarang itu tetap berongkos $\Theta(V+E)$. Lalu "apakah u dan v terhubung?" adalah `comp[u] == comp[v]`*. Buku: Subbab 13.8.1.*

Contoh Terurai: Jalan Shuttle



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.

Gambar 13.9: dua komponen; satu jalan 400 m PRK-ANX menyatukannya. AUD-PRK dan PRK-ANX adalah jembatan; LIB-CAF bukan. Union-find menangani jaringan yang tumbuh dalam waktu hampir konstan. Buku: Subbab 13.8.2–13.8.3 dan Gambar 13.9.

Urutan Topologis

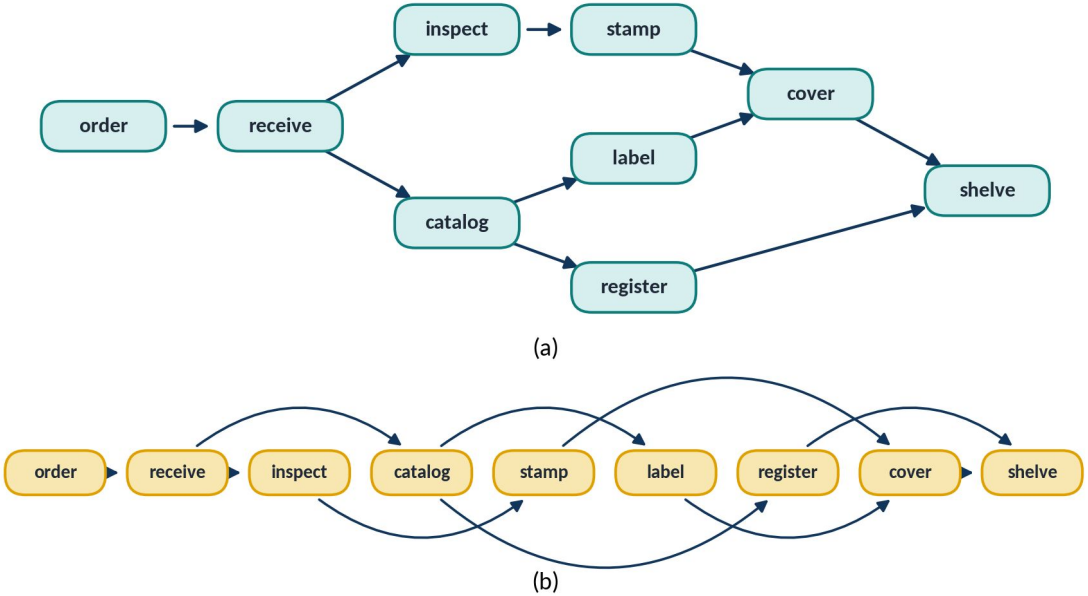
> **DEFINISI.** *Urutan topologis* sebuah graf berarah adalah daftar semua simpulnya sedemikian rupa sehingga untuk setiap panah (u, v), u muncul sebelum v.

Urutan itu ada tepat bila grafnya DAG: sebuah siklus akan memaksa suatu tugas mendahului dirinya sendiri.

Setiap DAG punya simpul berderajat-masuk 0 (berjalan mundur menyusuri panah masuk; pengulangan berarti siklus), sehingga algoritma Kahn tidak pernah macet; pada siklus ia mengeluarkan kurang dari $|V|$ simpul.

Urutan topologis biasanya tidak tunggal: inspect dan catalog boleh dikerjakan dalam urutan mana pun. Buku: Subbab 13.9.1–13.9.2.

Alur Kerja Buku Baru



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.

Gambar 13.10: order, receive, inspect, catalog, stamp, label, cover, register, shelve; panah berarti "harus mendahului". Dijajarkan dalam urutan topologis, setiap panah menunjuk ke kanan. Buku: Subbab 13.9.1 dan Gambar 13.10.

Algoritma Kahn

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
```

Pseudokode 13.5: keluarkan simpul berderajat-masuk 0, berulang-ulang; $\Theta(V+E)$. Buku: Subbab 13.9.1–13.9.2.

Contoh Terurai: Kahn pada Alur Kerja

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 $1 \to 0$	receive	receive
2	receive	inspect $1 \to 0$, catalog $1 \to 0$	inspect, catalog	inspect catalog
3	inspect	stamp $1 \to 0$	stamp	catalog stamp
4	catalog	label $1 \to 0$, register $1 \to 0$	label, register	stamp label register
5	stamp	cover $2 \to 1$	tidak ada	label register
6	label	cover $1 \to 0$	cover	register cover
7	register	shelve $2 \to 1$	tidak ada	cover
8	cover	shelve $1 \to 0$	shelve	shelve
9	shelve	tidak ada	tidak ada	(kosong)

Tabel 13.3: pada langkah 5 cover masih menunggu label; ia baru bebas pada langkah 6. Keluaran: order, receive, inspect, catalog, stamp, label, register, cover, shelve. Buku: Subbab 13.9.2.

Pengurutan Topologis dengan DFS

Pseudokode 13.6 `TOPOLOGICAL-SORT`: urutan topologis menurut waktu selesai DFS yang menurun.

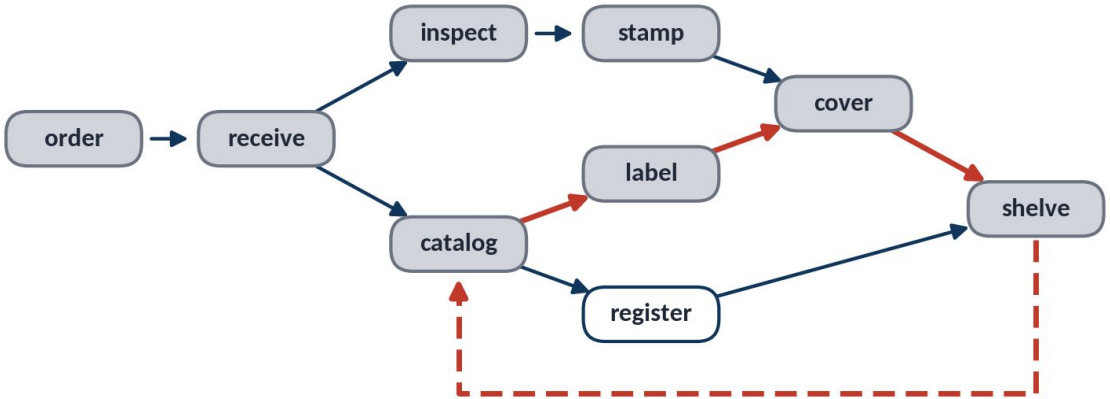
```
TOPOLOGICAL-SORT(G):
 1.  L ← an empty list
 2.  run DFS(G); each time a vertex u finishes (line 10 of DFS-VISIT),
 3.      insert u at the front of L
 4.  return L
```

Untuk setiap panah (u, v): v putih menjadi keturunan dan selesai lebih dulu; v hitam sudah selesai; v abu-abu berarti siklus. Jadi $v.f < u.f$.

Siklus berarah $\Leftrightarrow$ panah menuju simpul **abu-abu**; panah menuju simpul hitam tidak berbahaya. Tiga warna diperlukan.

Pseudokode 13.6 memberi order, receive, catalog, register, label, inspect, stamp, cover, shelve: urutan sah yang lain. Buku: Subbab 13.9.3–13.9.4.

Panah Keliru Menciptakan Siklus



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.

Gambar 13.11: dengan shelve → catalog, Kahn hanya mengeluarkan 4 dari 9 tugas dan DFS menemukan sisi balik label → cover. Pendeteksi membuktikan adanya siklus; panah mana yang salah diputuskan pustakawan. Buku: Subbab 13.9.4 dan Gambar 13.11.

Kahn dalam C++

Kode 13.3 `kahn` dari `topo.cpp`: urutan yang lebih pendek daripada V berarti ada siklus.

```
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]);
            }
        }
```

Kode 13.3; `make`, pengelola paket, dan lembar kerja semuanya mengurutkan secara topologis. Buku: Subbab 13.9.5.

Lintasan Terpendek Berbobot: Relaksasi

BFS menghitung sisi; ia tidak melihat bahwa LIB, ADM, ENG (140 m) mengalahkan LIB-ENG langsung (150 m).

Simpan jarak sementara $d[v]$, mula-mula ∞, dan perbaiki dengan **merelaksasi** sebuah sisi, Persamaan (13.10):

$$\text{jika } d[u]+w(u,v)<d[v]: d[v] \leftarrow d[u]+w(u,v), \pi[v] \leftarrow u \tag{13.10}$$

Dijkstra (1959): berulang-ulang ambil simpul yang belum final dengan d terkecil, nyatakan final, relaksasi sisi-sisinya. Bobot harus tak negatif.

Relaksasi tidak pernah membuat taksiran salah: $d[v]$ selalu panjang rute nyata, sehingga hanya bisa terlalu besar. Buku: Subbab 13.10.1–13.10.3 dan Persamaan (13.10).

Dijkstra dengan Priority Queue Malas

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))
```

Pseudokode 13.7: queue pustaka tidak dapat memperbarui entri, sehingga jarak yang lebih baik menyisipkan entri baru dan entri lama yang basi dilewati kemudian. Dengan semua bobot 1 ia berperilaku persis seperti BFS. Buku: Subbab 13.10.3.

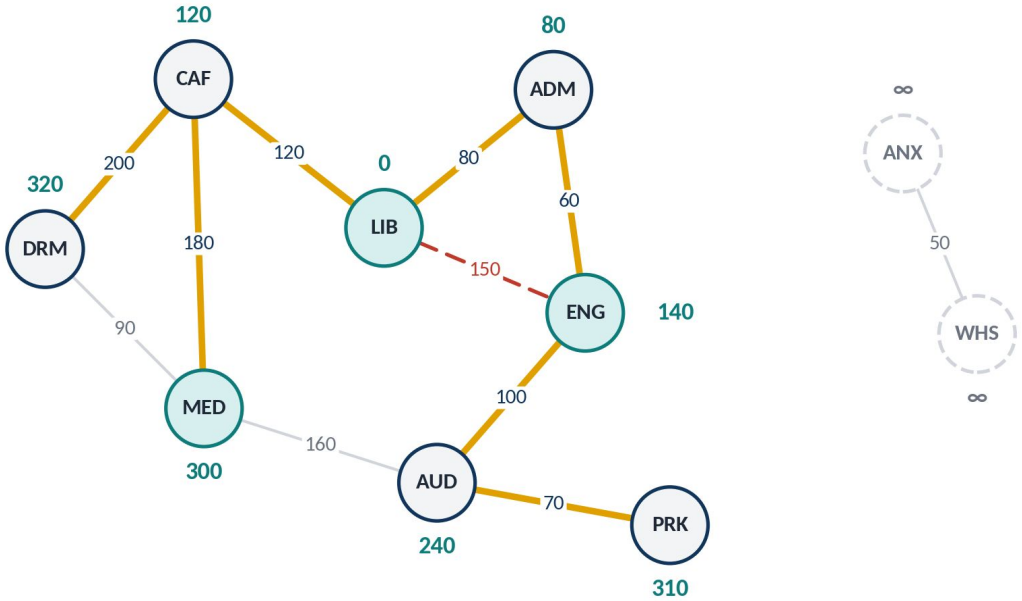
Contoh Terurai: Dijkstra dari LIB

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			$d[\text{LIB}]=0$	(0, LIB)
1	(0, LIB)	final	CAF 120, ADM 80, ENG 150	(80, ADM) (120, CAF) (150, ENG)
2	(80, ADM)	final	ENG $150 \to 140$	(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)

Tabel 13.4: pada langkah 2, ADM memperbaiki ENG dari 150 menjadi 140; pada langkah 5 (150, ENG) yang lama keluar dalam keadaan basi. Buku: Subbab 13.10.4.

Jarak Jalan Kaki Terpendek



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.

Gambar 13.12: PRK berjarak 310 m lewat ADM; LIB-ENG merah putus-putus ada pada rute dengan langkah paling sedikit tetapi tidak pada rute terpendek. Buku: Subbab 13.10.4 dan Gambar 13.12.

Mengapa Benar, dan Berapa Ongkosnya

Ketika u dikeluarkan, rute P yang lebih pendek akan meninggalkan himpunan simpul final di suatu y dengan $d[y] \le \text{panjang}(P) < d[u]$, sehingga y akan keluar lebih dulu.

Langkah itu memerlukan $w \ge 0$: dengan $w(s,a)=2$, $w(s,b)=3$, $w(b,a)=-2$, Dijkstra memfinalkan a pada 2 dan melewatkan 1. Bobot negatif memerlukan Bellman-Ford.

$$T_{\text{Dijkstra}} = O\big((V+E)\log V\big) \tag{13.11}$$

Persamaan (13.11): semua yang tidak linear berasal dari priority queue; Bab 14 membuka kotak itu. Pemindaian larik memberi $\Theta(V^2)$, lebih baik untuk graf padat. Buku: Subbab 13.10.5–13.10.6.

Dijkstra dalam C++

Kode 13.4 Gelung `dijkstra` dalam `dijkstra.cpp`; *queue* `pq` dideklarasikan dengan `std::greater<Item>`, yang mengubah *max-heap* bawaan menjadi *min-heap*.

```
    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});
                std::printf(" %s=%d", CODE[e.to], dist[e.to]);
            }
        }
        std::printf("\n");
    }
```

Kode 13.4; `PriorityQueue` *Java dan* `heapq` *Python sudah berupa min-heap. Buku: Subbab 13.10.7.*

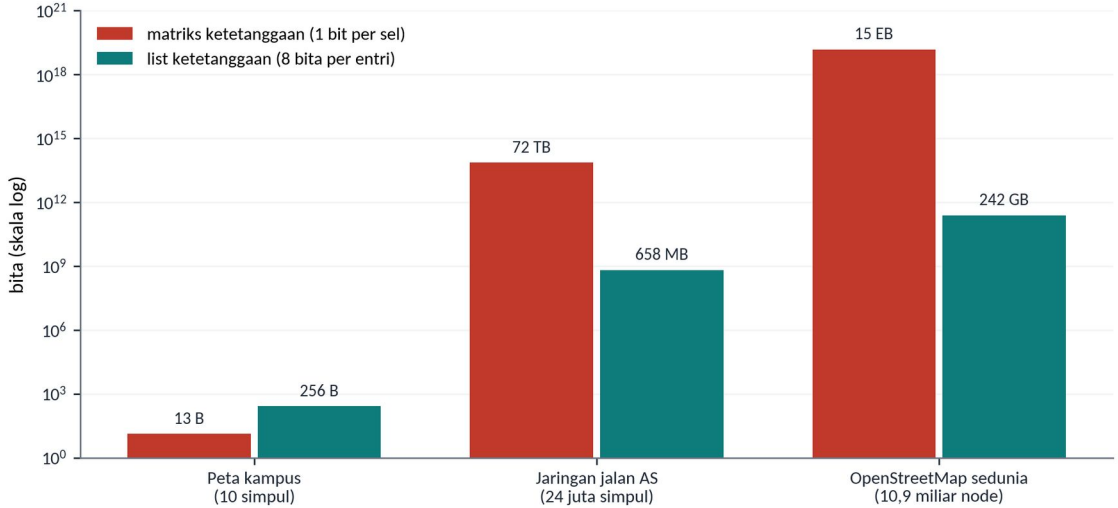
Graf pada Skala Dunia Nyata

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 [16]	3,60 miliar orang aktif harian	tidak dipublikasikan	tidak dipublikasikan	≈ 1,6 EB	bergantung pada banyaknya sisi

Tabel 13.5: OpenStreetMap memuat 10.882.374.688 simpul pada 23 September 2026; aplikasi Meta punya 3,60 miliar orang aktif harian pada Juni 2026. Matriks graf sebesar itu tidak mungkin ada; list ketetanggaan muat di sebuah server. Buku: Subbab 13.11.1–13.11.2.

Matriks lawan List, Diplot



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.

Gambar 13.13: pilih representasi menurut E, bukan V; bidik waktu linear; dan pada jaringan jalan DFS masuk jutaan simpul ke dalam, sehingga kode produksi memakai stack eksplisit. Buku: Subbab 13.11.3 dan Gambar 13.13.

Memilih Algoritma

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*	$O((V+E)\log V$	$O(V+E)$

Tabel 13.6: gelung yang sama, "keluarkan simpul dari sebuah wadah dan lihat tetangganya", menjadi BFS dengan queue, DFS dengan stack, dan Dijkstra dengan priority queue. Buku: Subbab 13.12.

Profitina dalam Praktik

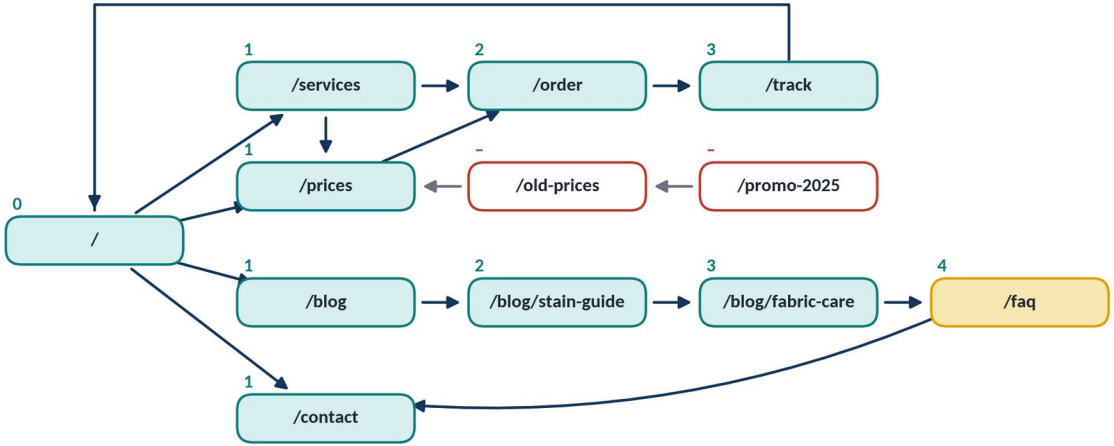
Situs web adalah **graf berarah**: halaman adalah simpul, *hyperlink* adalah panah; *crawler*, termasuk milik mesin penjawab AI, mengikutinya.

Terjangkau dari beranda? Satu BFS; halaman di *sitemap* yang tidak terjangkau adalah *halaman yatim*.

Berapa klik dalamnya? Jarak langkah. **Paling banyak ditaut?** Derajat-masuk. **Ada putaran?** Tentu; tandai halaman yang sudah dikunjungi.

Profitina adalah platform visibilitas AI (GEO); kita melukiskan bentuk setiap pertanyaan, bukan implementasi produknya. Buku: Subbab 13.13.

Contoh Terurai: Situs Profitina Laundry



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.

Gambar 13.14: `/prices` punya tiga tautan masuk; `/faq` berada empat klik ke dalam; `/promo-2025` dan `/old-prices` adalah halaman yatim. Program membuat strukturnya tampak; perbaikannya keputusan pemilik situs. Buku: Subbab 13.13 dan Gambar 13.14.

Kesalahan yang Sering Terjadi

Lupa menandai simpul yang dikunjungi, atau menandai dalam BFS ketika simpul *keluar* dari *queue*.

Menyimpan sisi tak berarah hanya ke satu arah; **matriks untuk graf besar yang jarang**; `list.pop(0)` **sebagai *queue* Python**.

BFS untuk rute berbobot; **Dijkstra dengan bobot negatif**.

Menganggap orang tua sebagai siklus; **dua warna untuk siklus berarah**; **rekursi terlalu dalam**; **bergantung pada satu urutan kunjungan tertentu**.

Pemeriksaan jabat tangan langsung menangkap arah yang hilang: panjang semua list harus berjumlah $2|E|$. *Buku: Subbab 13.14.*

Ringkasan: Lima Hal untuk Diingat

1. $G=(V,E)$; derajat berjumlah $2|E|$; graf nyata itu jarang, jadi simpanlah sebagai *list* ketetanggaan.

2. BFS dengan *queue* menemukan rute dengan langkah paling sedikit dalam $\Theta(V+E)$; tandai saat masuk.

3. DFS dengan *stack* memberi waktu penemuan dan selesai yang bersarang seperti tanda kurung; sisi balik berarti siklus.

4. Komponen: satu penelusuran per simpul yang belum berlabel. Urutan topologis: derajat-masuk Kahn, atau waktu selesai yang menurun.

5. Dijkstra memfinalkan simpul terdekat dan merelaksasi sisi-sisinya: $O((V+E)\log V)$ dengan *heap*, hanya untuk bobot tak negatif.

Buku: Subbab 13.15.

Pertemuan Berikutnya

Bab 14 — Heap dan Priority Queue. *Binary heap* di balik `std::priority_queue`, `PriorityQueue`, dan `heapq`: mengapa sisip dan ambil-minimum berongkos $O(\log n)$.

Decrease-key, *heapsort*, dan Dijkstra dengan *heap* sungguhan.

Sebelum itu: kerjakan Soal Latihan 13.1, 13.4, 13.8, dan 13.10, dalam C++ lalu dalam Java atau Python.

Penyelesaiannya, lengkap dengan program, ada di buku. Buku: Subbab 13.16 dan 13.17.

Kuliah 14 — Heap dan Priority Queue

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 14

Dosen: Irfan Subakti — yifana@gmail.com

Hari ini kita membuka kotak yang dipakai Bab 3, 5, dan 13 tanpa melihat isinya: binary heap, pohon yang tinggal di dalam larik biasa. Buku: Bab 14, Subbab 14.1–14.18.

