Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

DAFTAR ISI

Penyegaran C/C++ & Model Memori → Fondasi Array 7
1.1 Selamat Datang di Struktur Data 7
1.2 Penyegaran C/C++: Variabel, Memori, dan Pratinjau Pointer 8
1.3 Tata Letak Memori dan Aritmatika Alamat 10
1.4 Array 1D: Deklarasi, Pengisian, dan Penelusuran 12
1.5 Array 2D: Rak sebagai Grid 12
1.6 Lampiran 1.A — Log Validasi 13
1.7 Ringkasan Bab dan Poin-Poin Kunci 15
1.8 Pertanyaan Tinjauan 15
Referensi 16

Pencarian 18
2.1 Mengapa Pencarian Penting: Kontrak Ditemukan / Tidak Ditemukan 18
2.2 Sequential (Linear) Search 19
2.3 Sentinel Search: Menghilangkan Pemeriksaan Batas 20
2.4 Binary Search: Membagi Dua Masalah Setiap Langkah 21
2.5 Interpolation Search: Menebak Lebih Cerdas daripada Titik Tengah 23
2.6 Membandingkan Empat Algoritma: Memperkenalkan Notasi Big-O 24
2.7 Lampiran 2.A — Log Validasi 26
2.8 Ringkasan Bab dan Poin-Poin Kunci 28
2.9 Pertanyaan Tinjauan 28
Referensi 29

Pengurutan 31
3.1 Ulasan: Mengapa Pengurutan Penting, dan Apa yang Sudah Kita Ketahui 31
3.2 Bubble Sort 32
3.3 Exchange Sort 34
3.4 Selection Sort 34
3.5 Insertion Sort 35
3.6 Quicksort 36
3.7 Merge Sort 38
3.8 Heap Sort 39
3.9 Membandingkan Ketujuhnya: Kompleksitas dan Stabilitas 40
3.10 Lampiran 3.A — Log Validasi 42
3.11 Ringkasan Bab dan Poin-Poin Kunci 46
3.12 Pertanyaan Tinjauan 47
Referensi 48

Kuis 1 50
4.1 Ulasan: Bab 2 dan 3 Secara Singkat 50
4.2 Cara Menggunakan Bab Latihan Ini 51
4.3 Kumpulan Soal Kuis 1 52
4.4 Format dan Penilaian Kuis 1 55
4.5 Ringkasan Bab 56
Lampiran 4.A — Daftar Periksa Mandiri (Tidak Dinilai) 56
Referensi 57

Stack & Queue 59
5.1 Ulasan: Bab 1-4 Sejauh Ini 59

5.2 Stack: LIFO, dan Sebuah Teka-teki Motivasi 60
5.3 Stack Berbasis Array yang Lengkap 61
5.4 Studi Kasus: Kalkulator Ilmiah 62
5.5 Queue: FIFO dan Intuisi Dunia Nyata 66
5.6 Queue Berbasis Array yang Lengkap, dan Masalah "False Full" 67
5.7 Stack vs. Queue: Memilih Alat yang Tepat 70
5.8 Lampiran 5.A — Log Validasi 71
5.9 Rangkuman Bab dan Poin-Poin Kunci 75
5.10 Pertanyaan Ulasan 76
Referensi 77

Pointer & Fungsi 79
6.1 Ulasan: Pratinjau Pointer Bab 1, Dilengkapi di Sini 79
6.2 Fondasi Pointer 80
6.3 Aritmetika Pointer, Dilengkapi 82
6.4 Fungsi: Deklarasi vs. Definisi 84
6.5 Pass-by-Value vs. Pass-by-Reference vs. Pass-by-Pointer 85
6.6 Menoleh ke Belakang, dan ke Depan: Bab 5 dan Bab 7 88
6.7 Lampiran 6.A — Log Validasi 89
6.8 Rangkuman Bab dan Poin-Poin Kunci 93
6.9 Pertanyaan Ulasan 94
Referensi 95

Singly Linked List 97
7.1 Ulasan: Mengapa Linked List, Padahal Kita Sudah Punya Array? 97
7.2 Node: Tempat Pointer Bab 6 Menjadi Krusial 99
7.3 Singly Linked List Head-Only 99
7.4 Singly Linked List Head + Tail 101
7.5 Circular Singly Linked List (Baru di Bab Ini) 103
7.6 Menoleh ke Depan: Doubly Linked List Bab 9 105
7.7 Lampiran 7.A — Log Validasi 105
7.8 Rangkuman Bab dan Poin-Poin Kunci 109
7.9 Pertanyaan Ulasan 109
Referensi 110