Cakupan Pertemuan Ini

1. ***Priority queue*** — melayani yang paling mendesak lebih dulu, dan mengapa larik serta pohon pencarian tidak cukup.
2. **Pohon *complete* di dalam larik** — rumus indeks, tinggi, letak daun; sifat *heap*.
3. ***Sift-up* dan *sift-down*** — dan di sekelilingnya *insert*, *extractMax*, *getMin*, *increaseKey*, *decreaseKey*, *isMaxHeap*.
4. **Membangun dalam waktu linear, dan *heap sort* secara utuh.**
5. **Pustaka dan aplikasi** — `std::priority_queue`, `PriorityQueue`, `heapq`; *top-k*, penggabungan k jalur, Dijkstra.

Contoh berjalan: permintaan pesanan buku di Pustaka Ceria, masing-masing dengan skor urgensi; meja sirkulasi selalu melayani yang paling mendesak berikutnya. Buku: Subbab 14.1.

Titik Tolak: Tiga Janji

Subbab 3.10 mengurutkan dengan ***heap sort*** dan menjanjikan pembahasan *heap* yang semestinya.

Subbab 5.15 bertemu ***priority queue*** (unit gawat darurat melayani yang paling mendesak, bukan yang datang lebih dulu) dan mendapati bahwa larik biasa membuat satu operasi lambat.

Subbab 13.10 menjalankan **Dijkstra** dengan *priority queue* pustaka dan membiarkan terbuka cara kerjanya.

Bab 11 memberi kosakata pohon, dan bahkan menyimpan pohon biner dalam larik tanpa pointer.

Binary heap tidak memerlukan pointer: induk dan anak hanya sejauh satu perkalian atau pembagian dengan dua. Sekitar 20 langkah untuk sejuta kunci. Buku: Subbab 14.1.

Priority Queue

DEFINISI. *Max-priority queue* adalah kumpulan kunci dengan operasi `insert(x)`, `getMax()`, `extractMax()`, `increaseKey(i, k)`, dan `isEmpty()`. *Min-priority queue* memiliki `getMin`, `extractMin`, dan `decreaseKey` sebagai gantinya.

Meskipun namanya demikian, ia bukan *first-in, first-out*: urutan kedatangan hanya penting untuk memecah seri, dan itu pun hanya bila kita memintanya.

DENGAN BAHASA SEHARI-HARI. *Priority queue* tidak pernah memerlukan seluruh kumpulan terurut, hanya butir mana yang lebih dulu. *Heap* menyimpan urutan sebanyak itu saja, tidak lebih.

Min-priority queue melayani kunci terkecil: tenggat yang lebih awal, jarak yang lebih pendek. Buku: Subbab 14.2.1.

Operasi yang Akan Kita Bangun

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

***Tabel 14.1**: ongkosnya adalah inti bab ini, jadi kita melihatnya sebelum mendapatkannya. Buku: Subbab 14.2.1.*

Mengapa Struktur yang Jelas Gagal

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

Larik tak terurut bekerja terlalu sedikit saat butir datang, larik terurut terlalu banyak; *heap* menyimpan urutan **secukupnya**, murah dipulihkan sesudah setiap perubahan.

***Tabel 14.2**. Williams menerbitkan binary heap pada Juni 1964 bersama heapsort; enam bulan kemudian Floyd membangunnya dari bawah ke atas dalam waktu linear. Enam puluh tahun kemudian ketiga pustaka baku masih memakainya. Buku: Subbab 14.2.2–14.2.3.*

Pohon Biner Complete

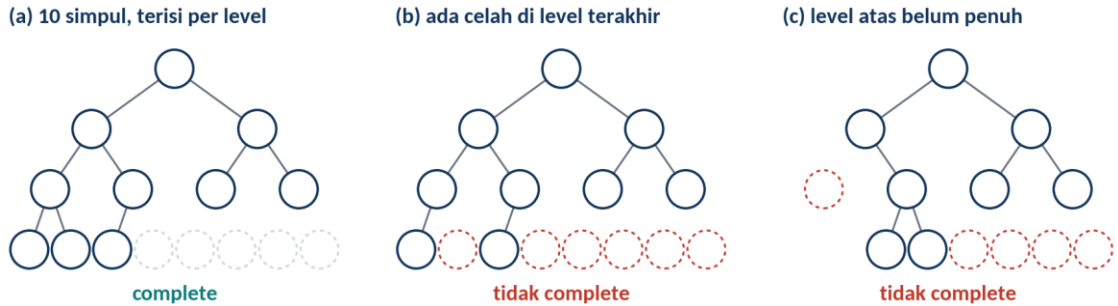
> **DEFINISI.** Pohon biner bertinggi h disebut *complete* bila setiap level $0,1,\ldots,h-1$ penuh, dan simpul-simpul level terakhir h dirapatkan sejauh mungkin ke kiri, tanpa celah.

Pohon *complete* dengan n simpul ditentukan oleh n saja: beri nomor simpulnya, dan nomor itu menyatakan letak setiap simpul.

Cormen dkk. menyebut bentuk ini *nearly complete*; namanya berbeda, bentuknyalah yang penting.

Pohon itu tumbuh di tepat satu tempat, posisi kosong berikutnya pada level terakhir. Buku: Subbab 14.3.1.

Complete, dan Bukan Complete



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.

***Gambar 14.1**: (b) punya celah di level terakhirnya; (c) kehilangan sebuah simpul di atas level terakhir. Buku: Subbab 14.3.1 dan Gambar 14.1.*

Menomori Level demi Level

Nomori simpul level demi level, kiri ke kanan, dan simpan simpul i di slot larik i. Dengan akar di indeks 0, Persamaan (14.1):

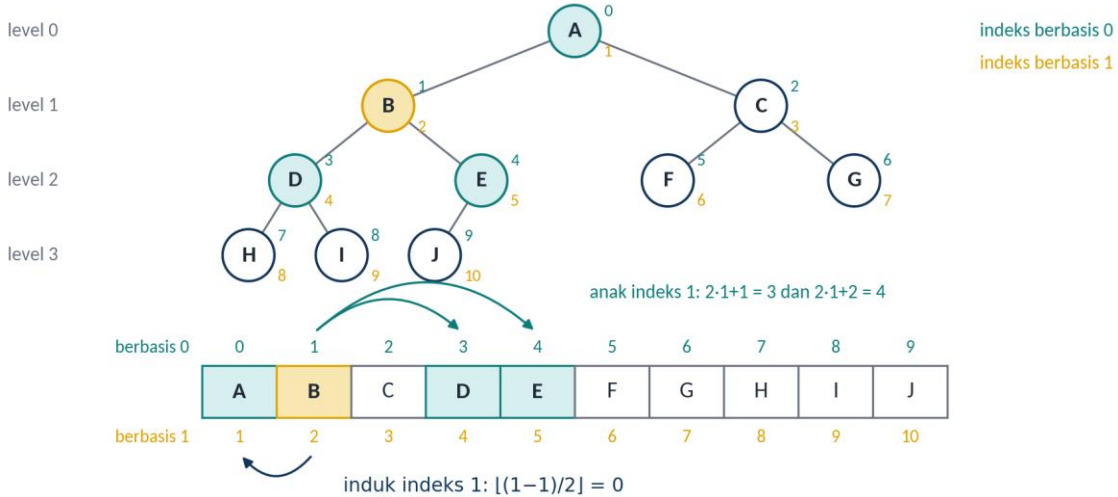
$$\text{parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor, \qquad \text{left}(i) = 2i + 1, \qquad \text{right}(i) = 2i + 2. \tag{14.1}$$

Dengan akar di indeks 1 (Williams, Cormen dkk.), Persamaan (14.2):

$$\text{parent}(k) = \left\lfloor \frac{k}{2} \right\rfloor, \qquad \text{left}(k) = 2k, \qquad \text{right}(k) = 2k + 1. \tag{14.2}$$

Substitusi $k = i + 1$ mengubah yang satu menjadi yang lain. Buktinya menghitung $2^d - 1$ simpul di atas level d; pada pohon bercelah rumus itu gagal, jadi heap harus tetap complete. Semua program di sini memakai indeks berbasis 0. Buku: Subbab 14.3.2–14.3.3.

Pohon dan Larik, Berdampingan



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.

***Gambar 14.2**: B di indeks 1 punya anak D dan E di 3 dan 4, dan induk A di 0; E di indeks 4 hanya punya anak kiri J di 9. Buku: Subbab 14.3.2 dan Gambar 14.2.*

Aritmetika Indeks

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	$2i+1 < n$	$2k \leq n$
adalah daun	$i \geq \lfloor n/2 \rfloor$	$k > \lfloor n/2 \rfloor$

Tabel 14.3*: pilih satu penomoran dan pakailah di mana-mana; $2i$ dan $2i+1$ pada larik berbasis 0 membuat akar menjadi anak kirinya sendiri. Buku: Subbab 14.3.2.*

Tinggi, dan Letak Daun

$$2^h \leq n \leq 2^{h+1} - 1. \tag{14.3}$$

$$h = \lfloor \lg n \rfloor. \tag{14.4}$$

$$\text{simpul internalnya adalah } 0,1, \ldots, \lfloor n/2 \rfloor - 1, \quad \text{dan daunnya adalah } \lfloor n/2 \rfloor, \ldots, n - 1. \tag{14.5}$$

Menurut Persamaan (14.3) dan Persamaan (14.4), sepuluh kunci bertinggi 3 dan sejuta kunci bertinggi 19. Menurut Persamaan (14.5) daun-daun adalah separuh kedua larik: paling sedikit separuh dari semua simpul. Buku: Subbab 14.3.4.

Sifat Heap

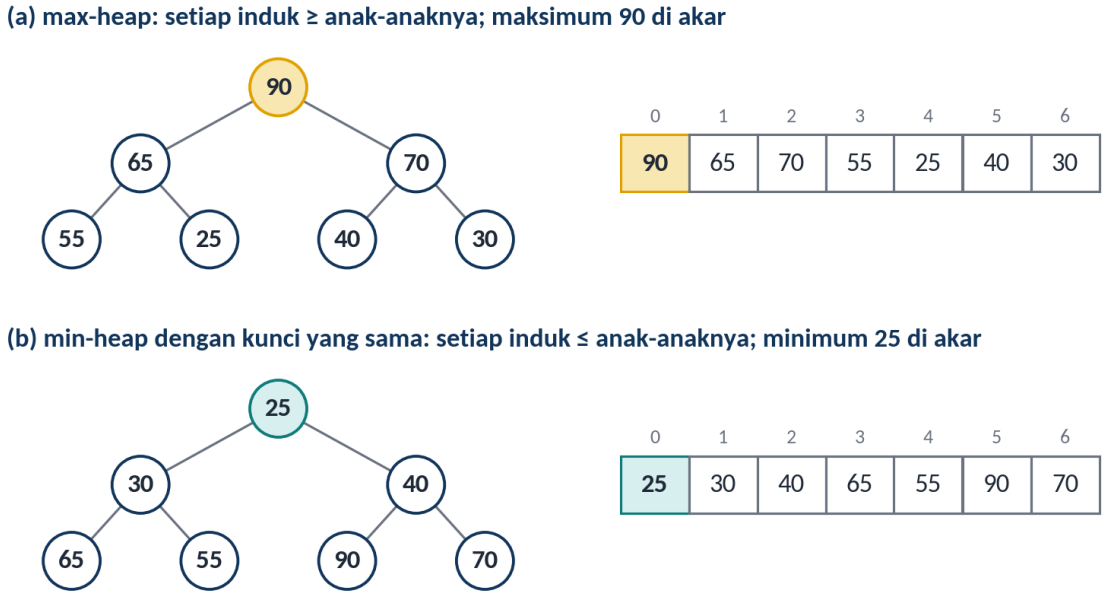
DEFINISI. Larik $A[0..n-1]$, dibaca sebagai pohon biner *complete*, adalah *max-heap* bila setiap simpul selain akar tidak lebih besar daripada induknya, Persamaan (14.6), dan *min-heap* bila tidak lebih kecil, Persamaan (14.7).

$$A[\text{parent}(i)] \geq A[i] \qquad \text{untuk setiap } i = 1,2,\ldots,n-1. \tag{14.6}$$

$$A[\text{parent}(i)] \leq A[i] \qquad \text{untuk setiap } i = 1,2,\ldots,n-1. \tag{14.7}$$

Sifat itu hanya membandingkan simpul dengan induknya, namun dengan mengikuti induk sampai akar, $A[0]$ adalah kunci terbesar, dan setiap lintasan akar-ke-daun tidak naik. Buku: Subbab 14.4.1–14.4.2.

Max-Heap dan Min-Heap



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.

***Gambar 14.3**: tujuh kunci yang sama; pada max-heap lintasan 90, 65, 55 menurun, dan pada min-heap setiap perbandingan menunjuk ke arah sebaliknya. Buku: Subbab 14.4.1 dan Gambar 14.3.*

Apa yang Tidak Dijanjikan Heap

Saudara tidak terurut: 65 di kiri 70 tidak apa-apa, dan menukar kedua subpohon menghasilkan *heap* lain.

Level tidak terurut, dan **lariknya tidak terurut**: 90, 65, 70, 55, 25, 40, 30.

Heap bukan pohon pencarian: sebuah kunci bisa ada di subpohon mana saja, sehingga mencari kunci sembarang memindai sampai n elemen.

> **PERINGATAN.** *Heap* juga nama daerah memori tempat `new` mengalokasikan (Subbab 1.3.5). Kedua makna itu hanya berbagi nama.

Urutan yang lebih lemah ini adalah harga kecepatan: urutan parsial yang murah diperbaiki, cukup untuk menjawab satu-satunya pertanyaan priority queue. Buku: Subbab 14.4.3.

Sift-Up: Kunci yang Terlalu Besar

Pseudokode 14.1 `SIFT-UP`: biarkan kunci di indeks i naik sampai induknya paling sedikit sama besar atau sampai ia mencapai akar.

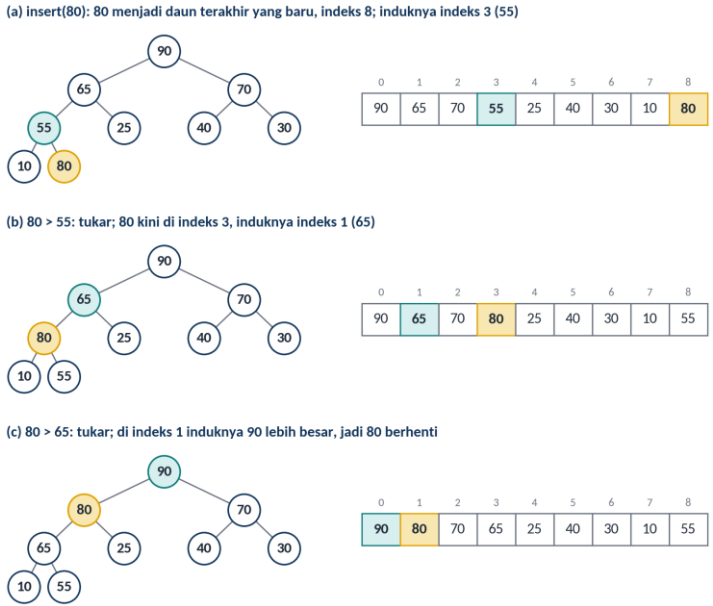
```
SIFT-UP(A, i)                          // A[i] may be larger than its parent
 1  while i > 0 and A[parent(i)] < A[i]
 2      exchange A[i] with A[parent(i)]
 3      i = parent(i)
```

Invarian: sifatnya berlaku untuk setiap $j \neq i$, dan induk i paling sedikit sebesar anak-anak i.

Setiap iterasi naik satu level: paling banyak $\lfloor \lg n \rfloor$ pertukaran dan $\lfloor \lg n \rfloor + 1$ perbandingan, $O(\lg n)$.

***Pseudokode 14.1**. Disebut juga heapify-up, bubble-up, atau swim. Sebuah penyempurnaan menyalin induk yang lebih kecil ke bawah ke dalam lubang dan menulis kuncinya sekali (Soal Latihan 14.14). Buku: Subbab 14.5.*

Contoh Terurai: Sift-Up untuk 80



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.

***Gambar 14.4**: induk indeks 8 adalah indeks 3 (55), ditukar; lalu indeks 1 (65), ditukar; lalu akar 90 lebih besar, berhenti. Dua pertukaran, tiga perbandingan. Buku: Subbab 14.5.1 dan Gambar 14.4.*

Sift-Down: Kunci yang Terlalu Kecil

Pseudokode 14.2 `SIFT-DOWN`: biarkan kunci di indeks i turun, menukarnya dengan anak yang lebih besar, sampai tidak ada anak yang lebih besar.

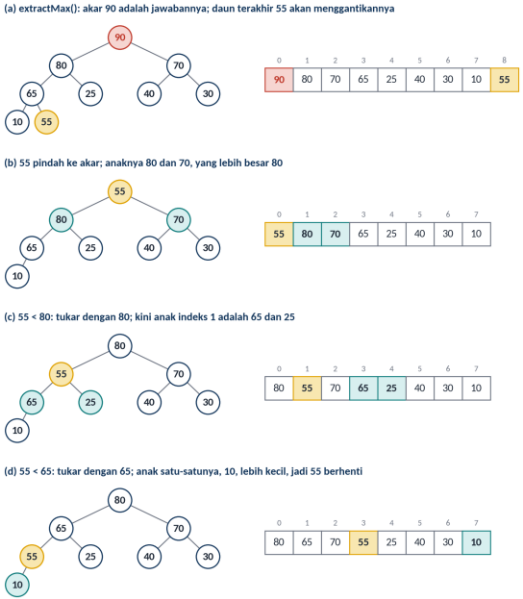
```
SIFT-DOWN(A, i, n)                  // heap is A[0..n-1]; A[i] may be too small
 1  loop
 2      largest = i
 3      l = 2i + 1;  r = 2i + 2
 4      if l < n and A[l] > A[largest]  then largest = l
 5      if r < n and A[r] > A[largest]  then largest = r
 6      if largest == i  then return        // A[i] ≥ both children
 7      exchange A[i] with A[largest]
 8      i = largest
```

Tukar dengan anak yang **lebih besar**: ia menjadi paling sedikit sebesar kunci yang turun dan saudaranya.

$$\text{perbandingan SIFT-DOWN} \le 2\lfloor \lg n \rfloor, \qquad \text{pertukaran} \le \lfloor \lg n \rfloor. \tag{14.8}$$

Pseudokode 14.2 *menerima ukuran n sebagai parameter, sebab heap sort melakukan sift di bagian depan larik. Persamaan (14.8): dua kali perbandingan sift-up, dua kandidat per level. Dari simpul bertinggi h: paling banyak h pertukaran. Buku: Subbab 14.6.*

Contoh Terurai: Sift-Down untuk 55



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.

***Gambar 14.5**: anak 80 dan 70, tukar dengan 80; anak 65 dan 25, tukar dengan 65; satu-satunya anak, 10, lebih kecil, berhenti. Buku: Subbab 14.6.1 dan Gambar 14.5.*

Kedua Perbaikan dalam C++

Kode 14.1 `siftUp` dan `siftDown` dari `max_heap.cpp`.

```
    // 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 14.1**: `l < n` dan `r < n` menjaga anak yang mungkin tidak ada. Setiap operasi lain hanyalah beberapa baris di sekitar salah satu dari kedua gelung ini. Buku: Subbab 14.12.2.*

`insert`: Tambahkan, lalu Naikkan

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)
```

```
  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
```

***Pseudokode 14.3**: daun terakhir yang baru menjaga pohon tetap complete. Kerjakan `insert(90)` dengan tangan: 90 di indeks 5, ditukar dengan 40, lalu dengan 70; ia mencapai akar. $O(\lg n)$. Buku: Subbab 14.7.2.*

`extractMax`: Daun Terakhir Naik

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
```

Hanya membuang daun *terakhir* yang menjaga pohon tetap *complete*; jadi pindahkan ia ke akar lalu biarkan tenggelam.

Pada sembilan permintaan pesanan: mengembalikan 90, lalu 80; *heap*-nya menjadi 70, 65, 40, 55, 25, 10, 30.

***Pseudokode 14.4**: bentuk dulu, dengan memindahkan daun terakhir; urutan kemudian, dengan sift. Itulah metode di balik setiap operasi heap. Paling banyak $2\lfloor \lg n \rfloor$ perbandingan. Buku: Subbab 14.7.3.*

`insert` dan `extractMax` dalam C++

Kode 14.2 `insert`, `getMax`, dan `extractMax` dari `max_heap.cpp`.

```
    // 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 14.2**: heap kosong melempar `std::underflow_error` alih-alih mengembalikan nilai karangan. Buku: Subbab 14.7 dan 14.12.*

`getMin` pada Max-Heap

Pseudokode 14.5 `GET-MIN` untuk *max-heap*: minimum berada di daun, jadi pindailah daun-daun, yaitu paruh kedua larik.

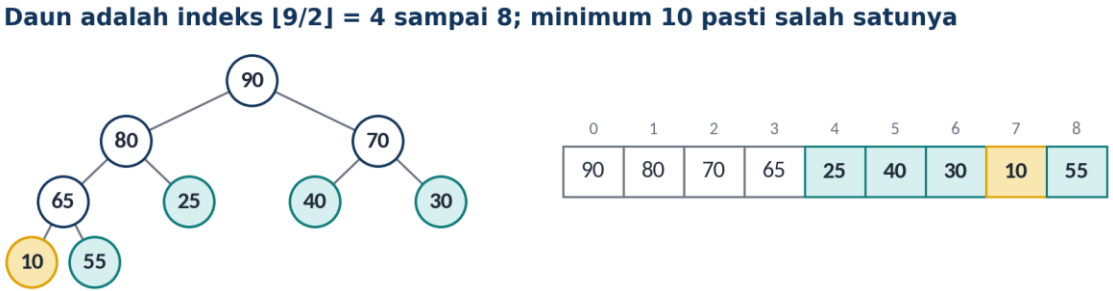
```
GET-MIN(A, n)                          // A[0..n−1] is a max-heap, n ≥ 1
 1  smallest = A[⌊n/2⌋]                // the first leaf
 2  for i = ⌊n/2⌋ + 1 to n − 1
 3      if A[i] < smallest  then smallest = A[i]
 4  return smallest
```

Suatu daun memuat kunci terkecil: ikuti anak-anak ke bawah darinya, masing-masing tidak lebih besar. Jadi pindai daun-daunnya, separuh kedua larik.

Tidak ada metode yang lebih baik: turunkan daun mana pun di bawah semua kunci lain dan lariknya tetap *max-heap*, sehingga setiap daun harus dilihat: $\Omega(n)$.

***Pseudokode 14.5**. Bila kedua ekstrem diperlukan dengan cepat: dua heap berdampingan, min-max heap, atau pohon pencarian seimbang. Buku: Subbab 14.8.*

Daun-Daun dari Sembilan Kunci



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.

Gambar 14.6*: indeks* $\lfloor 9/2 \rfloor = 4$ *sampai 8 memuat 25, 40, 30, 10, 55; empat perbandingan menemukan 10. Buku: Subbab 14.8 dan Gambar 14.6.*

`increaseKey` dan `decreaseKey`

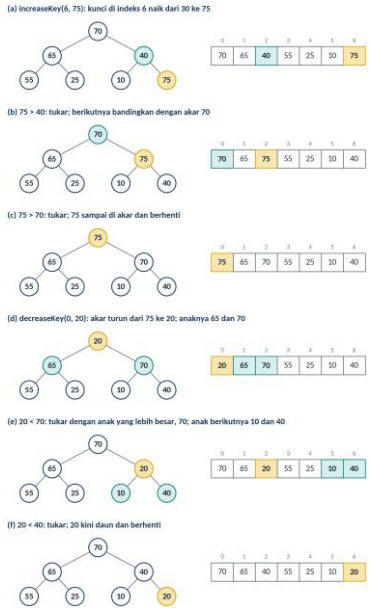
Pseudokode 14.6 `INCREASE-KEY` dan `DECREASE-KEY` pada *max-heap*: ubah kuncinya, lalu biarkan ia naik atau turun.

```
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)
```

Pseudokode 14.6*: menaikkan kunci hanya bisa membuatnya terlalu besar, menurunkan terlalu kecil; satu penugasan dan satu perbaikan, $O(\lg n)$. Tolak perubahan ke arah yang salah. Pemanggil harus tahu indeksnya, yang terus berpindah: priority queue berindeks menyimpan tabel kedua. Buku: Subbab 14.9.*

Contoh Terurai: 75 Naik, 20 Tenggelam



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.

Gambar 14.7: `increaseKey(6, 75)` *memberi 75, 65, 70, 55, 25, 10, 40;* `decreaseKey(0, 20)` *memberi 70, 65, 40, 55, 25, 10, 20. Buku: Subbab 14.9 dan Gambar 14.7.*

Membangun dari Bawah ke Atas

Pseudokode 14.7 `BUILD-MAX-HEAP`: lakukan *sift-down* pada setiap simpul internal, dari yang terakhir mundur sampai akar.

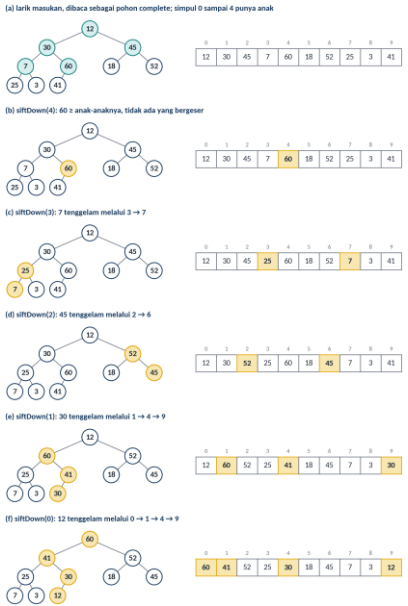
```
BUILD-MAX-HEAP(A, n)
 1  for i = ⌊n/2⌋ − 1 downto 0        // every internal node, bottom-up
 2      SIFT-DOWN(A, i, n)
```

Daun-daun sudah berupa *heap* satu simpul; lakukan *sift-down* pada setiap simpul internal dari $\lfloor n/2 \rfloor - 1$ mundur sampai akar.

Invarian: pada iterasi untuk i, setiap simpul $i+1, \ldots, n-1$ adalah akar sebuah *max-heap*, persis prasyarat *sift-down*.

Pseudokode 14.7 *adalah metode Floyd (1964). Mulailah di* $\lfloor n/2 \rfloor - 1$*, bukan di* $n-1$ *dan bukan di 0 sambil naik. Buku: Subbab 14.10.1.*

Contoh Terurai: `buildMaxHeap`



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.

***Gambar 14.8**: simpul 4, 3, 2, 1, 0 dalam urutan itu; akar 12 tenggelam melewati 60, 41, dan 30. Hasilnya 60, 41, 52, 25, 30, 18, 45, 7, 3, 12, sesudah tujuh pertukaran. Buku: Subbab 14.10.1 dan Gambar 14.8.*

Mengapa Pembangunannya Linear

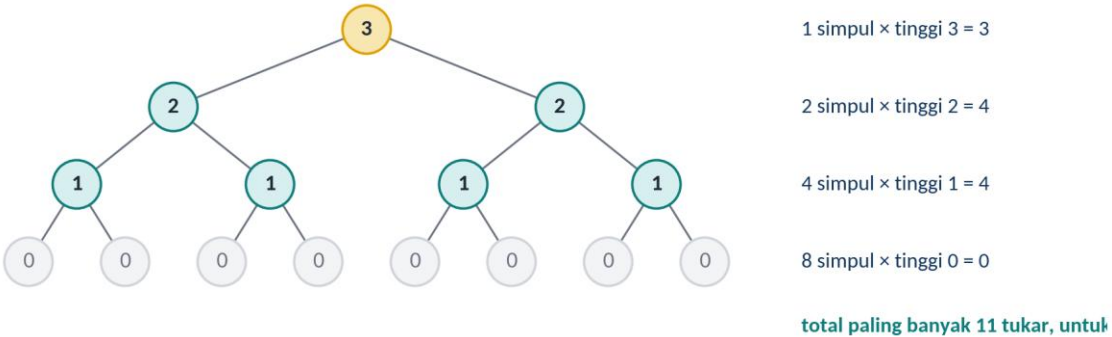
$$S = \sum_{d=0}^{H} 2^d\,(H-d) = \sum_{h=0}^{H} h\;2^{H-h} = 2^H \sum_{h=0}^{H} \frac{h}{2^h}, \tag{14.9}$$

$$T - \frac{T}{2} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots = 1, \qquad \text{jadi} \qquad \sum_{h=0}^{\infty} \frac{h}{2^h} = T = 2. \tag{14.10}$$

$$E_{\text{bangun}}(n) < n, \qquad C_{\text{bangun}}(n) < 2n, \qquad T_{\text{bangun}}(n) = \Theta(n), \tag{14.11}$$

Separuh simpul adalah daun, seperempat dapat tenggelam satu level, seperdelapan dua. Persamaan (14.9) dengan Persamaan (14.10) memberi $S < n + 1$; tepatnya $n - lg(n+1)$ untuk heap sempurna, 11 untuk 15 simpul. Maka berlaku Persamaan (14.11). Buku: Subbab 14.10.2.

Tinggi dalam Heap 15 Simpul



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.

***Gambar 14.9**: paling banyak* $8 \cdot 0 + 4 \cdot 1 + 2 \cdot 2 + 1 \cdot 3 = 11$ *pertukaran. Kerja dilakukan di tempat yang hanya sedikit kerjanya. Buku: Subbab 14.10.2 dan Gambar 14.9.*

Membangun lewat Penyisipan Memang Lebih Lambat

$$\sum_{m=2}^{n} \lfloor \lg m \rfloor = (n+1)\lfloor \lg n \rfloor - 2^{\lfloor \lg n \rfloor + 1} + 2 = \Theta(n \lg n). \tag{14.12}$$

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

Persamaan (14.12): kunci menaik masing-masing naik sampai akar, 14,69 perbandingan per kunci pada $n = 100.000$; pembangunan dari bawah tetap di bawah atau sama dengan 2,00. Penyisipan acak rata-rata sekitar 2,28 per kunci. Buku: Subbab 14.10.3.

`isMaxHeap`: Memeriksa Sifatnya

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
```

Pada 10, 7, 8, 5, 2, 3, 1 setiap simpul tidak lebih besar daripada induknya; pada 10, 7, 6, 5, 1, 8, 2 kunci 8 berada di bawah 6. Jawaban `true` harus sudah memeriksa semua $n - 1$ pasangan.

Pseudokode 14.8 *menggelung atas anak-anak, sehingga larik kosong dan larik satu elemen tidak memerlukan kasus khusus; batas gelung seperti* `(v.size() - 2) / 2` *berputar balik di C++. Buku: Subbab 14.11.*

Kelas `MaxHeap`

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()`	$O(n)$	—

***Tabel 14.4**; galat berupa eksepsi. Buku: Subbab 14.12.*

Heap Sort secara Utuh

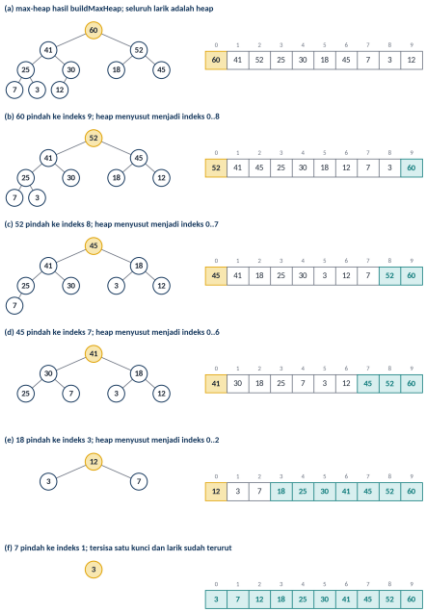
Pseudokode 14.9 `HEAP-SORT`: bangun *max-heap*, lalu berulang kali pindahkan akar ke ujung *heap* yang menyusut dan perbaiki akarnya.

```
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]
```

Invarian: $A[0..end]$ adalah *max-heap* berisi $end + 1$ kunci terkecil; $A[end + 1..n - 1]$ memuat kunci-kunci terbesar secara menaik.

Pseudokode 14.9*: fase 2 adalah* `extractMax` *yang memarkir maksimum di slot yang baru saja dilepas heap yang menyusut. Buku: Subbab 14.13.1–14.13.2.*

Contoh Terurai: Heap Sort Sepuluh Kunci



Gambar 14.10 *Heap sort* pada *heap* dari **Gambar 14.8**: sesudah setiap pertukaran dan *sift-down*, *heap* (pohon, akar berwarna kuning tua) menyusut satu dan ekor terurut (teal) bertambah satu; panel (e) dan (f) melewati langkah-langkah antara.

Gambar 14.10*: 60 pindah ke indeks 9, lalu 52 ke indeks 8, dan seterusnya; program menghitung 38 perbandingan. Buku: Subbab 14.13.1 dan Gambar 14.10.*

Ongkos Heap Sort

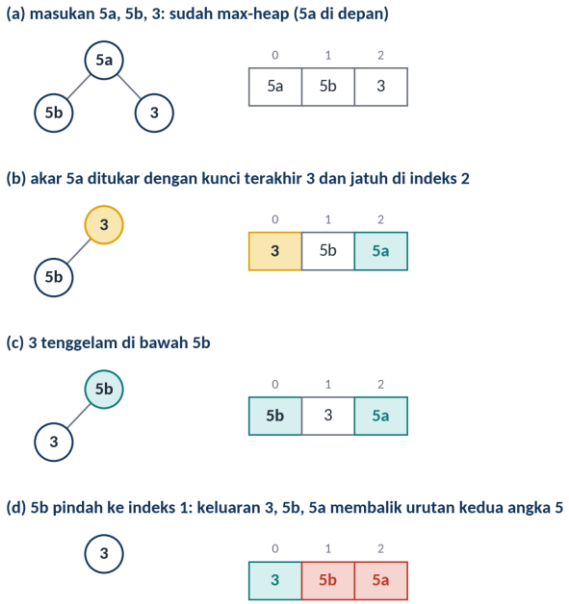
$$C_{\text{heap sort}}(n) \leq 2n + 2(n-1)\lfloor \lg n \rfloor = O(n \lg n) \tag{14.13}$$

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

*Persamaan (14.13) dalam setiap kasus, memori tambahan $O(1)$: satu-satunya pengurutan Bab 3 yang menggabungkan jaminan kasus terburuk dengan pengurutan di tempat (**Tabel 14.5**). Pemakaian cache yang buruk membuatnya lebih lambat daripada quicksort yang baik; introsort beralih kepadanya hanya bila rekursi tumbuh terlalu dalam. Buku: Subbab 14.13.3.*

Mengapa Heap Sort Tidak Stabil



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.

***Gambar 14.11**: pertukaran pertama melempar 5a dari depan ke belakang, melewati 5b. Mengurutkan buku menurut tahun menaruh Amba sebelum Pulang; `std::stable_sort` tidak. Buku: Subbab 14.13.4 dan Gambar 14.11.*

Priority Queue dalam Pustaka Baku

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

***Tabel 14.6**: `std::priority_queue` adalah max-heap yang perbandingannya berarti “dilayani lebih belakangan daripada”, dan `pop()` tidak mengembalikan apa pun; `PriorityQueue` Java adalah min-heap; tidak satu pun menyediakan `decreaseKey`. Buku: Subbab 14.14.1–14.14.2.*

heapq Python, dan Nilai Seri

Kode 14.3 Dari `library_pq.py`: tupel `(-urgency, arrival, member, title)` dalam *min-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}")
```

Kode 14.3*: tupel dibandingkan dari kiri ke kanan, sehingga nomor kedatangan memecah seri dan mencegah rekaman dibandingkan menurut nama. Python 3.14 menambahkan `heapify_max`, `heappush_max`, `heappop_max`, `heappushpop_max`, dan `heapreplace_max`. Buku: Subbab 14.14.3.*

Top-k dari Aliran Data

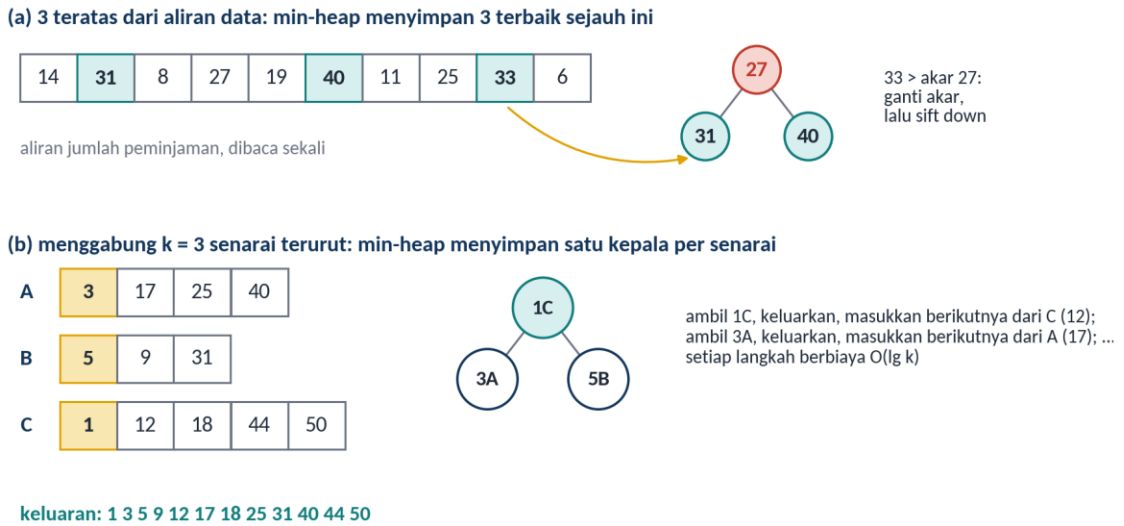
Pseudokode 14.10 `TOP-K`: simpan k butir terbaik di dalam *min-heap* yang akarnya adalah yang terlemah di antara mereka.

```
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
```

$$T_{\text{top-}k}(n) = O(n \lg k) \text{ waktu}, \qquad O(k) \text{ memori}. \tag{14.14}$$

***Pseudokode 14.10**: **min**-heap berukuran k yang akarnya adalah yang terlemah di antara yang terbaik; butir yang tidak mengalahkannya langsung dibuang. Persamaan (14.14): metode ini bekerja pada aliran data yang terlalu besar untuk disimpan. Buku: Subbab 14.15.1.*

Tiga Teratas, dan Penggabungan Tiga Jalur



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.

***Gambar 14.12**: yang terlemah yang disimpan naik 8, 14, 19, 27, 31; tiga teratas adalah 40, 33, 31. Penggabungan selalu mengeluarkan akar. Buku: Subbab 14.15.1–14.15.2 dan Gambar 14.12.*

Menggabung k Senarai Terurut

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))
```

$$T_{\text{merge}}(N, k) = O(N \lg k). \tag{14.15}$$

***Pseudokode 14.11**, Persamaan (14.15): heap-nya tidak pernah memuat lebih dari k entri. Gagasan yang sama mengurutkan data yang terlalu besar untuk memori: run terurut digabung dengan satu heap (pengurutan eksternal). Buku: Subbab 14.15.2.*

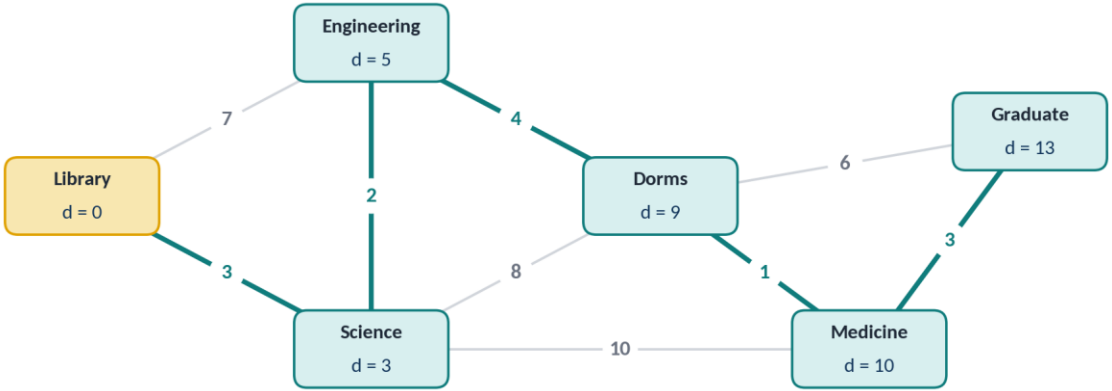
Top-k dalam C++

Kode 14.4 Bagian 1 dari `topk_merge.cpp`: `priority_queue` dengan `greater` menyimpan tiga jumlah peminjaman terbaik.