Ujian Tengah Semester (UTS) 112
8.1 Ulasan: Kuliah 5-7 Secara Singkat 112
8.2 Cara Menggunakan Bab Latihan Ini 113
8.3 Kumpulan Soal UTS 114
8.4 Format dan Penilaian UTS 119
8.5 Ringkasan Bab 120
Lampiran 8.A — Daftar Periksa Mandiri (Tidak Dinilai) 121
Referensi 122

Doubly Linked List 124
9.1 Ulasan: Apa yang Tidak Bisa Dilakukan Singly Linked List Bab 7 124
9.2 Node: Satu Pointer Baru, prev 125
9.3 Deklarasi dan Inisialisasi Headed (Head+Tail) 125
9.4 Penyisipan Depan dan Belakang: Empat Pointer, Bukan Dua 126
9.5 Penghapusan Depan dan Belakang: Hasilnya — O(1) Akhirnya 128
9.6 Pencarian dan Penjelajahan Dua Arah 129
9.7 Circular Doubly Linked List (Baru di Bab Ini) 129

9.8 Kapan Pointer Tambahan Itu Sepadan? DLL vs. SLL....131
9.9 Menoleh ke Belakang: Menutup Celah yang Dimulai Bab 8....132
9.10 Lampiran 9.A — Log Validasi....132
9.11 Rangkuman Bab dan Poin-Poin Kunci....134
9.12 Pertanyaan Ulasan....135
Referensi....135

Rekursi....135
10.1 Fondasi Rekursi: Base Case, Recursive Case, dan Call Stack....137
10.2 Rekursi di Wilayah yang Familiar: Katalog Pustaka Ceria....138
10.3 Faktorial: Contoh Rekursif Pertama yang Klasik....139
10.4 Fibonacci — Lima Cara....140
10.5 Empat Varian Fibonacci Tail-Recursive....143
10.6 GCD via Algoritma Euclidean....145
10.7 Menara Hanoi — Menutup Celah yang Dimulai Bab 5....145
10.8 Eksponensiasi: Naif O(n) vs. Cepat O(log n)....147
10.9 Intro Analisis Algoritma: Best, Average, dan Worst Case....148
10.10 Lampiran 10.A — Log Validasi....148
10.11 Rangkuman Bab dan Poin-Poin Kunci....150
10.12 Pertanyaan Ulasan....151
Referensi....152

Tree dan Binary Search Tree (BST)....154
11.1 Fondasi dan Terminologi Tree....154
11.2 Mengklasifikasikan Binary Tree: Full, Complete, dan Skewed....155
11.3 Properti Binary Search Tree (BST) — Mengapa Ini Penting....156
11.4 Contoh Kerja Berulang: insert(65, 5, 70, 3, 10)....157
11.5 Traversal: PreOrder, InOrder, PostOrder, dan LevelOrder....158
11.6 Operasi BST Lainnya: Search, Count, Height, Minimum, Leaf-Finder....160
11.7 General Tree vs. Binary Tree: Konversi dan Rekonstruksi....162
11.8 Delete-Node — Ketiga Kasus (Baru di Bab Ini)....163
11.9 Rangkuman Bab dan Poin-Poin Kunci....166
11.10 Pertanyaan Ulasan....167
11.11 Lampiran 11.A — Log Validasi....167
Referensi....168

Kuis 2....171
12.1 Ulasan: Bab 11 Secara Singkat....171
12.2 Cara Menggunakan Bab Latihan Ini....173
12.3 Kumpulan Soal Kuis 2....173
12.4 Format dan Penilaian Kuis 2....180
12.5 Ringkasan Bab....182
Lampiran 12.A — Daftar Periksa Mandiri (Tidak Dinilai)....183
Referensi....183

Graf....186
13.1 Terminologi dan Dasar-Dasar Graf....186
13.2 Merepresentasikan Graf dalam Kode....188
13.3 Traversal: Mengunjungi Setiap Simpul yang Terjangkau....192
13.4 Konektivitas: Connected Components dan "Apakah Graf Ini Terhubung?"....195
13.5 Ringkasan Bab dan Poin-Poin Kunci....197
13.6 Pertanyaan Tinjauan....198

13.7 Lampiran 13.A — Log Validasi 198
Referensi 199

Heap & Priority Queue 202
14.1 Konsep Priority Queue 202
14.2 Binary Heap sebagai Array 203
14.3 Operasi Inti: insert, deleteRoot/extractMax, getMax 205
14.4 Membangun Heap dari Array Sembarang 208
14.5 Lima Fungsi Ujian Akhir 209
14.6 isMaxHeap() — Pemeriksa Validitas 211
14.7 Heap Sort — Melengkapi Survei Pengurutan Bab 3 212
14.8 Ringkasan Bab dan Poin-Poin Kunci 213
14.9 Pertanyaan Tinjauan 214
14.10 Lampiran 14.A — Log Validasi 214
Referensi 215