```
    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);
    }
```

***Kode 14.4**: hasilnya diperiksa terhadap pengurutan penuh. `heapq.nlargest` dan `heapq.merge` di Python mengimplementasikan kedua algoritma yang sama. Buku: Subbab 14.15.*

Kait ke Bab 13: Dijkstra



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*.

***Gambar 14.13**: menurunkan $d[v]$ adalah `decreaseKey`; tanpanya, versi malas mendorong entri baru dan melewati entri basi kemudian. Empat dari sepuluh entri basi. Buku: Subbab 14.15.3 dan Gambar 14.13.*

Ongkos Dijkstra, Dijelaskan

$$T_{\text{Dijkstra}} = O((|V| + |E|)\lg|V|). \tag{14.16}$$

Paling banyak satu dorongan per arah sisi: *heap* memuat $O(|E|)$ entri, setiap *push* atau *pop* $O(\lg|V|)$ karena $|E| < |V|^2$.

Heap berindeks dengan `decreaseKey` sungguhan menyimpan paling banyak $|V|$ entri; *Fibonacci heap* memberi $O(|E| + |V|\lg|V|)$.

Persamaan (14.16): pada graf biasa, binary heap sederhana dengan penghapusan malas biasanya yang tercepat dalam praktik. Buku: Subbab 14.15.3.

Profitina dalam Praktik

Apa yang jatuh tempo berikutnya: pemindaian ulang otomatis dengan irama berbeda membentuk *min-priority queue* berkunci waktu jatuh tempo.

Puncak papan peringkat: *top-k* dengan *min-heap* berukuran k, tanpa mengurutkan semua orang.

Dasbor dari banyak umpan: penggabungan k jalur yang boleh berhenti sesudah 20 butir terbaru.

Garis depan *crawl*: *priority queue* menurut kepentingan; *queue* FIFO adalah BFS biasa.

Servernya Python dengan FastAPI, jadi `heapq` adalah heap yang terdekat; kita melukiskan bentuknya, bukan implementasi produknya. Dalam setiap kasus heap menggantikan pemindaian atau pengurutan penuh yang berulang dengan satu langkah logaritmik. Buku: Subbab 14.16.

Kesalahan yang Sering Terjadi

Mencampur rumus berbasis 0 dan berbasis 1; ***sift* ke anak yang salah**; **membaca anak yang tidak ada**.

Membuang akar tanpa memindahkan daun terakhir; **mengira *heap* terurut**; **mencari di *heap* seperti di pohon pencarian**.

Membangun lewat n penyisipan padahal semua kunci diketahui; **mengubah kunci tanpa memperbaiki**.

Perbandingan `std::priority_queue` yang terbalik; **menyerahkan nilai seri pada kebetulan**; **batas gelung *unsigned***.

Juga: mencampuradukkan struktur data ini dengan daerah memori yang bernama sama. Buku: Subbab 14.17.

Ringkasan: Lima Hal untuk Diingat

1. Pohon *complete* tinggal di larik: anak $2i + 1$, $2i + 2$, induk $\lfloor (i - 1)/2 \rfloor$, tinggi $\lfloor \lg n \rfloor$.

2. *Max-heap*: setiap simpul tidak lebih besar daripada induknya, sehingga akar adalah maksimum; selain itu tidak ada yang terurut.

3. *Sift-up* dan *sift-down* memperbaiki satu kunci yang salah dalam $O(\lg n)$; *insert*, *extractMax*, dan perubahan kunci dibangun di atasnya.

4. Bangun dari bawah ke atas dalam $\Theta(n)$; *heap sort* dalam $O(n \lg n)$ dengan memori $O(1)$, tetapi tidak stabil.

5. Pustaka: C++ terbesar lebih dulu, Java dan Python terkecil lebih dulu; *top-k*, penggabungan k jalur, dan Dijkstra semuanya berjalan di atas *heap*.

Buku: Subbab 14.18.

Pertemuan Berikutnya

Bab 15 — Hashing dan Tabel Hash. Menemukan kunci dalam waktu harapan $O(1)$ dengan menghitung tempat tinggalnya.

Fungsi *hash*, tabrakan, *chaining* dan *open addressing*, faktor muatan dan *rehashing*.

Sebelum itu: kerjakan Soal Latihan 14.3, 14.5, 14.8, dan 14.11, dalam C++ lalu dalam Java atau Python.

Penyelesaiannya, lengkap dengan program, ada di buku. Buku: Subbab 14.19 dan 14.20.

Kuliah 15 — Hashing dan Tabel Hash

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 15

Dosen: Irfan Subakti — yifana@gmail.com

Hari ini kita berhenti membandingkan kunci dan mulai menghitung tempat tinggalnya: cari, sisip, dan hapus dalam waktu konstan secara rata-rata. Buku: Bab 15, Subbab 15.1–15.14.

Cakupan Pertemuan Ini

1. **Gagasannya** — dari $O(n)$ ke $O(\lg n)$ ke $O(1)$; pengalamatan langsung; kosakatanya.
2. **Fungsi *hash*** — pembagian, perkalian, *hash* polinomial untuk *string*, pembaruan bergulir, *avalanche*.
3. **Tabrakan** — batas ulang tahun; *separate chaining*; pengalamatan terbuka dengan nisan.
4. **Faktor muat dan *rehashing*, diukur**; kasus terburuk dan *hash flooding*.
5. **Pustaka** — `std::unordered_map` lawan `std::map`, `HashMap` Java, `dict` Python; penerapan.

Kita membangun dua tabel dengan tangan, mengukurnya, merusak satu dengan sengaja, lalu membaca apa yang sesungguhnya dilakukan pustaka baku. Buku: Subbab 15.1.

Titik Tolak: Sebuah Celah

Subbab 2.12.2: tidak ada algoritma yang hanya **membandingkan** kunci yang dapat mengalahkan $\lceil \lg(n+1) \rceil$ penyelidikan pada kasus terburuk.

Pohon pencarian seimbang (Bab 11) mencapai $O(\lg n)$, tetapi juga mesin pembanding.

Algoritma yang melakukan **aritmetika** pada kunci tidak terikat bukti itu. Kita sudah bertemu dengannya: 256 pencacah (Subbab 4.4.3), himpunan nilai yang sudah dilihat (8.7.3), *map* dari judul ke simpul (9.9).

Harganya: tidak ada urutan (tidak ada daftar terurut, kueri rentang, *successor*), dan waktu konstannya adalah **rata-rata**, bukan jaminan.

Subbab 15.8 memperlihatkan seberapa buruk kasus terburuknya dan bagaimana sistem sungguhan bertahan. Buku: Subbab 15.1.

Pustakawan yang Tidak Pernah Mencari

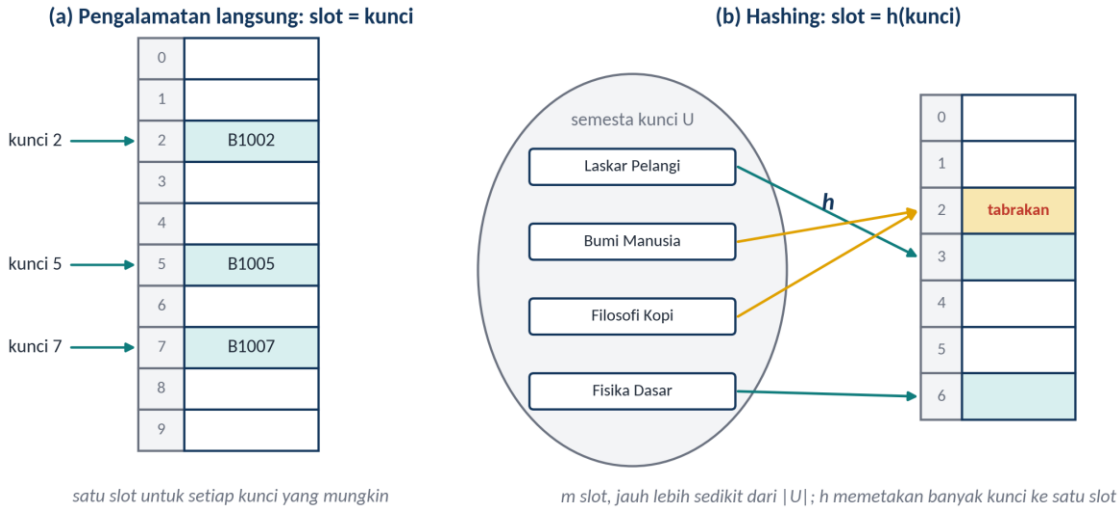
Relawan memeriksa tumpukan satu per satu: $O(n)$. Mahasiswa menjaga buku terurut dan membagi dua: $O(\lg n)$.

Kepala pustakawan punya 26 baki, A sampai Z, dan menaruh setiap buku di baki huruf pertamanya; ditanya tentang *Filosofi Kopi*, ia merogoh baki F.

Aturannya adalah **fungsi *hash***, bakinya adalah ***bucket***, *Fisika Dasar* dan *Filosofi Kopi* di baki yang sama adalah **tabrakan**.

Seribu buku berjudul "Sejarah …" akan mengubur baki S. Seluruh bab ini menyempurnakan baki-baki itu: aturan yang lebih baik, banyak baki yang tepat, rencana untuk baki bersama, dan baki tambahan ketika tumpukan membesar. Buku: Subbab 15.2.1.

Pengalamatan Langsung dan Hashing



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.

***Gambar 15.1**: (a) nomor anggota 0 sampai 999 langsung menjadi indeks larik, $O(1)$ pada kasus terburuk; tetapi tabel untuk semua kunci 64 bit memerlukan 18 eksabita. (b) Hashing mempertahankan akses cepat dan membuang larik raksasa itu. Buku: Subbab 15.2.2–15.2.3 dan Gambar 15.1.*

Kosakatanya

DEFINISI. *Fungsi hash* untuk tabel ber-m slot adalah fungsi $h: U \to \{0,1,\ldots,m-1\}$; $h(k)$ adalah *slot rumah* k. *Tabel hash* adalah larik $T[0..m-1]$ yang menyimpan k di, atau mulai dari, slot $h(k)$. Kunci berbeda dengan slot yang sama *bertabrakan*. *Faktor muat* n kunci dalam m slot adalah $\alpha = n/m$.

Memorandum IBM Luhn tahun 1953 sudah merantai rekaman yang bertabrakan; kata *hashing* (mencincang dan mengaduk) muncul di media cetak pada 1968.

Tabel hash ada di dalam setiap compiler, setiap basis data, setiap peramban, dan inti Python, yang objek, modul, dan kelasnya semuanya kamus. Buku: Subbab 15.2.3.

Perjalanan itu dalam Satu Tabel

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 "lebih tinggi atau lebih rendah": sekitar lg$n$ pertanyaan. *Hashing* itu seperti mengetahui nomor rumah: langsung berjalan ke pintunya.

***Tabel 15.1**: kolom terakhir menyatakan harga setiap struktur di luar waktu. Buku: Subbab 15.2.4.*

Apa yang Harus Dilakukan Fungsi Hash yang Baik

Deterministik, **cepat** (beberapa instruksi per bita), **seragam** pada kunci yang benar-benar disimpan, dan **memakai seluruh kunci**.

DEFINISI. Di bawah *simple uniform hashing*, setiap kunci berpeluang sama menuju slot mana pun, lepas dari kunci-kunci lain, Persamaan (15.1).

$$\Pr[h(k) = j] = \frac{1}{m} \quad \text{untuk setiap slot } j \in \{0,1,\dots,m-1\}. \tag{15.1}$$

Tidak ada fungsi tetap yang memenuhi Persamaan (15.1) untuk setiap himpunan kunci: lawan yang mengetahui h dapat memilih kunci yang bertabrakan. Asumsi ini tetap sasaran yang tepat, dan semua rata-rata dalam bab ini diturunkan di bawahnya. Buku: Subbab 15.3.1 dan Persamaan (15.1).

Contoh Terurai: Metode Pembagian

$$h(k) = k \bmod m. \tag{15.2}$$

```
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
```

Delapan harga buku, semuanya ribuan bulat: $m = 10$ hanya melihat digit terakhir; $m = 16$ hanya empat bit terakhir; bilangan prima $m = 11$ menyebar semuanya.

Persamaan (15.2): hindari $m = 2^p$ dan $m = 10^p$; pilih bilangan prima yang tidak terlalu dekat dengan pangkat 2. Langkah d dengan $\gcd(d, m) = g$ hanya menjangkau m/g slot. Buku: Subbab 15.3.2 dan Persamaan (15.2).

Metode Perkalian

$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor, \tag{15.3}$$

$$h(k) = (k \cdot s \bmod 2^w) \gg (w - p), \qquad s = \lfloor 2^w A \rfloor. \tag{15.4}$$

Persamaan (15.3) bekerja untuk m berapa pun, termasuk pangkat 2; Knuth menyarankan $A = (\sqrt{5} - 1)/2$. Untuk $k = 45000, m = 16$: bagian pecahan 0,529 …, slot 8.

Persamaan (15.4) adalah Fibonacci hashing: satu perkalian dengan `0x9E3779B97F4A7C15` *dan satu geseran, mengambil bit-bit atas. Delapan harga yang sama tersebar ke enam dari 16 slot. Buku: Subbab 15.3.3.*

Meng-hash String

Menjumlahkan kode karakter mengabaikan urutan: *kasur* dan *rusak* sama-sama berjumlah 550. Kode buku B0000 sampai B1999 berjumlah 258–286: 72 dari 101 *bucket* tetap kosong.

Obatnya: perlakukan *string* sebagai bilangan berbasis B, ***hash* polinomial** Persamaan (15.5):

$$h(s) = s_0 B^{L-1} + s_1 B^{L-2} + \cdots + s_{L-2} B + s_{L-1} = \sum_{i=0}^{L-1} s_i \, B^{L-1-i}. \tag{15.5}$$

`String.hashCode` *Java adalah Persamaan (15.5) dengan* $B = 31$ *dalam 32 bit; program kita memakai 64 bit. Kini kasur memberi 101820730 dan rusak 108880570. Buku: Subbab 15.3.4 dan Persamaan (15.5).*

Contoh Terurai: Aturan Horner pada “B1001”

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
```

```
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
```

Pseudokode 15.1*: satu perkalian dan satu penjumlahan per karakter. 62459810 persis* `"B1001".hashCode()` *di Java. Dengan aritmetika unsigned 64 bit, reduksi modulo* 2^{64} *gratis; Python menulis* `& MASK`*. Buku: Subbab 15.3.4.*

Fungsi Hash dalam C++

Kode 15.1 *Fibonacci hashing*, penjumlahan karakter yang buruk, dan *hash* polinomial, dari `hash_functions.cpp`.

```
// 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*: pada 2000 kode dan 101 bucket, hash polinomial memakai setiap bucket dan tidak pernah menaruh lebih dari 23 kode di satu bucket, mendekati ideal 19,8; penjumlahan menumpuk 150 di satu bucket. Buku: Subbab 15.3.7.*

Bergulir dan Avalanche

$$h_i = (h_{i-1} - s_{i-1}B^{W-1}) \cdot B + s_{i+W-1}. \tag{15.6}$$

Persamaan (15.6): *hash* jendela berikutnya dalam $O(1)$, jantung Rabin-Karp dan *content-defined chunking*.

Avalanche: membalik satu bit masukan seharusnya membalik separuh bit keluaran. Identitas 1,00, `k*31+7` 3,67, Fibonacci 16,02 tetapi buta di bit 63, *finalizer* SplitMix64 31,97.

> **HASH UNTUK TABEL BUKAN HASH UNTUK KATA SANDI.** *Hash* tabel mengejar kecepatan; *hash* kata sandi (Argon2id, bcrypt) sengaja lambat dan rakus memori.

`std::hash<int>` *GCC adalah identitas; banyak tabel modern mengakhiri setiap hash dengan langkah pengadukan yang murah. Buku: Subbab 15.3.5–15.3.7 dan Persamaan (15.6).*

Soal Ulang Tahun

$$\Pr[\text{tak ada tabrakan di antara } n \text{ kunci}] = \prod_{i=1}^{n-1}\left(1 - \frac{i}{m}\right). \tag{15.7}$$

$$\Pr[\text{tak ada tabrakan}] \approx \exp\left(-\sum_{i=1}^{n-1}\frac{i}{m}\right) = \exp\left(-\frac{n(n-1)}{2m}\right). \tag{15.8}$$

Tabrakan tidak dapat dihindari (prinsip sarang merpati). Yang mengejutkan adalah betapa *cepat*: 23 orang memberi peluang setengah untuk ulang tahun yang sama.

Persamaan (15.7): kunci ke-$(i+1)$ menghindari i slot terisi dengan peluang $1 - i/m$. Dengan $1 - x \approx e^{-x}$, hasil kalinya menjadi Persamaan (15.8). Buku: Subbab 15.4.1–15.4.2.

Akar Kuadrat Adalah Seluruh Ceritanya

$$n_{1/2} \approx \sqrt{2\ln 2 \cdot m} \approx 1.1774\sqrt{m}. \tag{15.9}$$

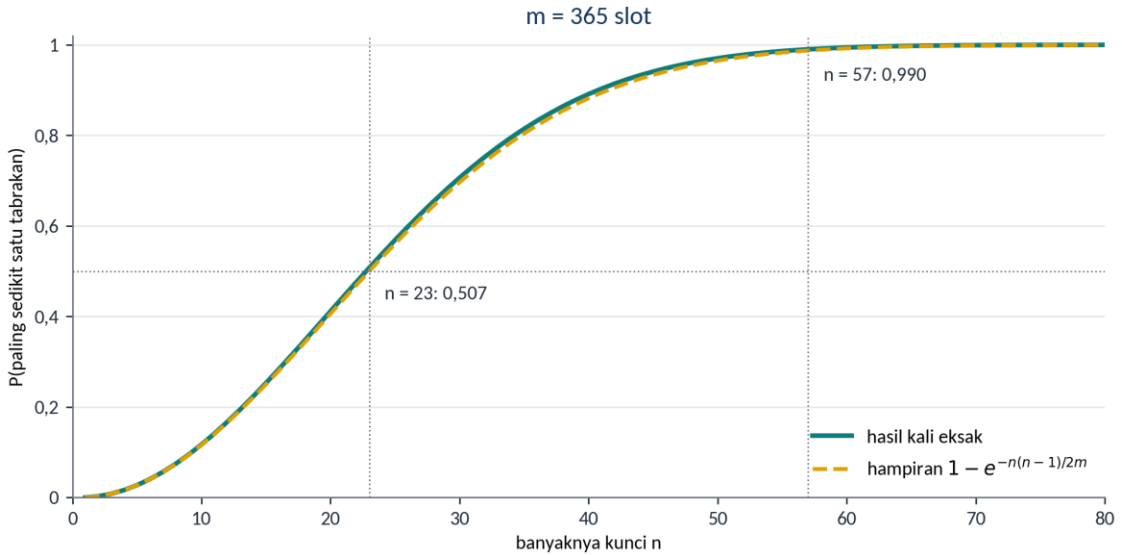
$$E[\text{banyaknya pasangan bertabrakan}] = \frac{n(n-1)}{2m}. \tag{15.10}$$

Tabel 15.2 Berapa banyak kunci acak yang membuat tabrakan mungkin terjadi, dihitung 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}

***Tabel 15.2**: bahkan 2^{64} nilai hash bertabrakan sesudah sekitar lima miliar kunci. Menurut Persamaan (15.10), 23 kunci dalam 365 slot mengharapkan 0,69 pasangan bertabrakan. Buku: Subbab 15.4.3.*

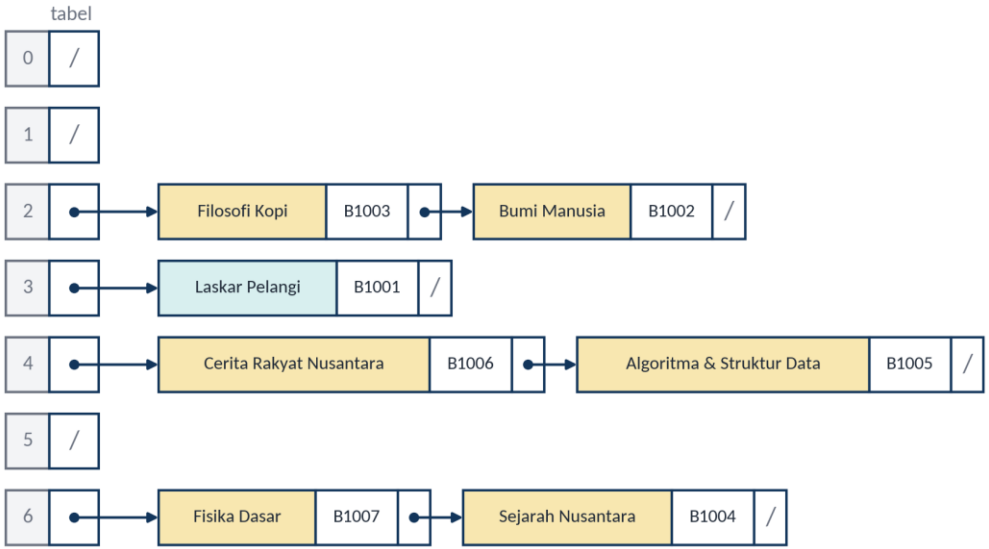
Kurva Ulang Tahun



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).

***Gambar 15.2**: tabel tidak dapat menghindari tabrakan, sebab ia akan memerlukan sekitar n^2 slot; ia merencanakannya. 1000 kunci dalam 1000 bucket menyisakan 353 kosong. Buku: Subbab 15.4.3–15.4.4 dan Gambar 15.2.*

Separate Chaining



n = 7 kunci, m = 7 bucket, h(k) = poly31(k) mod 7; rantai kuning berisi kunci yang bertabrakan

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.

***Gambar 15.3**: setiap slot memuat head sebuah single linked list dari Bab 7; bucket 2, 4, dan 6 masing-masing memuat dua judul. Buku: Subbab 15.5.1 dan Gambar 15.3.*

Cari, Sisip, Hapus dengan Rantai

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
```

***Pseudokode 15.2**: sisipkan di depan sesudah pencarian (kunci yang sudah ada diperbarui, tidak digandakan); penghapusan adalah `deleteKey` dari Subbab 7.6.5 dengan pointer pengekornya; baris 7 mengawasi faktor muat. Buku: Subbab 15.5.1.*

`put` dalam C++

Kode 15.2 `put` dari `chaining.cpp`: perbarui di tempat, atau sisipkan di depan dan lakukan *rehash* bila muatan melewati batas.

```
    // 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()));
    }
```

Kode 15.2*: judul kedelapan membuat* $8/7 > 1$*; tabel tumbuh menjadi 17 bucket, bilangan prima berikutnya sesudah 14, dengan memindahkan simpul alih-alih menyalinnya. Valgrind: ke-25 blok dilepas. Buku: Subbab 15.5.2–15.5.3.*

Seberapa Panjang Rantainya?

$$E[\text{perbandingan, gagal}] = \alpha, \qquad \text{waktu } \Theta(1+\alpha). \tag{15.11}$$

$$E[\text{perbandingan, berhasil}] = \frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{n-i}{m}\right) = 1+\frac{n-1}{2m} \approx 1+\frac{\alpha}{2}. \tag{15.12}$$

Sebuah kunci jatuh di *bucket* tertentu dengan peluang $1/m$: panjang rantai harapan adalah α.

Jaga α di bawah suatu konstanta dengan menumbuhkan m seiring n, dan setiap operasi menjadi $O(1)$ secara rata-rata.

Persamaan (15.11) menyusuri seluruh rantai; Persamaan (15.12) melewati kunci-kunci yang disisipkan sesudah x ke bucket-nya. Keduanya hanya bergantung pada α, bukan pada n. Buku: Subbab 15.5.4.

Pengalamatan Terbuka

Setiap kunci tinggal *di dalam* larik, satu per slot; tanpa pointer, tanpa simpul, satu blok yang ramah *cache*, tetapi $\alpha \leq 1$.

$$h(k,0),\ h(k,1),\ \ldots,\ h(k,m-1), \tag{15.13}$$

Penyisipan menyusuri Persamaan (15.13) sampai slot kosong; pencarian menyusuri barisan *yang sama* sampai kuncinya atau slot kosong, yang membuktikan ketiadaan.

Jejak berjalannya menyisipkan 1003, 1019, 1035, 1012, 1028, 1013, 1044, 1051 ke 16 slot: empat menginginkan slot 11, tiga menginginkan slot 4. Buku: Subbab 15.6.1 dan Persamaan (15.13).

Tiga Barisan Penyelidikan

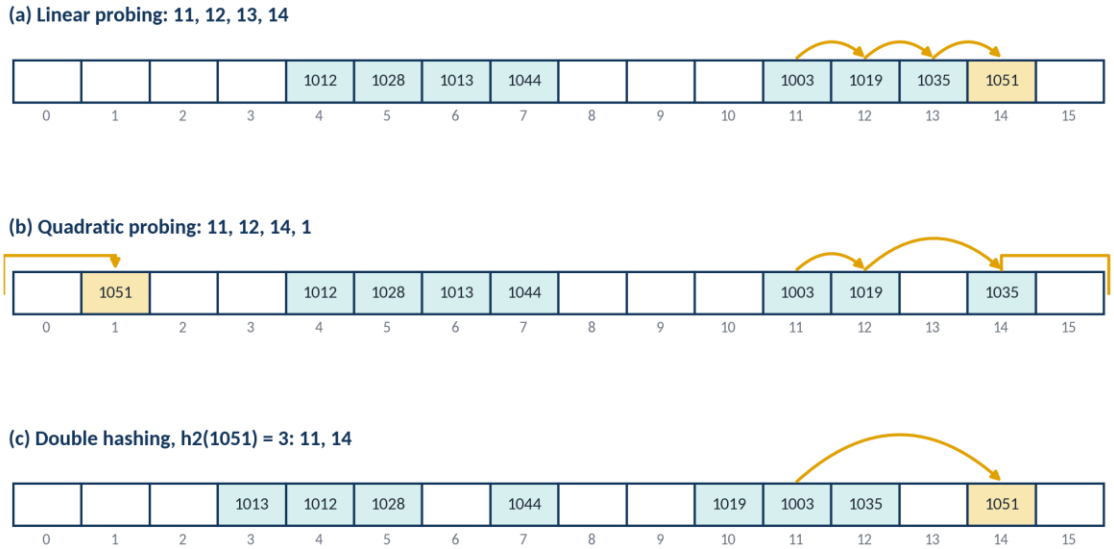
$$h(k,i) = (h_1(k) + i) \bmod m. \tag{15.14}$$

$$h(k,i) = \left(h_1(k) + \frac{i(i+1)}{2}\right) \bmod m. \tag{15.15}$$

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m. \tag{15.16}$$

Persamaan (15.14) menderita primary clustering: deretan terisi menyatu dan memanjang. Persamaan (15.15) dengan pergeseran segitiga mengunjungi setiap slot tabel berukuran pangkat dua, tetapi kunci dengan rumah yang sama menempuh jalur yang sama. Persamaan (15.16) membuat langkahnya bergantung pada kunci. Buku: Subbab 15.6.2–15.6.4.

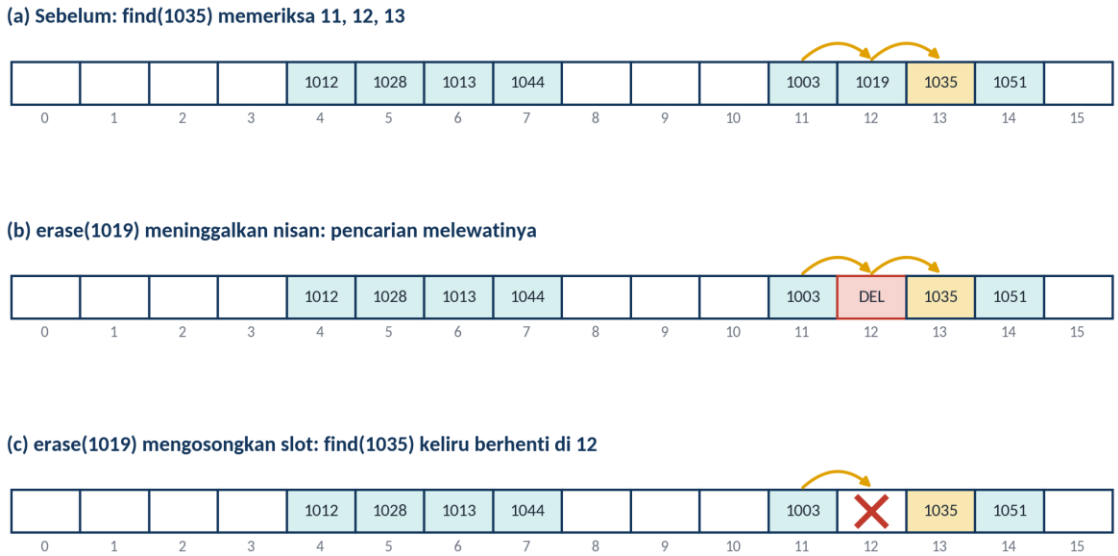
Contoh Terurai: Delapan Kunci, Tiga Cara



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.

***Gambar 15.4**: menyisipkan 1051 memerlukan 4, 4, dan 2 penyelidikan; kedelapan kunci memerlukan 19, 18, dan 15. Buku: Subbab 15.6 dan Gambar 15.4.*

Penghapusan Memerlukan Nisan



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.

***Gambar 15.5**: mengosongkan slot 12 memutus jalur penyelidikan 1035, yang lalu "tidak ditemukan". Penanda DELETED dilewati oleh pencarian dan dipakai ulang oleh penyisipan. Buku: Subbab 15.6.5 dan Gambar 15.5.*

`find` dan `erase` dalam C++

Kode 15.3 `find` dan `erase` dari `open_addressing.cpp`: EMPTY mengakhiri pencarian, DELETED tidak.

```
    // 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;
    }
```

***Kode 15.3**. Penyisipan harus terus berjalan melewati nisan sampai slot EMPTY sebelum memakai ulang nisan pertama, atau ia bisa menyimpan salinan kedua kunci itu. Buku: Subbab 15.6.5–15.6.6.*

Chaining atau Pengalamatan Terbuka?

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

***Tabel 15.3**: C++ menjanjikan bahwa referensi bertahan melewati rehash dan menyediakan antarmuka bucket, yang hanya wajar untuk chaining. Buku: Subbab 15.6.7.*

Satu-Satunya Bilangan yang Penting

$$\alpha = \frac{n}{m}, \tag{15.17}$$

$$E[\text{probe, gagal}] \leq 1 + \alpha + \alpha^2 + \cdots = \frac{1}{1-\alpha}. \tag{15.18}$$

$$E[\text{probe, berhasil}] \leq \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-i/m} \approx \frac{1}{\alpha}\int_0^{\alpha}\frac{dx}{1-x} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}. \tag{15.19}$$

Persamaan (15.17): janji $O(1)$ adalah janji bahwa α tetap terbatas, dan tabel itulah yang harus menepatinya. Di bawah uniform hashing berlaku Persamaan (15.18) dan Persamaan (15.19); double hashing mendekatinya. Buku: Subbab 15.7.1–15.7.2.

Linear Probing, dan Pengukurannya

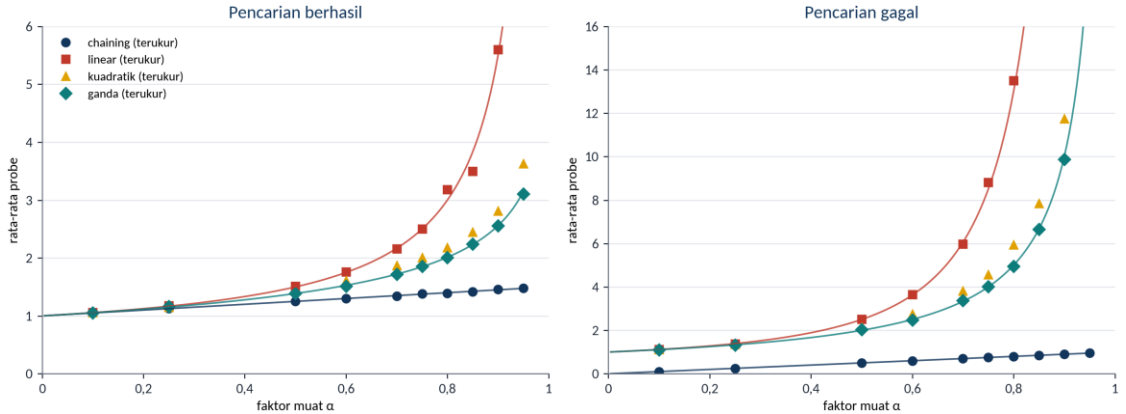
$$E[\text{probe, gagal}] \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right), \tag{15.20}$$

$$E[\text{probe, berhasil}] \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right). \tag{15.21}$$

	chaining		linear		quadratic		double	
alpha	succ	unsucc	succ	unsucc	succ	unsucc	succ	unsucc
0.50	1.25	0.50	1.51	2.52	1.43	2.15	1.39	2.03
0.75	1.38	0.76	2.50	8.81	2.01	4.56	1.86	4.01
0.90	1.46	0.90	5.60	50.07	2.81	11.74	2.56	9.87

Persamaan (15.20) dan Persamaan (15.21) adalah analisis algoritma pertama Knuth (1962): kuadrat itu adalah primary clustering. Pada $\alpha = 0{,}9$: 50,5 diramalkan, 50,07 terukur. Buku: Subbab 15.7.2–15.7.3.

Terukur lawan Teramalkan



Gambar 15.6 Rata-rata penyelidikan per pencarian dalam tabel 16.384 slot, terukur (penanda) dan teramalkan 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*.

***Gambar 15.6**: teorinya benar; linear probing yang hampir penuh tidak menentu; chaining menurun dengan anggun. Pustaka melakukan rehash pada 1,0 (C++), 0,75 (Java), 2/3 (Python). Buku: Subbab 15.7.3 dan Gambar 15.6.*

Bagaimana Panjang Rantai Tersebar

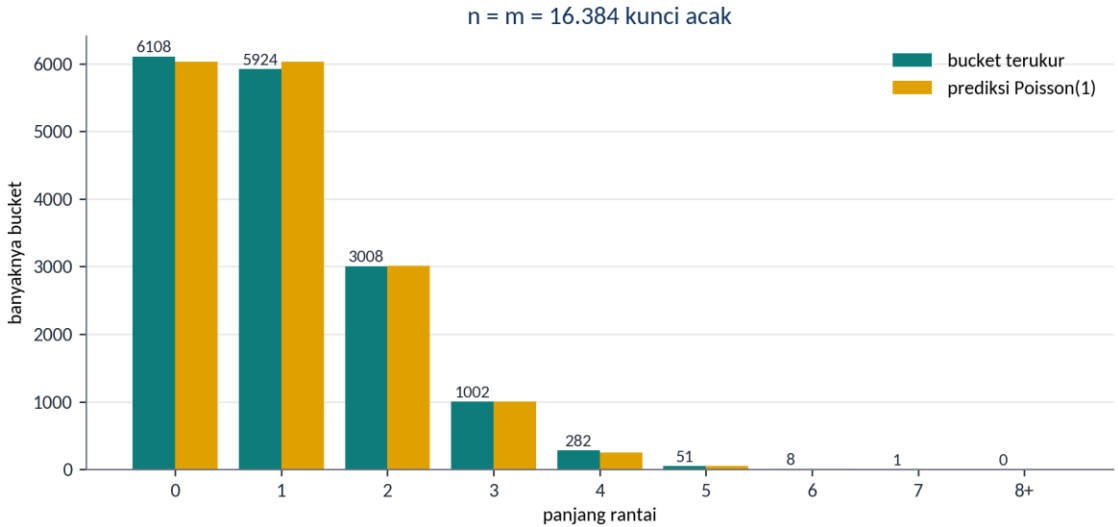
$$\Pr[\text{sebuah bucket berisi tepat } j \text{ kunci}] \approx \frac{e^{-\alpha}\alpha^{j}}{j!}. \tag{15.22}$$

Untuk $\alpha = 1$: 36,8% *bucket* kosong, 36,8% berisi satu kunci, 18,4% dua, 6,1% tiga.

Terukur pada $n = m = 16384$: 6108 kosong (6027 diramalkan), rantai terpanjang 7.

Persamaan (15.22) menjelaskan mengapa `HashMap` Java baru mengubah bin menjadi pohon pada 8 entri: dengan kode hash acak itu terjadi sekitar sekali dalam sepuluh juta. Buku: Subbab 15.7.4 dan Persamaan (15.22).

Panjang Rantai, Diplot



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.

***Gambar 15.7**: lebih dari sepertiga bucket kosong; tidak ada yang memuat lebih dari 7 kunci. Buku: Subbab 15.7.4 dan Gambar 15.7.*

Rehashing, dan Mengapa Penggandaan Itu Murah

Setiap slot bergantung pada m, jadi tabel yang lebih besar berarti menyisipkan ulang setiap kunci: $\Theta(n)$ untuk satu *rehash*.

Penggandaan membuat total pemindahan kurang dari $2n$, Persamaan (15.23):

$$\text{total kunci dipindahkan} \leq n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) < 2n. \tag{15.23}$$

Terukur: 200.000 penyisipan, 16 *rehash*, 393.226 pemindahan, 1,966 per penyisipan.

Tumbuh dengan jumlah tetap malah $\Theta(n^2)$ (Soal Latihan 15.13). Panggil `reserve(n)` atau beri Java kapasitas bila banyaknya kunci sudah diketahui. Buku: Subbab 15.7.5 dan Persamaan (15.23).

Tagihan Lengkapnya

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	$\Theta(n+m)$	$\Theta(m)$	$\Theta(n+m)$

***Tabel 15.4**: kasus terburuknya adalah semua kunci di satu slot: satu list panjang atau satu deretan panjang, $n(n-1)/2$ perbandingan untuk menyisipkan n kunci. Buku: Subbab 15.8.1.*

Hash Flooding

Penyerang yang mengetahui h mengirim kunci yang semuanya bertabrakan: Crosby dan Wallach (2003); Klink dan Wälde (2011) melambatkan hampir setiap *platform* web dengan satu permintaan rakitan.

"Aa" dan "BB" bertabrakan di basis 31; 11 blok memberi $2^{11} = 2048$ *string* 22 huruf yang saling bertabrakan.

```
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
```

2.096.128 perbandingan, tepat $2048 \cdot 2047/2$*, lawan 935 untuk kunci biasa; dengan sejuta kunci tabel itu perlu berjam-jam untuk terisi. Buku: Subbab 15.8.2.*

Pertahanannya: Hashing Teracak

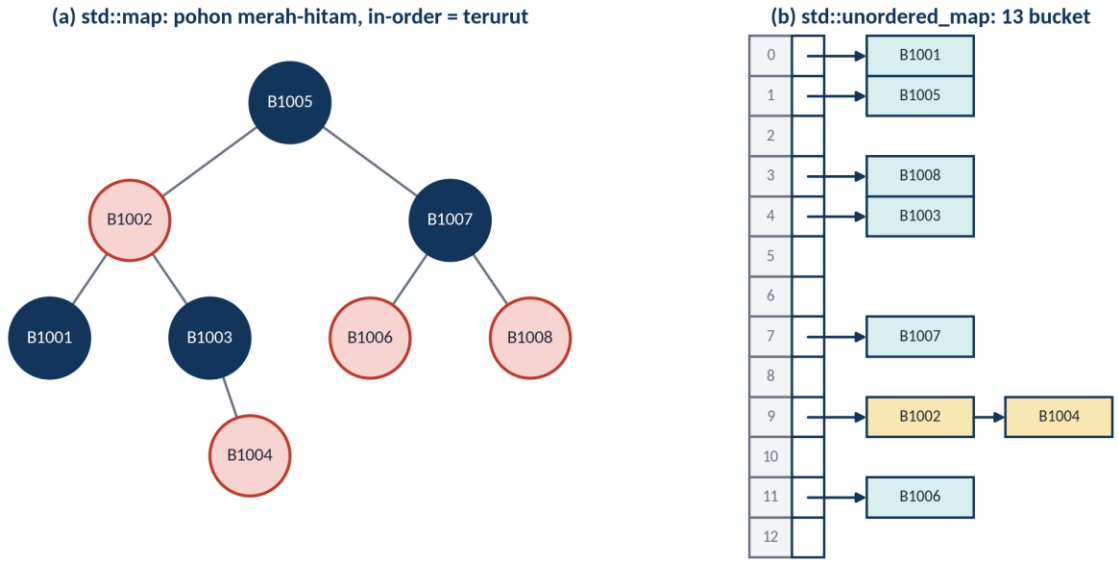
DEFINISI. Keluarga fungsi *hash* $\mathcal{H}$ bersifat *universal* bila untuk setiap pasangan kunci berbeda berlaku Persamaan (15.24), atas pemilihan h secara acak.

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}, \tag{15.24}$$

$$h_r(s) = \left(\sum_{i=0}^{L-1} s_i \; r^{L-1-i} \right) \bmod p. \tag{15.25}$$

Persamaan (15.25) dengan r rahasia modulo $2^{31} - 1$: dua string berbeda bertabrakan dengan peluang paling besar $(L-1)/p$; kunci serangan tersebar (rantai terpanjang 4). Python memakai SipHash-1-3 berkunci saat mulai; Java 8 mengubah bin yang sesak menjadi pohon. Buku: Subbab 15.8.3.

`std::map` lawan `std::unordered_map`



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.

***Gambar 15.8**: pohon mengembalikan kode-kode terurut; tabel hash dalam urutan yang bukan terurut dan bukan urutan penyisipan, dan dapat berubah sesudah rehash berikutnya. Buku: Subbab 15.9.1 dan Gambar 15.8.*

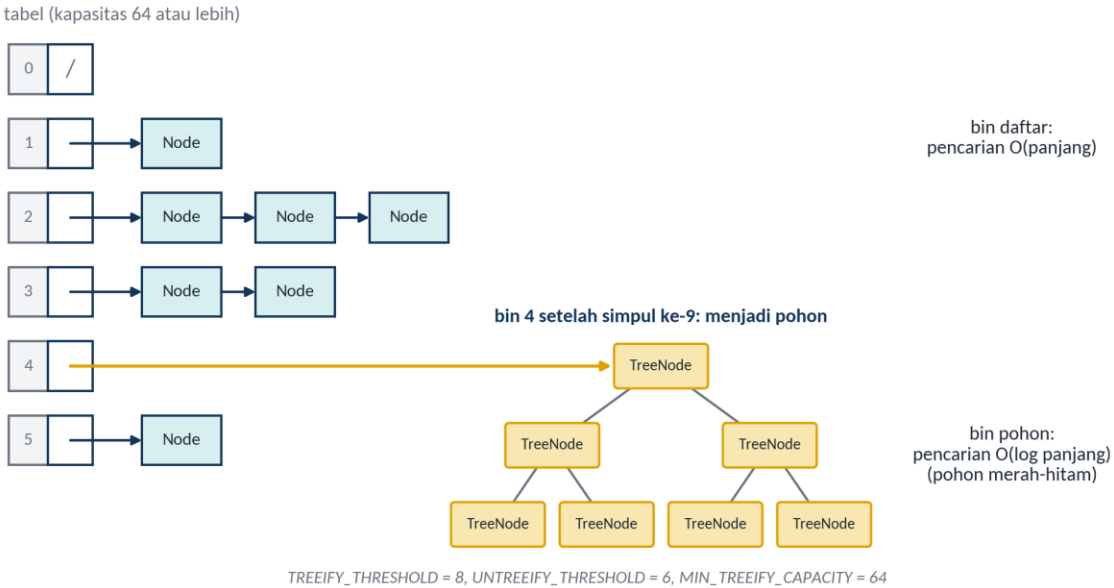
Antarmuka Sama, Mesin Berbeda

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

***Tabel 15.5**: `std::map` adalah pohon red-black, apa pun sebutan soalnya. Pada sejuta kunci acak, tabel hash menyisip sekitar 4 kali dan menemukan sekitar 25 kali lebih cepat: pohon membayar cache miss per level. Buku: Subbab 15.9.*

Di Dalam HashMap Java



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.