Hashing & Tabel Hash 218
15.1 Motivasi: Langkah Berikutnya Setelah Searching dan BST 218
15.2 Fungsi Hash 220
15.3 Penyelesaian Tabrakan: Separate Chaining 221
15.4 Load Factor dan Rehashing 223
15.5 std::unordered_map — Tabel Hash Sungguhan Milik STL Sendiri 224
15.6 Contoh Kerja: Merekonstruksi Itinerari dengan HashMap 225
15.7 Kaitan Aplikasi: Pencarian Buku Pustaka Ceria 228
15.8 Ringkasan Bab dan Poin Penting 229
15.9 Soal Latihan 229
15.10 Lampiran 15.A — Log Validasi 230
Referensi 232

Ujian Akhir Semester (UAS) 234
16.1 Ulasan: Bab 14-15 Secara Singkat 234
16.2 Cara Menggunakan Bab Latihan Ini 235
16.3 Tugas 1 — Lengkapi Skeleton MaxHeap (50 poin, 5 × 10) 236
16.4 Tugas 2 — Hashing: Rekonstruksi Itinerary + Laporan Tertulis (50 poin) 242
16.5 Format dan Penilaian Ujian Akhir 245
16.6 Ringkasan Bab — dan Akhir Mata Kuliah Ini 246
Lampiran 16.A — Daftar Periksa Mandiri (Tidak Dinilai) 246
Referensi 247

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 1

Penyegaran C/C++ & Model Memori → Fondasi Array

Bab pertama dari mata kuliah ini, dan fondasi yang menjadi dasar setiap bab selanjutnya. Setiap baris kode di bab ini benar-benar dikompilasi dengan g++ -Wall -Wextra dan dijalankan — transkrip terminal aslinya direproduksi di Lampiran 1.A.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Menjelaskan, dalam satu paragraf, apa yang dipelajari "Struktur Data" sebagai sebuah disiplin ilmu dan mengapa hal ini penting seiring bertambahnya program dan data. (2) Membedakan nilai suatu variabel dari alamatnya, dan membaca sintaks pratinjau `&` / pointer dasar dalam C/C++. (3) Menjelaskan bagaimana array dari sebuah struct disusun secara berurutan (contiguous) di memori, dan menurunkan alamat elemen mana pun dari alamat basis, indeks, dan `sizeof`. (4) Mendeklarasikan, mengisi, dan menelusuri array 1D dari tipe struct. (5) Mendeklarasikan array 2D dan menurunkan rumus aritmatika alamat row-major untuk `arr[baris][kolom]` mana pun.

Satu Toolchain, Tiga Sistem Operasi — Windows 11, macOS, Ubuntu

OS	Instalasi sekali saja	Kompilasi & jalankan (identik di mana pun)
Windows 11	MinGW-w64: winget install BrechtSanders.WinLibs (atau WSL2: wsl --install)	g++ -Wall -Wextra -std=c++17 -O2 -o prog prog.cpp lalu prog.exe / ./prog
macOS	xcode-select --install (clang Apple menyahut sebagai g++)	g++ -Wall -Wextra -std=c++17 -O2 -o prog prog.cpp lalu ./prog
Ubuntu/Linux	sudo apt update && sudo apt install build-essential	g++ -Wall -Wextra -std=c++17 -O2 -o prog prog.cpp lalu ./prog

Semua contoh di buku ini berjalan identik di Windows 11, macOS, dan Ubuntu/Linux; perintah hanya berbeda saat instalasi. Pilih baris Anda sekali dan ikuti terus di laptop Anda sendiri.

1.1 Selamat Datang di Struktur Data

"Struktur Data" adalah kajian tentang bagaimana mengorganisasikan data di memori sehingga operasi-operasi yang benar-benar dibutuhkan sebuah program — menyisipkan, menghapus, mencari, mengurutkan, dan menelusuri — berjalan secepat dan semudah mungkin untuk dinalar. Kumpulan data yang sama bisa disimpan dengan selusin cara berbeda, dan pilihan cara itu punya konsekuensi nyata yang terukur: pencarian yang hanya butuh mikrodetik pada satu tata letak bisa memakan waktu berdetik-detik pada tata letak lain, begitu koleksinya cukup besar. Mata kuliah ini bukan tentang mempelajari sepuluh trik yang tidak berhubungan; ini tentang mempelajari sejumlah kecil ide tata letak — array yang berurutan, rantai node yang tertaut, pohon hierarkis, graf yang saling terhubung, dan ember (bucket) yang di-hash — dan, untuk masing-masing, memahami persis apa yang dibuatnya cepat, apa yang dibuatnya lambat, dan mengapa.