***Gambar 15.9**: tabel pangkat dua mulai 16, muatan 0,75; bin yang sesak menjadi pohon. Buku: Subbab 15.10.1 dan Gambar 15.9.*

Di Dalam dict Python

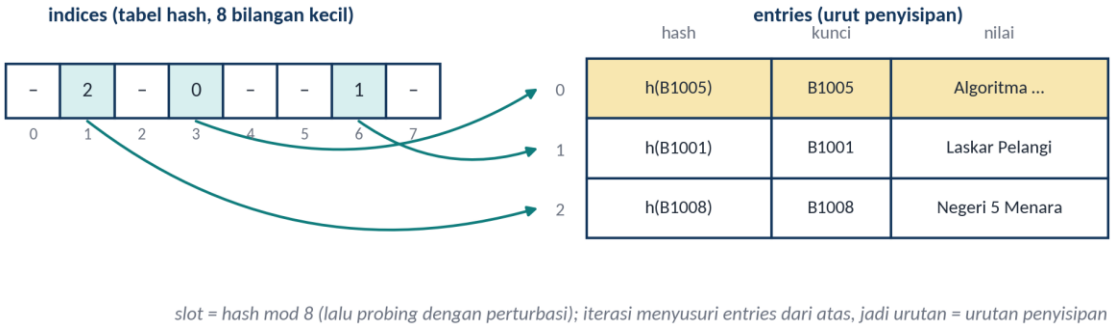
$$j \leftarrow (5j + 1 + \text{perturb}) \bmod 2^k, \qquad \text{perturb} \leftarrow \text{perturb} \gg 5, \tag{15.26}$$

Pengalamatan terbuka dalam 2^k slot; *perturb* menggeser masuk bit-bit tinggi, seperti *double hashing*, lalu $5j + 1$ mengunjungi setiap slot.

Ringkas sejak 3.6: tabel `indices` kecil menunjuk ke larik `entries` yang padat, sehingga iterasi mengikuti urutan penyisipan (dijamin sejak 3.7). Paling penuh 2/3.

Persamaan (15.26). Bilangan bulat di-hash menjadi dirinya sendiri modulo $2^{61} - 1$ (`hash(-1)` adalah -2); string memakai SipHash-1-3 dengan kunci per proses. Buku: Subbab 15.10.2 dan Persamaan (15.26).

Tata Letak Ringkas



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.)

***Gambar 15.10**: penyelidikan terjadi di larik `indices` yang kecil; entri yang besar tidak pernah berpindah karena tabrakan. `set` tidak punya larik `entries`, sehingga urutannya bukan urutan penyisipan. Buku: Subbab 15.10.2 dan Gambar 15.10.*

Tiga Pustaka Berdampingan

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`)

Tabel 15.6 *merangkum Subbab 15.9 dan 15.10 (GCC 13, JDK 25, CPython 3.14). Buku: Subbab 15.10.*

Penerapan, dan Cache LRU

Kode 15.4 Kelas `LruCache` dari `applications.cpp`: `std::list` dalam urutan kebaruan dan `unordered_map` dari kunci ke iterator *list*.

```
class LruCache {                                    // a hash map + a linked list
public:
    explicit LruCache(std::size_t capacity) : capacity_(capacity) {}

    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;
    }
```

***Kode 15.4**: sepuluh permintaan, lima hit. Program yang sama menghitung kata (`++count[w]`), mendeteksi pengembalian ganda, dan menemukan dua harga yang berjumlah Rp119.000 dalam satu lintasan. Map terbalik melayani rencana perjalanan di Bab 16. Buku: Subbab 15.11.*

Profitina dalam Praktik

Satu halaman, banyak alamat: kanonisasi URL, lalu himpunan halaman yang sudah dikunjungi menjawab “pernah dilihat?” dalam $O(1)$.

Halaman yang hampir sama: sidik jari *rolling hash* dari jendela-jendela kata, dibandingkan dengan pencarian di himpunan.

Menyaring tautan, **mencacah per pengguna**, **menyimpan jawaban yang mahal**: keanggotaan himpunan, pencacah berkunci akun, *cache* LRU.

Jenis *hash* yang lain: kata sandi disimpan dengan Argon2id, kebalikan dari *hash* tabel yang cepat.

Profitina adalah platform visibilitas AI (GEO) yang dibangun dengan FastAPI di atas PostgreSQL; kita melukiskan bentuknya, bukan implementasi produknya. Buku: Subbab 15.12.

Kesalahan yang Sering Terjadi

Sisa bagi dengan pangkat dua pada *hash* yang lemah; menjumlahkan karakter; sisa bagi negatif (`-7 % 5` adalah −2 di C++ dan Java).

Mengubah kunci selagi tersimpan; `equals` **tanpa** `hashCode` **yang sepadan.**

Mengosongkan slot saat menghapus; menghentikan penyisipan di nisan pertama; membiarkan faktor muat membesar.

Bergantung pada urutan iterasi; menyebut `std::map` **tabel *hash*;** `m[key] == 0` **untuk menguji keberadaan;** ***hash*** **publik untuk kunci dari orang asing.**

Dan jangan pernah hash tabel untuk kata sandi, atau hash kata sandi untuk tabel. Buku: Subbab 15.13.

Ringkasan: Lima Hal untuk Diingat

1. *Hashing* menghitung slot dari kuncinya dan lolos dari batas bawah perbandingan: $O(1)$ rata-rata, tanpa urutan, tanpa jaminan.

2. Fungsi yang baik: pembagian dengan m prima, perkalian Fibonacci yang mengambil bit atas, *hash* polinomial *string* dengan Horner.

3. Tabrakan datang pada sekitar $\sqrt{m}$ kunci; *chaining* berongkos sekitar $1 + \alpha/2$ dan α, pengalamatan terbuka $1/(1 - \alpha)$ dan memerlukan nisan.

4. Jaga α tetap terbatas dengan penggandaan: kurang dari dua pemindahan per penyisipan.

5. Kasus terburuknya $\Theta(n)$ dan dapat dipancing; *hash* berkunci dan *bin* yang menjadi pohon adalah pertahanannya. C++ merantai, Java merantai lalu menjadi pohon, Python menyelidiki indeks ringkas.

Buku: Subbab 15.14.

Pertemuan Berikutnya

Bab 16 — Evaluasi dan Pembahasan IV. Graf, *heap*, dan tabel *hash* dalam praktik, serta soal-soal yang memakai seluruh isi buku.

Rencana perjalanan dari setumpuk tiket kembali, dengan *map* dan *map* terbalik.

Sebelum itu: kerjakan Soal Latihan 15.1, 15.2, 15.7, dan 15.12, dalam C++ lalu dalam Java atau Python.

Penyelesaiannya, lengkap dengan program, ada di buku. Buku: Subbab 15.15 dan 15.16.

Kuliah 16 — Evaluasi dan Pembahasan IV: Graf, Heap, dan Tabel Hash dalam Praktik

Struktur Data (EF234201)

Buku: Irfan Subakti, *Struktur Data dengan C++, Java, dan Python*, Bab 16

Dosen: Irfan Subakti — yifana@gmail.com

Hari ini tidak ada struktur baru: sembilan soal yang mempekerjakan graf, heap, dan tabel hash, serta pertanyaan yang disiapkan seluruh buku, yaitu struktur mana yang harus dipilih. Buku: Bab 16, Subbab 16.1–16.12.

Cakupan Pertemuan Ini

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

Tabel 16.1: untuk setiap soal kita menggambar strukturnya, menelusurinya dengan tangan, membaca kodenya, dan menjalankan penyelesaian keliru yang menggoda. Lalu Subbab 16.8 mengukur setiap struktur buku ini pada sepuluh ribu buku yang sama. Buku: Subbab 16.1.

Titik Tolak Kita: Satu Gagasan

Bab 13: graf, *adjacency list*, BFS dengan *queue*, DFS dengan *stack*, *topological sort*, Dijkstra. **Bab 14**: *binary heap* dalam larik, *sift-up* dan *sift-down* dalam $O(\lg n)$. **Bab 15**: fungsi hash, *chaining*, pengalamatan terbuka.

> **WAWASAN KUNCI: SATU GAGASAN DI BALIK HAMPIR SETIAP SOAL DI SINI.** *Setiap struktur menjanjikan sesuatu, yaitu invarian, dan setiap operasi harus membiarkan janji itu tetap benar. Heap*: tidak ada anak yang lebih besar daripada induknya. BFS: simpul keluar dari *queue* menurut urutan jarak. Dijkstra: simpul yang diambil dari *heap* sudah memiliki jarak akhirnya. Tabel hash: kunci berada di slot yang ditunjuk hash-nya, atau lebih jauh di lintasan pencarian.

Bila sebuah program dalam bab ini keliru, hampir selalu karena ada operasi yang membiarkan janjinya rusak. Buku: Subbab 16.1.

Metode Sembilan Langkah

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?

Tabel 16.2: tujuh langkah Bab 8 ditambah dua langkah baru: Langkah 7 menghitung ongkos, Langkah 8 memutuskan apa yang dilakukan terhadap masukan yang melanggar aturan. Buku: Subbab 16.2.

Tiga Struktur, Tiga Bahasa

Tabel 16.3 Struktur-struktur Bab 13 sampai 15 dalam tiga bahasa (pilihan dari tabel buku). Di C++ kata `map` berarti pohon pencarian, bukan tabel hash.

Gagasan	C++	Java	Python
Priority queue	`std::priority_queue<T>`	`PriorityQueue<T>`	`heapq` pada `list`
Ujung mana yang keluar lebih dahulu	yang **terbesar**	yang **terkecil**	yang **terkecil**
Tabel hash (*map*)	`std::unordered_map<K, V>`	`HashMap<K, V>`	`dict`
Map terurut (pohon pencarian seimbang)	`std::map<K, V>`	`TreeMap<K, V>`	tidak ada
Queue FIFO untuk BFS	`std::queue<int>`	`ArrayDeque<Integer>`	`collections.deque`

Tabel 16.3: dua baris menimbulkan paling banyak galat. `std::priority_queue` mengeluarkan yang terbesar, `PriorityQueue` dan `heapq` yang terkecil; dan `std::map` adalah pohon pencarian, tabel hash-nya `std::unordered_map`. Buku: Subbab 16.2.

Soal 16.1 — Melengkapi Kelas Max-Heap

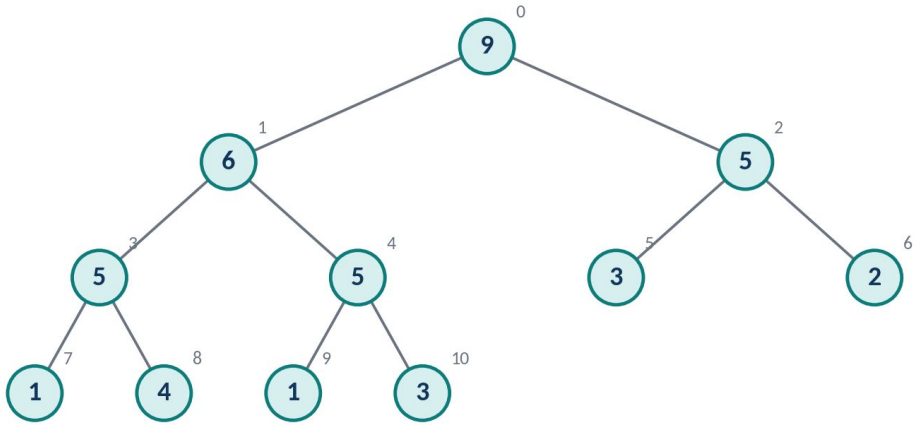
SOAL. Add to the class `MaxHeap` (a `vector<int> heap` with `heapifyDown` and `heapifyUp`): (1) `extractMax()`; (2) `increaseKey(index, newValue)`; (3) `getMin()`; (4) `buildMaxHeap(arr)`; (5) `isMaxHeap(arr)`; and `clear()`. Insert 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, then test each.

$$parent(i) = \left\lfloor \frac{i-1}{2} \right\rfloor, left(i) = 2i+1, right(i) = 2i+2. \tag{16.1}$$

DEFINISI. *Invarian max-heap*: `heap[parent(i)] >= heap[i]` untuk setiap $1 \le i < n$. Jadi `heap[0]` adalah elemen terbesar.

Persamaan (16.1): kerabat indeks i ditemukan dengan aritmetika. Setiap operasi baru adalah tugas satu kalimat yang dibaca dari invarian. Buku: Subbab 16.3.1–16.3.2.

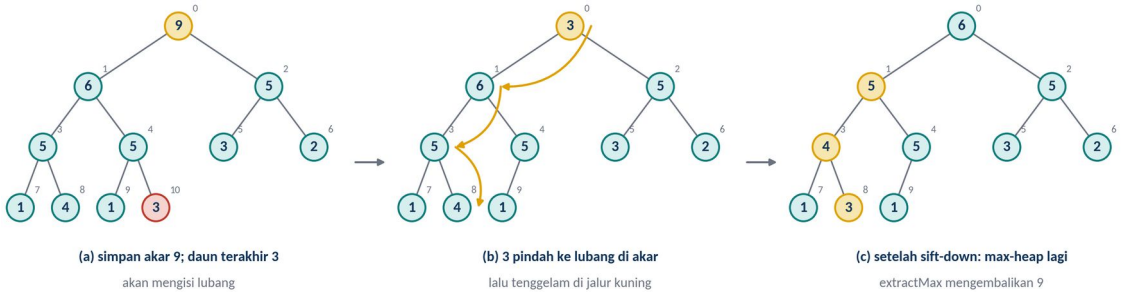
Heap Setelah Sebelas Penyisipan



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.

Gambar 16.1: setiap induk paling kecil sama dengan anak-anaknya, tetapi lariknya tidak terurut, dan memang tidak perlu. Buku: Subbab 16.3.3 dan Gambar 16.1.

Contoh Terurai: `extractMax` dalam Tiga Langkah



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.

Gambar 16.2: simpan 9, pindahkan daun terakhir 3 ke akar, tenggelamkan melewati 6, 5, dan 4: `[6, 5, 5, 4, 5, 3, 2, 1, 3, 1]`. *Buku: Subbab 16.3.3 dan Gambar 16.2.*

`extractMax` dan `increaseKey` dalam C++

Kode 16.1 `extractMax` dan `increaseKey` dari `p1601_max_heap.cpp`: simpan jawabannya lebih dahulu; kunci yang lebih besar hanya dapat naik.

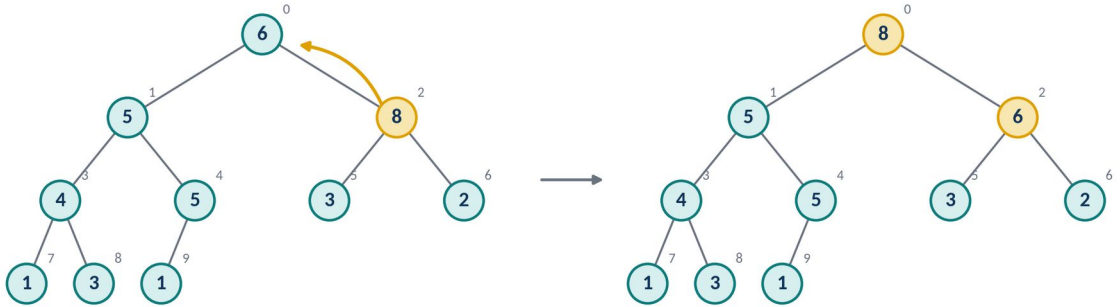
```cpp
int extractMax() {
    if (heap.empty()) throw underflow_error("extractMax on an empty heap");
    int maxVal = heap[0];            // SAVE the answer first
    heap[0] = heap.back();           // the last leaf fills the hole at the root
    heap.pop_back();
    if (!heap.empty()) heapifyDown(0);
    return maxVal;
}

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);
}
```

$$h = \lfloor \lg n \rfloor . \tag{16.2}$$

Kode 16.1: satu lintasan, $O(\lg n)$ menurut Persamaan (16.2); penurunan ditolak. Buku: Subbab 16.3.3–16.3.4.

`increaseKey(2, 8)`: Satu Pertukaran ke Atas



(a) indeks 2 dinaikkan dari 5 ke 8: lebih besar daripada induknya 6

(b) satu pertukaran dengan induk 8 di akar; bagian lain tidak tersentuh

Gambar 16.3 `increaseKey(2, 8)`: kunci yang dinaikkan hanya dapat menjadi terlalu besar bagi induknya, sehingga ia naik; satu pertukaran memulihkan invarian.

Gambar 16.3: 8 tidak dapat merugikan anak-anaknya (paling besar 5); ia hanya dapat terlalu besar bagi induknya, 6. Hasilnya `[8, 5, 6, 4, 5, 3, 2, 1, 3, 1]`*. Buku: Subbab 16.3.3 dan Gambar 16.3.*

`getMin`: Hanya Daun-Daunnya

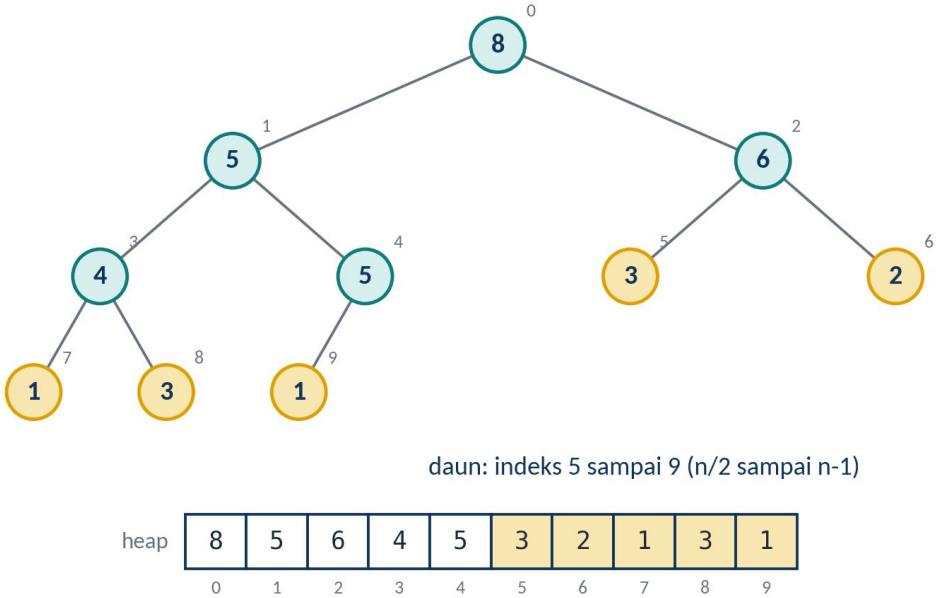
$$\text{daun} = \{\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \ldots, n-1\}, \text{sekitar } n/2 \text{ buah}. \tag{16.3}$$

Kode 16.2 `getMin` dari `p1601_max_heap.cpp`: pindai daun-daunnya saja.

```
    // 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;
    }
```

Persamaan (16.3): simpul yang punya anak bukan minimum, jadi Kode 16.2 memeriksa $\lceil n/2 \rceil$ elemen: separuh pemindaian, tetap $\Theta(n)$. Buku: Subbab 16.3.3.

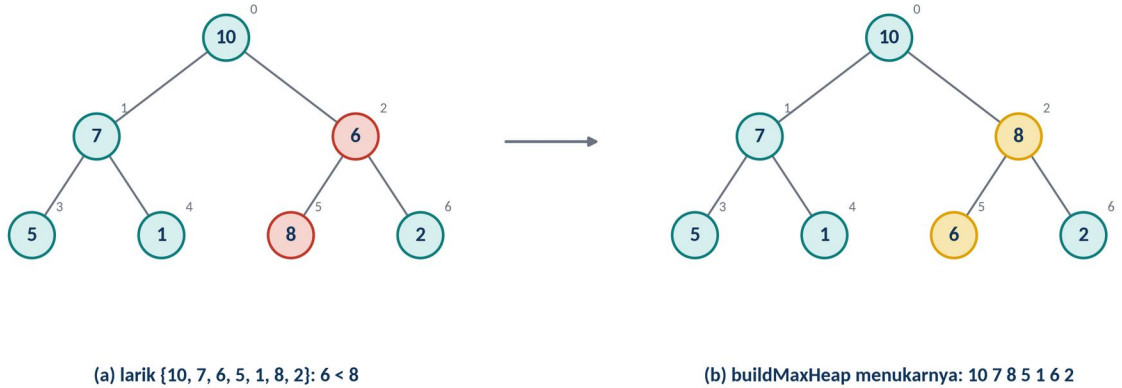
Minimumnya Adalah Daun



Gambar 16.4 Minimum sebuah *max-heap* selalu berupa daun, dan daun-daun itu adalah separuh akhir larik: indeks $\lfloor n/2 \rfloor$ sampai $n-1$.

Gambar 16.4: untuk $n=10$ daunnya adalah indeks 5 sampai 9, berisi 3, 2, 1, 3, 1; minimumnya 1. Buku: Subbab 16.3.3 dan Gambar 16.4.

`buildMaxHeap`, dari Bawah ke Atas



Gambar 16.5 `buildMaxHeap` pada {10, 7, 6, 5, 1, 8, 2}: hanya indeks 2 yang melanggar invarian, dan satu pertukaran memperbaikinya.

Gambar 16.5: `heapifyDown` pada indeks 2, 1, 0; hanya indeks 2 yang perlu ditukar: `[10, 7, 8, 5, 1, 6, 2]`. Buku: Subbab 16.3.3 dan Gambar 16.5.