Mengapa ini penting dalam praktiknya? Setiap program non-trivial menyimpan data: aplikasi buku telepon menyimpan kontak, sebuah game menyimpan pemain beserta inventarisnya, sebuah compiler menyimpan simbol-simbol yang sudah ditemuinya. Seiring jumlah item bertambah dari

sepuluh menjadi sepuluh ribu menjadi sepuluh juta, operasi yang memindai setiap item satu per satu — baik-baik saja pada sepuluh item — menjadi alasan mengapa sebuah aplikasi nyata terasa lambat. Bagian besar dari pekerjaan seorang programmer adalah mengenali struktur mana yang dibutuhkan oleh masalah yang sedang dihadapi, dan mata kuliah ini dirancang untuk memberi Anda kemampuan penilaian tersebut, bukan sekadar sintaksnya.

> **Peta jalan satu semester**
> Enam belas bab, satu semester. Bab 1–3 membangun fondasi array/pencarian/pengurutan; Bab 4 adalah Kuis 1. Bab 5–7 membahas stack, queue, pointer/fungsi, dan singly linked list; Bab 8 adalah UTS. Bab 9–11 membahas doubly linked list, rekursi, dan tree; Bab 12 adalah Kuis 2. Bab 13–15 membahas graph, heap, dan hashing — materi yang selama ini menjadi tujuan setiap bab sebelumnya — dan Bab 16 adalah UAS. Bab-bab asesmen (4, 8, 12, 16) menguji tiga bab materi tepat sebelumnya dan tidak menyertakan subfolder code/ sendiri.

1.1.1 Berkenalan dengan Pustaka Ceria

Alih-alih berganti contoh setiap bab, mata kuliah ini mengikuti satu studi kasus yang berjalan dari Bab 1 hingga Bab 15: Pustaka Ceria, sebuah perpustakaan kampus yang kecil dan ramah. Pustaka Ceria punya masalah nyata dan sederhana: ia perlu menyimpan katalog bukunya, mencari buku berdasarkan kode atau judul, menjaga katalog tetap terurut untuk keperluan penelusuran, membiarkan pustakawan melakukan push dan pop pada stack buku “baru dikembalikan, belum ditata ulang di rak”, menelusuri queue permintaan reservasi sesuai urutan kedatangannya, mengorganisasikan seluruh koleksinya sebagai pohon kategori yang bisa ditelusuri, memodelkan buku-buku mana yang sering dipinjam bersamaan sebagai sebuah graf, dan — begitu katalognya membesar — mencari buku berdasarkan kodenya dalam waktu konstan menggunakan tabel hash.

Setiap kebutuhan tersebut memetakan langsung ke sebuah bab di mata kuliah ini. Bab 1 mengerjakan versi paling dasar dari kebutuhan pertama: memberi setiap buku sebuah bentuk yang tetap dan dapat diprediksi di memori (sebuah `struct Book`), dan menyimpan beberapa di antaranya secara berurutan dalam sebuah array. Pustaka Ceria bukanlah kisah nyata dari institusi mana pun — ia semata-mata ada untuk memberi kode setiap bab satu kumpulan data bersama yang mudah dikenali, alih-alih contoh baru yang diciptakan setiap kali.

1.2 Penyegaran C/C++: Variabel, Memori, dan Pratinjau Pointer

Bagian ini adalah penyegaran, bukan materi baru — Anda hampir pasti sudah pernah menulis variabel, tipe primitif, dan fungsi sederhana sebelumnya. Yang penting bagi mata kuliah ini adalah model mental di baliknya: setiap variabel yang dideklarasikan program C/C++ menempati sebuah lokasi nyata yang dapat dialamatkan di memori komputer, dan memahami lokasi tersebut adalah dasar dari sisa bab ini (dan, lebih langsung lagi, cakupan pointer penuh di Bab 6).

1.2.1 Variabel dan Tipe Primitif

Deklarasi variabel seperti `int umur = 20;` melakukan dua hal sekaligus: ia mencadangkan blok memori berukuran tetap (4 byte, untuk `int` yang umum), dan memberi blok tersebut sebuah nama — `umur` — yang bisa dipakai sisa program alih-alih melacak lokasi memori mentahnya secara manual. Tipe-tipe primitif C/C++ masing-masing mencadangkan jumlah byte tetap yang berbeda: `char` (1 byte), `int` (umumnya 4 byte), `float` (4 byte), `double` (8 byte), dan seterusnya — ukuran pastinya bergantung pada implementasi, tetapi `sizeof(tipe)` selalu memberi tahu jawaban yang sebenarnya pada mesin tempat Anda mengompilasi, dan itulah persis alat yang diandalkan bab ini.