Mengapa Membangun Itu Linear, dan Pemeriksaannya

$$\sum_{j=0}^{\lfloor \lg n \rfloor} \frac{n}{2^{j+1}}\, j = \frac{n}{2} \sum_{j \geq 0} \frac{j}{2^{j}} \leq \frac{n}{2} \cdot 2 = n. \tag{16.4}$$

Kode 16.3 `buildMaxHeap` dan `isMaxHeap` dari `p1601_max_heap.cpp`.

```
    // 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;
    }
```

Persamaan (16.4): sebagian besar elemen berada di dekat dasar, tempat tenggelam itu murah. Kode 16.3: `i-- > 0` menghitung mundur tanpa pernah menghasilkan -1*; `isMaxHeap` menelusuri anak-anak mulai indeks 1. Buku: Subbab 16.3.3–16.3.4.*

Keliru: Elemen Terakhir, dan `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
```

Pada *heap* milik *driver*, `getMinWrong` menjawab 1 karena kebetulan; pada `[5, 3, 4]` ia menjawab 4, padahal minimumnya 3.

Untuk satu elemen, `arr.size() - 2` bukan -1 melainkan sekitar $1{,}8 \times 10^{19}$: AddressSanitizer melaporkan *heap-buffer-overflow* di `isMaxHeapWrong`.

Elemen terakhir memang daun, tetapi hanya satu dari sekitar $n/2$ daun. Telusuri anak-anak, `i = 1; i < arr.size()`, yang batasnya tidak dapat underflow. Buku: Subbab 16.3.6.

Soal 16.2 — Menyusun Tiket Menurut Urutan Perjalanan

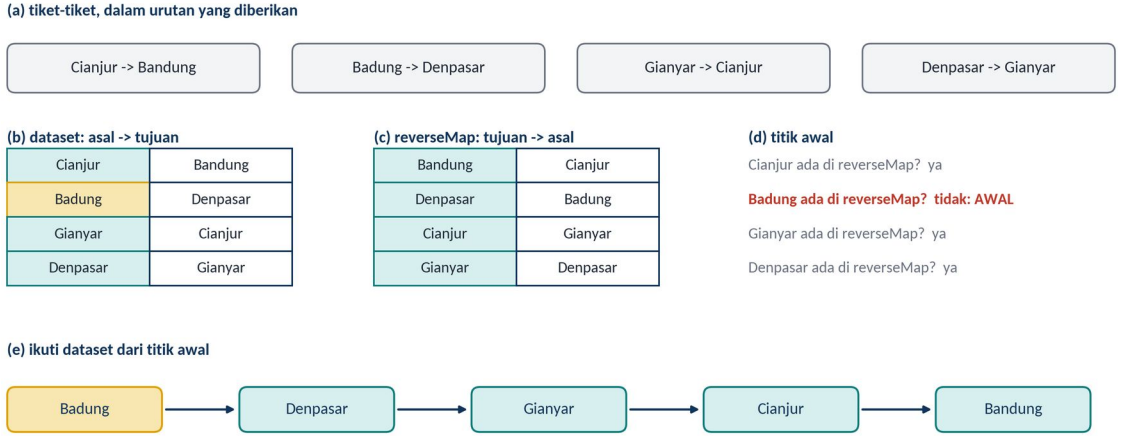
> **SOAL.** Given a list of tickets, find the itinerary in travel order, using hashing: a hash map `dataset` (*from* → *to*), a reverse map `reverseMap` (*to* → *from*); the key of `dataset` not found in `reverseMap` is the starting point. Input: Cianjur → Bandung, Badung → Denpasar, Gianyar → Cianjur, Denpasar → Gianyar.

Kota asal tiket pertama **tidak pernah menjadi kota tujuan**; setiap tiket lain berangkat dari kota tempat tiket sebelumnya tiba.

Setumpuk tiket *membentuk satu rantai* bila semua kota asalnya berbeda, tepat satu bukan kota tujuan, dan perjalanan darinya memakai setiap tiket.

Badung (Bali) dan Bandung (Jawa Barat) adalah dua tempat berbeda; selisih satu huruf itu termasuk yang membuat contohnya menjadi uji yang baik. Buku: Subbab 16.4.1–16.4.2.

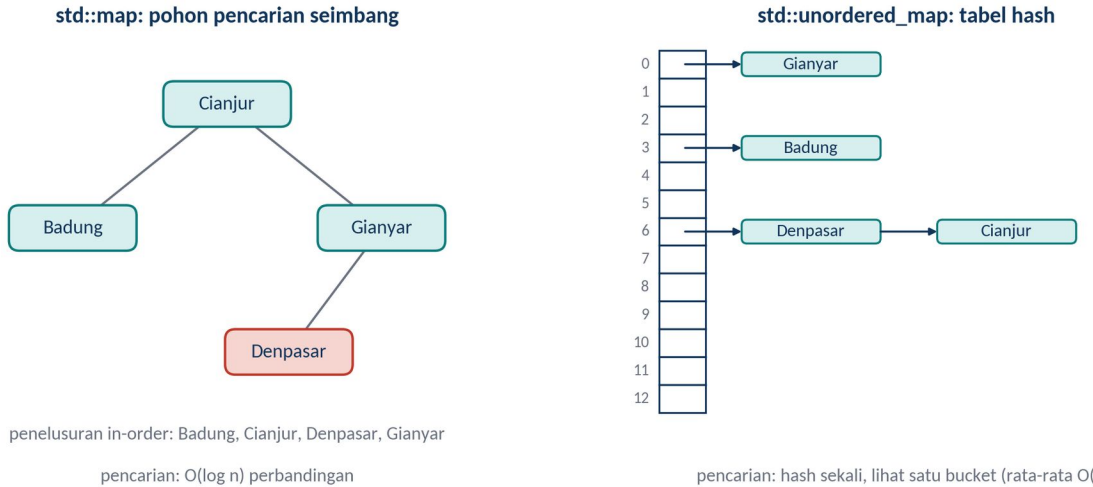
Contoh Terurai: Tiket dengan Tangan



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).

Gambar 16.6: Badung satu-satunya kota asal yang tidak pernah dituju; empat pencarian mencetak perjalanannya. Rata-rata $O(n)$, dibandingkan $O(n^2)$ bila tumpukan dicari setiap kali. Buku: Subbab 16.4.3 dan Gambar 16.6.

Map Mana yang *Hash Map*?



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.

Gambar 16.7: `std::map` mengunjungi Badung, Cianjur, Denpasar, Gianyar secara terurut dan memerlukan 4 perbandingan untuk menemukan Gianyar; `std::unordered_map` menghitung hash sekali dan melihat satu bucket. Buku: Subbab 16.4.4 dan Gambar 16.7.

`findItinerary` dalam C++

Kode 16.4 `findItinerary` dari `p1602_itinerary.cpp`: dua *hash map*, titik awal, dan perjalanannya.

```
    for (const Ticket& t : tickets) {
        if (!dataset.emplace(t.from, t.to).second)
            return "two tickets leave " + t.from;
        reverseMap[t.to] = t.from;

    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)";

    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";
```

Kode 16.4: titik awal dicari menurut urutan masukan, tidak pernah menurut urutan tabel yang tidak ditentukan; perjalanan berhenti setelah n tiket. Buku: Subbab 16.4.5.

Keliru: Mulai dari Tiket Pertama

```
  string city = tickets[0].first;      // WRONG: "the first ticket is the first leg"
```

```
Cianjur -> Bandung
```

Tiket pertama di tumpukan adalah potongan *terakhir*; dari Bandung tidak ada tiket, dan tiga perempat perjalanan hilang tanpa sepatah kata.

Kekeliruan kedua: pencacah koma dideklarasikan `int a;` dan tidak pernah diberi 0; Valgrind melaporkan "Conditional jump or move depends on uninitialised value(s)". Buku: Subbab 16.4.7.

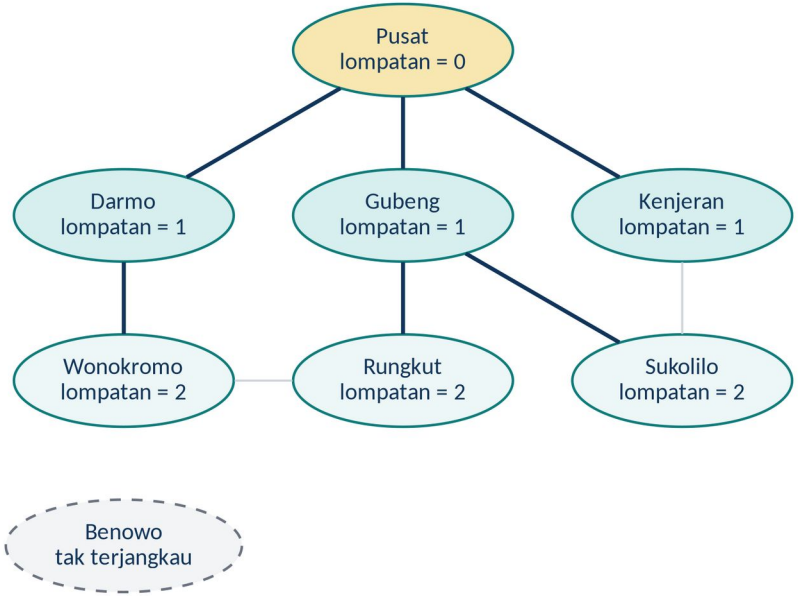
Soal 16.3 — Lompatan Paling Sedikit Antarcabang

SOAL. Jalan: Pusat–Gubeng, Pusat–Darmo, Pusat–Kenjeran, Gubeng–Sukolilo, Gubeng–Rungkut, Darmo–Wonokromo, Wonokromo–Rungkut, Kenjeran–Sukolilo; Benowo belum punya jalan. Setiap jalan satu lompatan. Tentukan lompatan paling sedikit dari Pusat ke setiap cabang, dan cetak tiga rute.

DEFINISI. *Invarian BFS*: *queue* memuat simpul menurut `dist` yang tidak menurun, yang pertama dan terakhir berselisih paling banyak 1. Jadi ketika v ditemukan dari u, `dist[v] = dist[u] + 1` sudah final.

Tetapkan `dist[v]` saat v pertama kali terlihat, bukan saat ia keluar dari queue; -1 berarti belum terlihat, jadi satu larik melayani jarak sekaligus tanda kunjungan. Buku: Subbab 16.5.1.

Cincin-Cincin Berjarak Sama



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.

Gambar 16.8: sisi gelap membentuk pohon BFS; rute ke Rungkut, dibaca mundur lewat induk-induknya, adalah Pusat → Gubeng → Rungkut. Buku: Subbab 16.5.1 dan Gambar 16.8.

`bfs` dalam C++

Kode 16.5 `bfs` dari `p1603_bfs_hops.cpp`: tandai sebelum masuk *queue*; penglihatan pertama sudah final.

```
    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);
            }
        }
    }
```

$$T_{\text{BFS}} = O(|V| + |E|). \tag{16.5}$$

Kode 16.5 berongkos Persamaan (16.5). Python harus memakai `collections.deque`, jangan `list.pop(0)`. Buku: Subbab 16.5.1.

Keliru: Kedalaman DFS sebagai Jarak

```
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);
}
```

```
Pusat 0
Gubeng 1
Darmo 4
Kenjeran 3
```

Darmo hanya satu jalan dari Pusat, tetapi DFS menyelam lewat Gubeng, Sukolilo, Kenjeran, Rungkut, Wonokromo lebih dahulu dan mencapainya paling akhir.

DFS menjawab “dapatkah saya sampai ke sana?”; hanya urutan cincin demi cincin milik queue yang menjawab “berapa lompatan paling sedikit?”. Buku: Subbab 16.5.1.

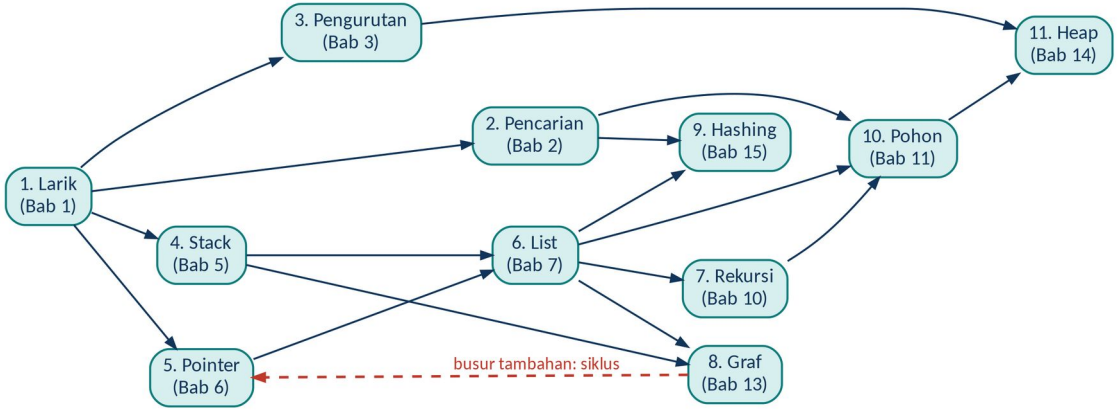
Soal 16.4 — Urutan Membaca Topik-Topik

SOAL. Kesebelas topik buku ini saling bergantung (larik sebelum pencarian, pengurutan, *stack*, dan pointer; …). Cetak urutan yang menempatkan setiap topik sesudah semua yang diperlukannya. Lalu tambahkan “graf sebelum pointer” dan laporkan apa yang terjadi.

DEFINISI. *Invarian Kahn*: setiap topik di urutan memiliki semua prasyaratnya lebih awal, dan `indeg[v]` menghitung prasyarat v yang belum ditempatkan. Bila *queue* kosong sementara masih ada topik yang tertinggal, ada siklus berarah.

Setiap “baca u sebelum v” adalah busur; urutan yang membuat setiap busur mengarah ke depan adalah urutan topologis, yang ada tepat untuk DAG. Buku: Subbab 16.5.2.

Sebelas Topik, Enam Belas Busur



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.

Gambar 16.9: Graphs dan Hashing mendahului Trees: sah, karena tidak ada busur yang melarangnya. Dengan busur merah, hanya Arrays, Searching, Sorting, Stacks yang tertempatkan. Buku: Subbab 16.5.2 dan Gambar 16.9.

`kahn` dalam C++

Kode 16.6 `kahn` dari `p1604_reading_order.cpp`: derajat masuk, *queue* berisi topik yang siap, dan `--indeg[v] == 0`.

```cpp
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;
}
```

Kode 16.6 berongkos $O(|V|+|E|)$, seperti BFS. Fungsi ini mengembalikan urutan; pemanggilnya membandingkan `order.size()` dengan n. Buku: Subbab 16.5.2.

Keliru: Tanpa Hitungan Setelah Gelung

```
Reading order: Arrays Searching Sorting Stacks
```

Dengan siklus itu, program mencetak empat topik tanpa peringatan; tujuh lainnya lenyap diam-diam.

Pada setiap masukan tanpa siklus program itu sempurna, sehingga kelalaian ini lolos dari pengujian dengan contohnya.

Langkah 5 metode, yang mencantumkan “siklus” di antara kasus tepi sebuah graf, adalah yang menangkapnya. Buku: Subbab 16.5.2.

Soal 16.5 — Rute Tercepat untuk Mobil Buku

SOAL. Jalan-jalan kini bermenit: 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. Tentukan waktu tercepat dengan *priority queue*.

DEFINISI. *Invarian Dijkstra*: setiap cabang yang selesai memiliki waktu tercepatnya; cabang yang diambil dari *heap* dengan waktu sementara terkecil boleh dinyatakan selesai, karena panjang jalan tidak pernah negatif.

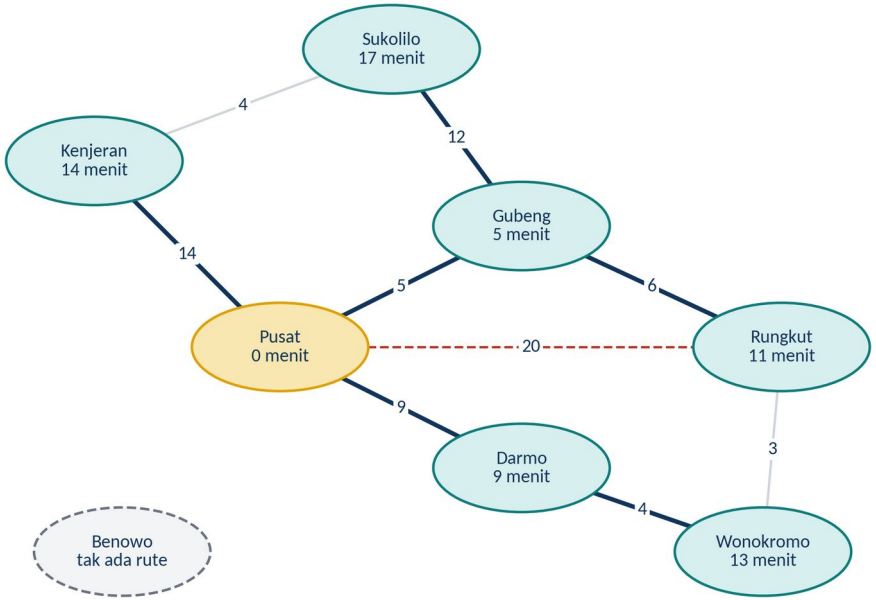
Lazy deletion: bila `dist[v]` *turun, dorong entri baru; entri untuk cabang yang sudah selesai sudah basi dan dilewati. Buku: Subbab 16.5.3.*

Contoh Terurai: Jejaknya

```
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
```

Gubeng pada 5 selesai: tidak ada rute lewat Darmo (9) atau Kenjeran (14) yang dapat mengalahkannya. Dari Gubeng, Rungkut turun dari 20 ke 11; entri lamanya tetap ada dan dilewati di akhir. Buku: Subbab 16.5.3.

Pohon Lintasan Terpendek



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).

Gambar 16.10: jalan langsung (20) kalah oleh jalan memutar lewat Gubeng (11). Buku: Subbab 16.5.3 dan Gambar 16.10.

`dijkstra` dalam C++

Kode 16.7 Inti `dijkstra` dari `p1605_dijkstra.cpp` (pencetakan jejak dihilangkan): *min-heap* berisi pasangan (waktu, cabang) dengan *lazy deletion*.

```
    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]) {

                dist[a.to] = d + a.minutes;
                parent[a.to] = u;
                heap.push({dist[a.to], a.to});
```

$$T_{\text{Dijkstra}} = O((|V|+|E|)\lg|V|). \tag{16.6}$$

Kode 16.7, ongkos Persamaan (16.6): `greater<Entry>` mengubah max-heap C++ menjadi min-heap (Tabel 16.3); `if (done[u])` adalah uji entri basi. Buku: Subbab 16.5.3.

Keliru: Final pada Penglihatan Pertama

```
  if (!seen[v]) { seen[v] = true; dist[v] = d + w; heap.push({dist[v], v}); }   // WRONG
```

```
Pusat 0
Gubeng 5
Darmo 9
Kenjeran 14
Rungkut 20
```

Rungkut dikunci pada 20 sebelum Gubeng, yang menawarkan 11, sempat diambil.

Pada BFS penglihatan pertama sudah final karena setiap sisi bernilai 1; dengan panjang jalan, waktu baru final ketika cabangnya keluar dari heap sebagai yang terkecil. Buku: Subbab 16.5.3.

Soal 16.6 — k Terbesar dengan Heap Berukuran k

SOAL. (a) Delapan buku dipinjam 12, 45, 7, 30, 18, 3, 26, dan 40 kali. Tentukan tiga yang paling sering dipinjam dengan *heap* yang tidak pernah memuat lebih dari tiga. (b) Tentukan lima buku dengan *rating* terbanyak dari katalog sepuluh ribu buku, dan periksa terhadap pengurutan.

DEFINISI. *Invarian top-k*: setelah i nilai, *heap* memuat $\min(i, k)$ nilai terbesar di antaranya, dan akarnya adalah nilai terkecil yang dimuatnya.

*Tanyakan pada setiap nilai: apakah ia lebih baik daripada yang terburuk yang saya simpan? Yang terburuk harus ditemukan seketika, jadi heap-nya **min-heap**. Buku: Subbab 16.6.1.*

Contoh Terurai: Tiga yang Paling Sering Dipinjam



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.

Gambar 16.11: B1006 (3) tidak mengalahkan akar 18 dan hanya berongkos satu perbandingan. Jawabannya B1002 (45), B1008 (40), B1004 (30). Buku: Subbab 16.6.1 dan Gambar 16.11.

`offer` dalam C++

Kode 16.8 `TopK::offer` dari `p1606_top_k.cpp`: *min-heap* berukuran k di dalam `vector`, dengan `push_heap` dan `pop_heap`.