1.2.2 Stack vs. Heap, secara Konseptual

Program C/C++ yang sedang berjalan memiliki (setidaknya) dua wilayah memori yang sangat berbeda untuk variabel. Stack adalah tempat variabel lokal biasa hidup — setiap variabel yang Anda deklarasikan di dalam sebuah fungsi (seperti `catalog` pada program demo bab ini) ditempatkan di stack secara otomatis saat fungsi dimulai, dan diklaim ulang secara otomatis begitu fungsi itu selesai (return). Stack cepat dan tidak memerlukan pencatatan manual, tetapi ukurannya tetap dan cukup terbatas (sering kali hanya beberapa megabyte), dan umur (lifetime) sebuah variabel terikat ketat pada fungsi tempat ia dideklarasikan. Heap, sebaliknya, adalah memori yang diminta program secara eksplisit saat runtime (di C++, dengan `new`; di C, dengan `malloc`) dan harus dilepaskan secara eksplisit ketika sudah selesai (`delete` / `free`) — ia bertahan melampaui fungsi yang menciptakannya, dengan konsekuensi programmer kini bertanggung jawab mengelola umurnya secara manual.

> **Mengapa di sini hanya pratinjau**
>
> Setiap `struct Book` yang dibuat bab ini hidup di stack — dideklarasikan sebagai array lokal biasa, tanpa `new`/`malloc` di mana pun. Itu disengaja: seluruh tugas bab ini adalah model memori dari array yang berurutan, dan mencampurkan alokasi heap serta kepemilikan pointer pada saat bersamaan akan mengaburkan dua pelajaran terpisah menjadi satu. Bab 6 (Pointer & Fungsi) adalah tempat alokasi heap, `new`/`delete`, dan pass-by-reference mendapat pembahasan penuh yang layak mereka dapatkan.

1.2.3 Alamat: Operator `&` dan Pratinjau Pointer

Setiap variabel, begitu ia ada, memiliki dua hal yang layak ditanyakan: nilainya, dan alamatnya — di mana ia hidup di memori. C/C++ memberi Anda operator address-of, `&`, untuk menanyakan pertanyaan kedua itu secara langsung: diberikan `int umur = 20;`, ekspresi `&umur` adalah alamat memori tempat 4 byte milik `umur` hidup, bukan nilai 20 itu sendiri. Pointer hanyalah sebuah variabel yang nilainya sendiri berupa alamat, bukan angka biasa; `int *p = &umur;` mendeklarasikan `p` sebagai “pointer ke `int`” dan menginisialisasinya untuk menyimpan alamat `umur`, dan `*p` (men-dereference `p`) membaca `int` yang tersimpan di alamat tersebut — yaitu 20, nilai yang sama dengan yang dipegang `umur` sendiri.

```c
int umur = 20;
int *p = &umur;      // p kini menyimpan alamat umur

printf("%d\n", umur);   // 20  -- nilainya
printf("%p\n", (void*)&umur); // mis. 0x7ffd1234abcd -- alamatnya
printf("%d\n", *p);   // 20  -- dereference p membaca nilai yang sama
```

Ini pratinjau, bukan materi lengkap

Bagian ini memperkenalkan `&` dan sintaks pointer secukupnya agar aritmatika alamat di Bagian 1.3 dapat dipahami — dengan sengaja berhenti jauh sebelum aritmatika pointer pada tipe sembarang, pengiriman pointer ke fungsi, atau alokasi dinamis. Bab 6 adalah tempat pointer menjadi topik utama tersendiri; perlakukan setiap contoh pointer antara di sini dan Bab 6 sebagai latar belakang untuk dibaca saja, bukan sesuatu yang diharapkan bisa Anda tulis sendiri dari nol.

1.2.4 Rekaman Fondasi: struct Book

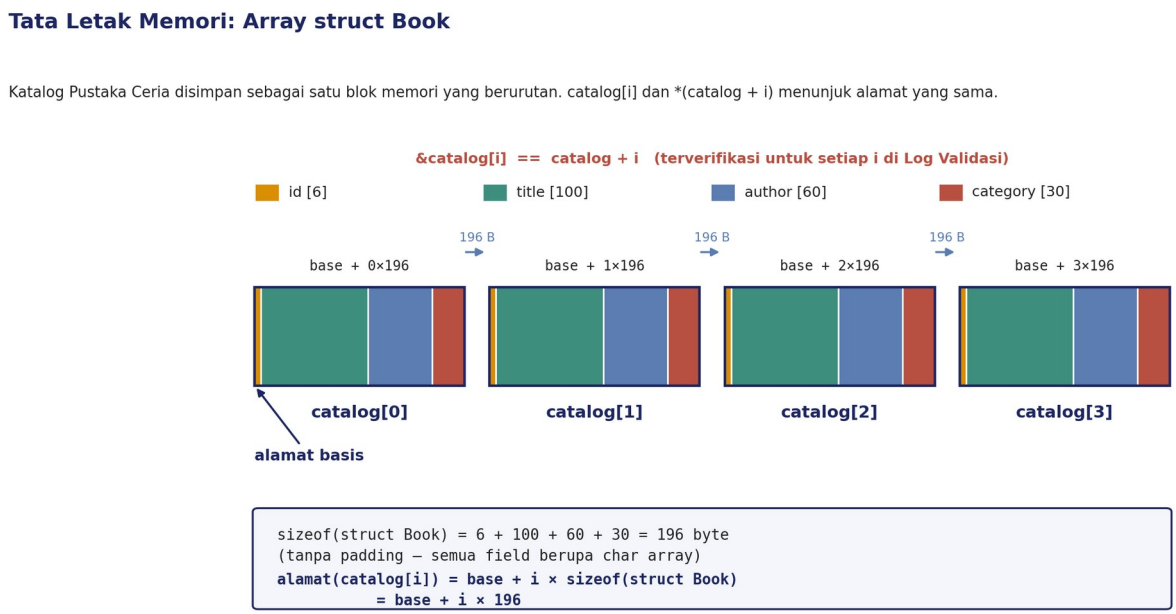
Katalog Pustaka Ceria adalah kumpulan buku, dan setiap buku perlu dicatat empat informasi yang sama: kode identitas singkat, judul, pengarang, dan kategori. Kata kunci `struct` di C/C++ memungkinkan kita mendeskripsikan bentuk itu persis sekali saja, sebagai tipe baru:

```c
struct Book {
    char id[6];         // mis. "B1001" (5 karakter + '\0')
    char title[100];
    char author[60];
    char category[30];
};
```

Setiap field di sini adalah array `char` berukuran tetap, bukan string berukuran dinamis — dengan sengaja, untuk bab ini: struct berukuran tetap tanpa pointer di dalamnya adalah hal paling sederhana yang bisa disusun berurutan di memori, dan itu persis yang perlu dijelaskan Bagian 1.3 dengan bersih. `struct Book` adalah satu-satunya tipe rekaman yang dipakai ulang setiap bab Pustaka Ceria selanjutnya — sebagai muatan (payload) sebuah node di linked list (Bab 7), sebagai nilai yang disimpan di tree (Bab 11), sebagai rekaman yang dipetakan tabel hash dari sebuah kode (Bab 15) — jadi ada baiknya bentuknya sudah benar sejak sekarang.

1.3 Tata Letak Memori dan Aritmatika Alamat

Deklarasikan sebuah array dari `struct Book` — `struct Book catalog[8];` — dan C/C++ membuat sebuah janji yang sangat spesifik: kedelapan `Book` tersebut disusun berurutan (back-to-back) dalam satu blok memori yang berurutan, tanpa celah dan tanpa perubahan urutan. `catalog[0]` menempati `sizeof(struct Book)` byte pertama dari blok tersebut, `catalog[1]` menempati `sizeof(struct Book)` byte berikutnya, dan seterusnya. Gambar 1.1 menunjukkan hal ini untuk struct `Book` kita yang memiliki 4 field.



Gambar 1.1 — Array struct Book yang disusun berurutan di memori, dengan offset byte untuk setiap elemen.

Tata letak inilah yang membuat pengindeksan array, `catalog[i]`, setara dengan aritmatika pointer, `*(catalog + i)`: nama array `catalog` meluruh (decay) menjadi pointer ke elemen pertamanya, dan menambahkan `i` ke pointer tersebut memajukannya persis sebesar `i * sizeof(struct Book)` byte — bukan `i` byte — karena aritmatika pointer pada `T*` selalu bergerak dalam satuan `sizeof(T)`. Itulah persis mengapa rumus di Gambar 1.1 berbunyi `alamat(catalog[i]) = base + i × sizeof(struct Book)`: compiler sedang melakukan persis perkalian itu setiap kali `catalog[i]` muncul di kode Anda.

> **sizeof(struct Book) tidak punya padding tersembunyi di sini**
>
> Pada kebanyakan struct, compiler menyisipkan byte padding yang tak terlihat di antara field-field agar setiap field dimulai pada alamat yang selaras dengan kebutuhan alignment tipenya sendiri (field `int`, misalnya, biasanya ditempatkan pada offset yang selaras 4-byte). `struct Book` sepenuhnya menghindari hal ini: setiap field adalah array `char`, dan `char` punya kebutuhan alignment persis 1 byte, sehingga compiler tidak punya apa pun untuk diselaraskan dan tidak menambahkan padding sama sekali. `sizeof(struct Book)` karenanya persis 6 + 100 + 60 + 30 = 196 byte — dikonfirmasi oleh keluaran program yang benar-benar dikompilasi di Lampiran 1.A. Begitu sebuah struct mencampurkan field `char` dengan field `int`/`double`, `sizeof` bisa menjadi jauh lebih besar dari jumlah sederhana ukuran field; itu adalah topik untuk bab selanjutnya, bukan bab ini.