```
    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;
    }
```

$$T_{\text{top-}k} = O(n \lg k), \text{memori tambahan } O(k). \tag{16.7}$$

Kode 16.8, ongkos Persamaan (16.7): `x <= heap.front()` *menolak dalam* $O(1)$*; nilai yang lebih besar menggantikan akar dalam* $O(\lg k)$*.* `heapq.heapreplace` *Python melakukan hal yang sama dalam satu sift-down. Buku: Subbab 16.6.1.*

Berapa Kali Penggantian?

$$\sum_{i=k+1}^{n} \frac{k}{i} \approx k \ln \frac{n}{k}, \tag{16.8}$$

```
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)
Same five as sorting all 10000 books: yes
```

Persamaan (16.8): sekitar $5 \ln 2000 \approx 38$ *pada urutan acak; urutan terbaik memberi 0, urutan terburuk memberi setiap satu dari 9.995 buku lainnya. Buku: Subbab 16.6.1.*

Keliru: *Max-Heap*

```
priority_queue<int> heap;            // WRONG: a MAX-heap; top() is the largest
heap.push(c); if (heap.size() > 3) heap.pop();
```

```
Top 3: 12 7 3
```

Setiap `pop` membuang nilai terbesar sejauh ini, tepat yang ingin kita simpan: hasilnya tiga yang *terkecil*.

Kode yang sama benar dengan `PriorityQueue` Java atau `heapq` Python, yang merupakan min-heap: baris berbahaya pertama Tabel 16.3. Buku: Subbab 16.6.1.

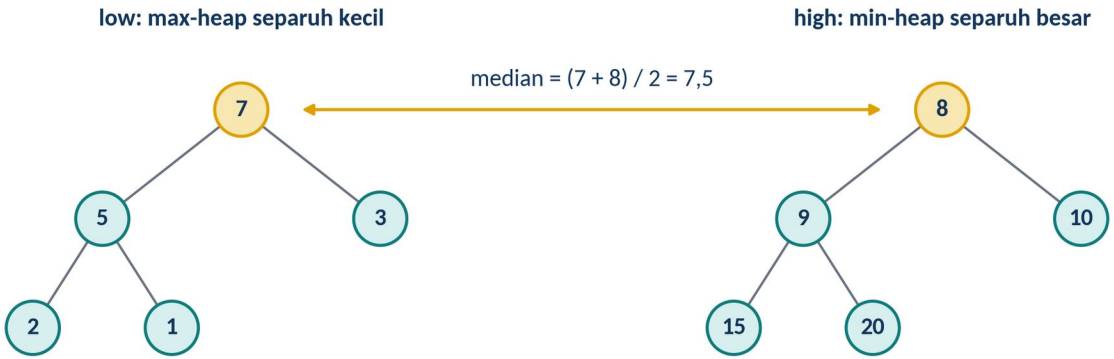
Soal 16.7 — Median Sejauh Ini, dengan Dua Heap

SOAL. Pengunjung menunggu 5, 15, 1, 3, 8, 7, 9, 10, 20, dan 2 menit. Setelah setiap pengunjung, cetak median sejauh ini. Setiap pembaruan harus berongkos $O(\lg n)$, bukan pengurutan.

DEFINISI. *Invarian dua heap*: (i) setiap nilai di *max-heap* `low` paling besar sama dengan setiap nilai di *min-heap* `high`; (ii) `low.size()` sama dengan `high.size()` atau `high.size() + 1`. Mediannya `low.top()`, atau rata-rata kedua puncak.

Tempatkan nilai dengan membandingkannya dengan `low.top()`; bila ukurannya melanggar (ii), pindahkan satu puncak ke seberang. Memindahkan puncak menjaga (i). Buku: Subbab 16.6.2.

Kedua Heap Setelah Sepuluh Nilai



Gambar 16.12 Kedua *heap* setelah kesepuluh waktu tunggu. Akar-akarnya adalah dua nilai tengah daftar terurut, sehingga median dibaca dalam waktu konstan.

Gambar 16.12: akar 7 dan 8 adalah tengah dari 1, 2, 3, 5, 7, 8, 9, 10, 15, 20; mediannya 7,5, dibaca dalam $O(1)$*. Buku: Subbab 16.6.2 dan Gambar 16.12.*

`RunningMedian` dalam C++

Kode 16.9 `add` dan `median` dari `p1607_running_median.cpp`: *max-heap* untuk separuh bawah, *min-heap* untuk separuh atas.

```
    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;
    }
```

Kode 16.9: di sini max-heap bawaan C++ tepat untuk `low`; Python menyimpan `low` dalam bentuk negatif. `2.0` membuat rata-ratanya 7,5, bukan 7. Buku: Subbab 16.6.2.

Keliru: Tanpa Penyeimbangan

```
add 5  median 5
add 15  median 10
add 1  median 5
add 3  median 10
add 8  median 5
```

Setelah 5, 15, 1, 3 kedua separuh adalah {5, 3, 1} dan {15}: bagian (i) berlaku, bagian (ii) tidak, dan median keempat 10, bukan 4.

Median adalah pernyataan tentang posisi; hanya syarat ukuran yang mengikat akar-akar ke posisi tengah. Buku: Subbab 16.6.2.

Soal 16.8 — *Separate Chaining* dengan Tangan

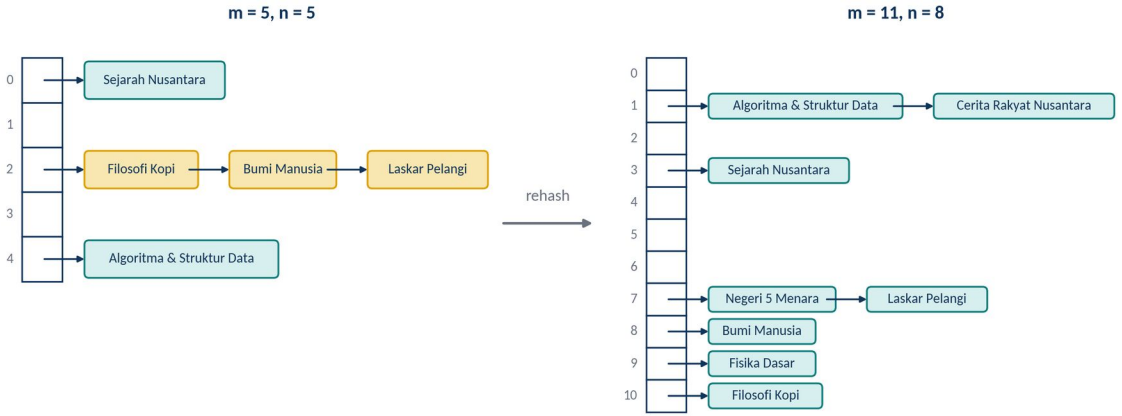
SOAL. Simpan delapan buku, dengan judul sebagai kunci, di tabel hash berantai yang dimulai dengan 5 *bucket* dan tumbuh menjadi $2m+1$ bila $n>m$. Hash: $h=(h\cdot 31+c)\bmod m$. Cari dua judul yang ada dan satu yang tidak; hapus satu judul dua kali.

$$\alpha = \frac{n}{m}, \tag{16.9}$$

$$T_{\text{rantai}} = \Theta(1+\alpha). \tag{16.10}$$

Persamaan (16.9) adalah panjang rantai rata-rata; menurut Persamaan (16.10), menjaga $\alpha \le 1$ membuat setiap operasi rata-rata $O(1)$. Invarian chaining: kunci k berada di rantai $h(k)$ tepat sekali. Buku: Subbab 16.7.1.

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.

Gambar 16.13: dengan 5 bucket tiga judul berbagi bucket 2; judul keenam membuat $6 > 5$*, dan setiap simpul ditautkan ulang ke 11 bucket. Buku: Subbab 16.7.1 dan Gambar 16.13.*

`hashOf` dan `remove` dalam C++

Kode 16.10 `hashOf` dan `remove` dari `p1608_chaining.cpp`: aritmetika *unsigned* yang direduksi di setiap langkah, dan pointer ke tautan.

```
    // 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;
    }

    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;
    }
```

Kode 16.10: `Node** link` *mula-mula menunjuk head bucket, lalu setiap* `next`*;* `*link = dead->next` *melepas simpul tanpa kasus khusus untuk simpul pertama. Valgrind: tidak ada kebocoran. Buku: Subbab 16.7.1.*

Keliru: Hash dalam `int` Bertanda

```
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
```

```
Laskar Pelangi -> bucket -1
Segmentation fault
```

UBSan: *signed integer overflow: 73191252 * 31 cannot be represented in type 'int'*; sisa bagi yang negatif memberi indeks -1.

Pakai aritmetika unsigned dan reduksi di setiap langkah; di Java, `Math.floorMod(h, m)`. Python tidak pernah meluap, tetapi nilainya tumbuh sampai ratusan digit. Buku: Subbab 16.7.1.

Soal 16.9 — Pengalamatan Terbuka, Menghapus dengan Aman

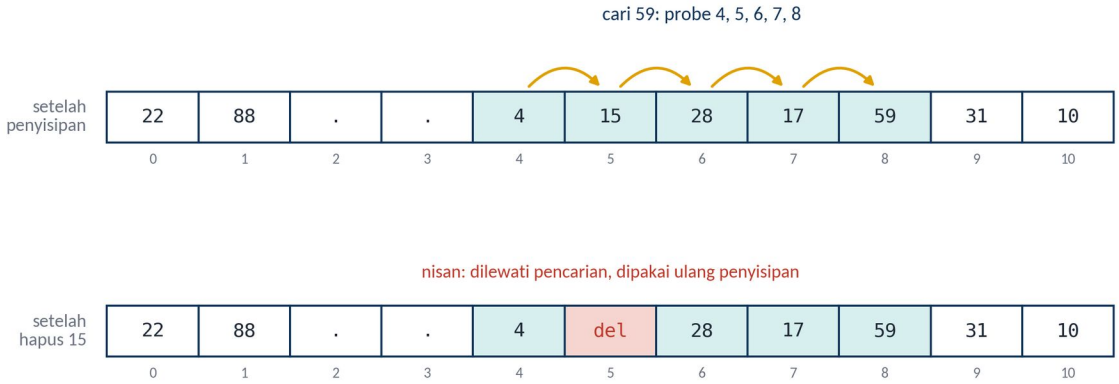
SOAL. Sisipkan 10, 22, 31, 4, 15, 28, 17, 88, 59 ke 11 slot dengan $h(k) = k \bmod 11$ dan *linear probing*. Kemudian hapus 15, cari 59 dan 26, lalu sisipkan 37.

DEFINISI. *Invarian probing*: bila k berada di slot j, setiap slot dari $h(k)$ sampai j terisi atau berupa *tombstone* (nisan). Pencarian melewati nisan dan hanya berhenti di slot kosong.

$$S \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right), U \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right). \tag{16.11}$$

Persamaan (16.11): pada $\alpha = 9/11$, sekitar 3,3 dan 15,6 probe. Mengosongkan slot dapat memutus lintasan probe kunci yang terdorong melewatinya. Buku: Subbab 16.7.2.

Contoh Terurai: Nisan di Slot 5



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.

Gambar 16.14: 59 menyelidiki 4, 5, 6, 7, 8. Setelah 15 dihapus, pencarian 59 melewati nisan; 37 memakai ulang slot 5 hanya setelah mencapai slot kosong 2. Buku: Subbab 16.7.2 dan Gambar 16.14.

`search` dan `remove` dalam C++

Kode 16.11 `search` dan `remove` dari `p1609_linear_probing.cpp`: EMPTY mengakhiri pencarian, DELETED tidak.

```
    // 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
    }

    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;
    }
```

Kode 16.11. `insert` *mengingat slot bebas pertama tetapi terus berjalan sampai slot EMPTY untuk membuktikan kuncinya tidak ada. Buku: Subbab 16.7.2.*

Keliru: Menghapus dengan Mengosongkan Slot

```
before: search 59 -> slot 8
after removing 15: search 59 -> slot -1
```

59 masih di slot 8, tetapi pencariannya bertemu slot 5 yang dikosongkan lalu berhenti: tabel *kehilangan* kunci yang dimilikinya.

Buku ada di rak, tetapi komputer bersikeras buku itu tidak ada. *Chaining* tidak dapat membuat kesalahan ini.

Nisan membuat setiap barisan operasi aman, bukan hanya satu; rehash menyapu bersih nisan-nisan itu. Buku: Subbab 16.7.2.

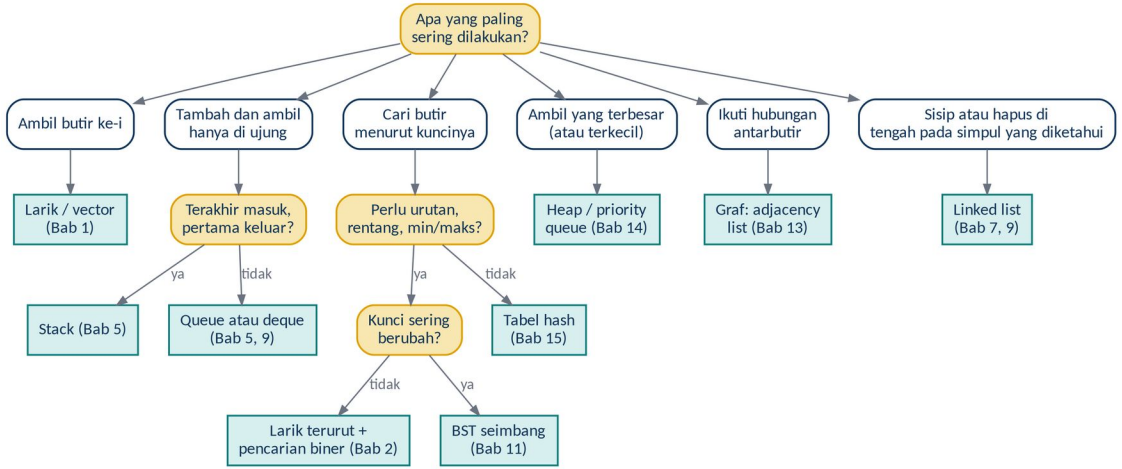
Memilih Struktur Data yang Tepat

Tabel 16.4 Memilih struktur data menurut operasi yang paling sering dikerjakan tugas (bentuk ringkas tabel buku).

Tugas paling sering perlu…	Pilih	Ongkos	Bab
membaca atau menulis butir ke-i	larik / `vector`	$O(1)$	1
mencari menurut kunci, mendaftar berurutan, jarang berubah	larik terurut + pencarian biner	$O(\lg n)$	2, 3
menambah dan mengambil di ujung terbaru	*stack*	$O(1)$	5
melayani menurut urutan kedatangan	*queue*	$O(1)$	5, 9
menyisipkan atau menghapus di simpul yang dipegang	*linked list*	$O(1)$	7, 9
mencari menurut kunci, plus urutan atau rentang, sering berubah	BST seimbang	$O(\lg n)$	11
berulang kali mengambil yang terbesar atau terkecil	*heap* / *priority queue*	$O(\lg n)$	14
mencari, menyisipkan, atau menghapus menurut kunci, tidak lebih	tabel hash	rata-rata $O(1)$	15
memodelkan hubungan	graf dengan *adjacency list*	$O(\lvert V\rvert+\lvert E\rvert)$	13

Tabel 16.4: seluruh buku dalam satu halaman. Namai operasi yang dominan lebih dahulu. Buku: Subbab 16.8.1.

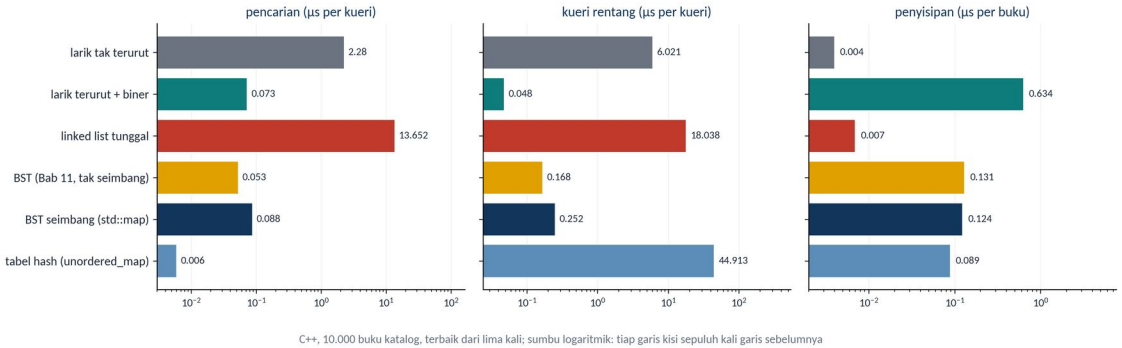
Keputusan sebagai Pohon



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.

Gambar 16.15: "mencari menurut kunci" berujung pada tabel hash hanya bila Anda tidak pernah memerlukan urutan; dengan "terkecil", "berikutnya", atau "di antara", pilih pohon pencarian atau larik terurut. Buku: Subbab 16.8.1 dan Gambar 16.15.

Percobaannya: Enam Struktur, Satu Katalog



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.

Gambar 16.16: membangun, 2.000 pencarian (separuhnya tidak ada), 200 kueri rentang, 1.000 penyisipan pada 10.000 identitas nyata; setiap struktur harus memberi jawaban yang sama. Buku: Subbab 16.8.2–16.8.3 dan Gambar 16.16.

Membaca Angka-Angkanya

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

$$|V|^2 = 5841^2 = 34{,}117{,}281 \text{ sel dibandingkan } |V| + 2|E| = 19{,}835 \text{ entri}, \tag{16.12}$$

Tabel 16.5: tabel hash unggul dalam pencarian dan kalah dalam rentang; larik terurut terbaik untuk rentang dan paling lambat disisipi. Persamaan (16.12): graf orang (5.841 orang) tempatnya di adjacency list. Buku: Subbab 16.8.3.

Perbandingan Soal-Soal

Tabel 16.6 Kesembilan soal: struktur yang dipakai masing-masing, ongkosnya, dan masukan tempat penyelesaian kelirunya gagal.

Soal	Struktur	Waktu	Penyelesaian keliru gagal pada
16.1	*max-heap* larik	$O(\lg n)$; bangun $O(n)$	elemen terakhir yang bukan minimum
16.2	dua *hash map*	rata-rata $O(n)$	tiket pertama yang bukan potongan pertama
16.3	*adjacency list* + *queue*	$O(\lvert V\rvert+\lvert E\rvert)$	DFS yang mencapai simpul lewat jalan jauh
16.4	*adjacency list* + *queue*	$O(\lvert V\rvert+\lvert E\rvert)$	siklus berarah
16.5	*adjacency list* + *min-heap*	$O((\lvert V\rvert+\lvert E\rvert)\lg\lvert V\rvert)$	jalan langsung yang lebih lama daripada jalan memutar
16.6	*min-heap* berukuran k	$O(n \lg k)$	masukan apa pun, dengan *max-heap*
16.7	*max-heap* + *min-heap*	$O(\lg n)$ per nilai	nilai keempat, tanpa penyeimbangan
16.8	tabel hash berantai	rata-rata $O(1+\alpha)$	kunci yang meluapkan `int`
16.9	*linear probing*	rata-rata $O(1/(1-\alpha)^2)$	lintasan melewati slot terhapus

Tabel 16.6: sebagian besar penyelesaian keliru yang menggoda lolos pada contoh yang diberikan. Tiga keluarga ongkos: satu lintasan menuruni pohon, satu kali lewat, satu lompatan terhitung. Buku: Subbab 16.9.

Profitina dalam Praktik

Situs web adalah graf: halaman adalah simpul, tautan adalah busur; halaman yatim tidak terjangkau, kedalaman klik adalah jarak BFS.

Himpunan "sudah dilihat" adalah tabel hash: URL dikanonisasi sebelum di-*hash*, agar satu halaman tidak tersimpan tiga kali.

Pemindaian ulang adalah *priority queue* berkunci waktu jatuh tempo; **daftar teratas** adalah *heap* berukuran k.

Profitina adalah platform visibilitas AI (GEO); servernya ditulis dalam Python, sehingga struktur ini muncul sebagai `dict`, `set`, `deque`, *dan* `heapq`. *Kita melukiskan bentuk setiap soal, bukan cara produknya dibangun. Buku: Subbab 16.10.*

Kesalahan yang Sering Terjadi

Heap: ujung *priority queue* yang salah; *max-heap* untuk k terbesar; tidak menyimpan akar; perbaikan ke arah yang salah; *underflow* `size() - 2`; lupa menyeimbangkan dua *heap*.

Graf: kedalaman DFS sebagai jarak; menandai saat keluar dari *queue*; *topological sort* tanpa menghitung; jarak Dijkstra yang dikunci pada penglihatan pertama.

Tabel hash: menyebut `std::map` *hash map*; bergantung pada urutan iterasi; hash dalam `int` bertanda; menghapus dengan mengosongkan slot; membiarkan tabel penuh.

Dan memilih struktur sebelum menamai operasinya: namai, lihat Tabel 16.4, lalu ukur. Buku: Subbab 16.11.

Ringkasan: Lima Hal untuk Diingat

1. Sembilan langkah: nyatakan ulang, pilih, gambar, invarian, kasus tepi, urutan aman, ongkos, masukan buruk, uji terhadap kembaran yang lambat.

2. Setiap struktur menjanjikan invarian; setiap operasi harus membiarkannya tetap benar.

3. *Heap*: tinggi $\lfloor \lg n \rfloor$; minimum di antara daun; dibangun dalam $O(n)$; k terbesar memerlukan *min*-heap; median memerlukan dua.

4. Graf: BFS untuk lompatan paling sedikit, Kahn untuk urutan atau siklus, Dijkstra dengan *heap* untuk rute berbobot. Tabel hash: jaga α tetap kecil, tanpa luapan, pakai nisan.

5. Pilih menurut operasi yang dominan, lalu ukur.

Buku: Subbab 16.12.

Langkah Selanjutnya

Ini bab terakhir buku ini. Larik, *list*, *stack* dan *queue*, rekursi, pohon, graf, *heap*, dan tabel hash kini menjadi satu kotak perkakas.

Kerjakan Soal Latihan 16.1, 16.3, 16.10, dan 16.12, dan bangun meja layanan Profitina Laundry pada Soal Latihan 16.14: *hash map* pesanan dan *priority queue* jam yang dijanjikan.

Menghadapi tugas nyata: namai operasinya, pilih dari Tabel 16.4, dan ukur sebelum memercayai keduanya.

Penyelesaiannya, dengan program dalam ketiga bahasa, ada di buku. Buku: Subbab 16.13 dan 16.14.