`catalog[i]` dan `*(catalog + i)` bukan sekadar setara dalam apa yang mereka hitung — mereka adalah persis alamat yang sama, setiap saat, dan ini adalah sesuatu yang bisa Anda verifikasi langsung alih-alih menerimanya begitu saja:

```
for (int i = 0; i < N; i++) {
    void *byIndex        = (void *) &catalog[i];
    void *byPointerArith = (void *) (catalog + i);
    // byIndex == byPointerArith, selalu
}
```

1.4 Array 1D: Deklarasi, Pengisian, dan Penelusuran

Dengan `struct Book` sudah didefinisikan dan tata letak memorinya dipahami, mendeklarasikan seluruh katalog Pustaka Ceria sebagai array 1D hanyalah satu baris:

```
struct Book catalog[N];   // N = 8 untuk demo bab ini
```

Mengisinya berarti mengisi field-field setiap elemen — fungsi bantu `populateBook()` di bab ini menyalin empat C-string ke dalam field-field berukuran tetap milik `Book`, selalu menyisakan setiap field diakhiri null meskipun string yang diberikan pemanggil lebih panjang dari kapasitas field (memotong dengan aman alih-alih meluap ke buffer):

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

Menelusuri katalog — mengunjungi setiap elemen persis satu kali, secara berurutan — adalah perulangan `for` biasa yang dipakai setiap program C/C++ pada sebuah array:

```
for (int i = 0; i < N; i++) {
    printBook(catalog[i]);
}
```

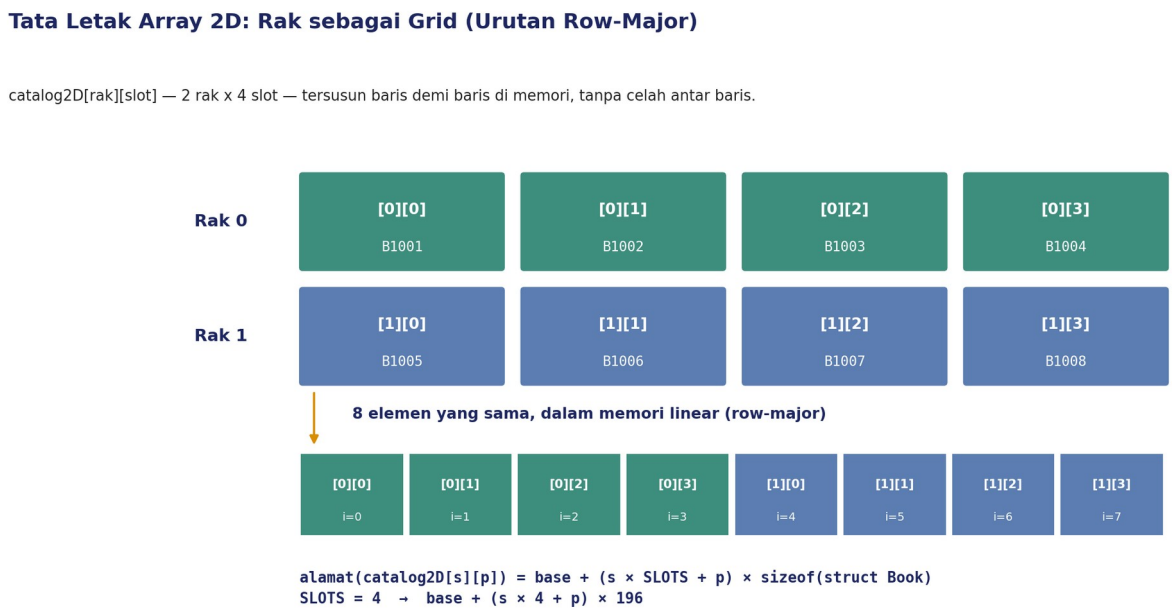
Satu bentuk perulangan ini — deklarasikan, isi, telusuri — adalah pola yang menjadi dasar kode berbasis array setiap bab selanjutnya (pencarian di Bab 2, pengurutan di Bab 3).

1.5 Array 2D: Rak sebagai Grid

Gedung Pustaka Ceria memiliki struktur fisik alami yang tidak ditangkap oleh katalog 1D yang datar: buku-buku duduk di rak fisik, dan setiap rak memiliki jumlah slot yang tetap. Array 2D memodelkan ini secara langsung — `struct Book catalog2D[SHELVES][SLOTS];` mendeklarasikan sebuah grid di mana `catalog2D[s][p]` adalah buku di rak `s`, slot `p`.

```
const int SHELVES = 2;
const int SLOTS = 4;
struct Book catalog2D[SHELVES][SLOTS];
```

C/C++ menyusun array 2D secara row-major: seluruh baris pertama (`SLOTS` buku milik rak 0) menempati `SLOTS × sizeof(struct Book)` byte pertama, langsung diikuti oleh seluruh baris kedua (rak 1), tanpa celah di antaranya — persis seolah-olah grid 2D tersebut “digulung terbuka” menjadi satu array 1D panjang dalam urutan baris. Gambar 1.2 menunjukkan ini secara langsung: kedelapan buku yang sama, pertama sebagai grid 2 rak, kemudian digulung terbuka menjadi satu blok berurutan yang benar-benar mereka tempati di memori.



Gambar 1.2 — Array 2D dari struct Book (rak × slot) dan tata letak memori linearnya secara row-major.

Menggeneralisasi rumus alamat 1D di Bagian 1.3 ke dua dimensi: elemen `catalog2D[s][p]` adalah elemen ke-`(s × SLOTS + p)` dari tata letak 1D yang digulung terbuka tersebut, sehingga alamatnya mengikuti bentuk base-plus-offset yang sama, hanya dengan indeks linear row-major disubstitusikan menggantikan `i`:

```
// alamat(catalog2D[s][p])
//     = base + (s * SLOTS + p) * sizeof(struct Book)

void *base = (void *) &catalog2D[0][0];
long linearIndex = (long) s * SLOTS + p;
void *computed = (void *) ((char *) base + linearIndex * (long) sizeof(struct
Book));
// computed == (void *) &catalog2D[s][p], untuk setiap s, p
```

Row-major adalah pilihan, bukan hukum alam

C/C++ secara spesifik menjamin tata letak row-major untuk array multidimensi — indeks paling kanan berubah paling cepat saat Anda menelusuri memori dalam urutan alamat. Beberapa bahasa lain (Fortran, misalnya) secara default menggunakan column-major, di mana indeks paling kiri yang berubah paling cepat. Rumus di bagian ini spesifik untuk jaminan row-major C/C++; memindahkan kode aritmatika alamat antar bahasa dengan tata letak array yang berbeda adalah sumber bug yang nyata dan mudah terlewat.

1.6 Lampiran 1.A — Log Validasi

Kedua program demo di bawah ini dikompilasi dengan `g++ -Wall -Wextra -std=c++17` (nol peringatan dari keduanya) dan benar-benar dijalankan; transkrip di bawah adalah keluaran terminal yang asli dan tidak diedit — bukan keluaran “yang diharapkan” yang ditulis tangan. Alamat akan berbeda pada setiap eksekusi (OS mengacak alamat stack demi keamanan), tetapi hubungan yang diklaim bab ini —

`&catalog[i] == catalog + i`, dan celah 196-byte yang konstan antar elemen — berlaku pada setiap eksekusi, seperti yang ditunjukkan.

1.6.1 demo_1d.cpp

```
$ g++ -Wall -Wextra -std=c++17 -o demo_1d book.cpp demo_1d.cpp
$ ./demo_1d
=== Pustaka Ceria: 1D Catalog (8 books) ===

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

=== Address Arithmetic: catalog[i] vs *(catalog + i) ===

sizeof(struct Book) = 196 bytes

i   &catalog[i]        catalog + i        same?
0   0x7ffd2397fd60     0x7ffd2397fd60     yes
1   0x7ffd2397fe24     0x7ffd2397fe24     yes
...(all 8 rows: yes)...

=== Byte offset between consecutive elements ===

&catalog[1] - &catalog[0] = 196 bytes (sizeof(Book) = 196) -> matches
...(all 7 gaps: matches)...
```

1.6.2 demo_2d.cpp

```
$ g++ -Wall -Wextra -std=c++17 -o demo_2d book.cpp demo_2d.cpp
$ ./demo_2d
=== Pustaka Ceria: Shelves as a 2D Grid (2 shelves x 4 slots) ===

--- Shelf 0 ---   B1001, B1002, B1003, B1004
--- Shelf 1 ---   B1005, B1006, B1007, B1008

=== Row-Major Address Arithmetic ===

sizeof(struct Book) = 196 bytes, SLOTS = 4
base address (&catalog2D[0][0]) = 0x7ffc7a0d21a0

s    p    &catalog2D[s][p]   base + (s*SLOTS+p)*sizeof(Book) same?
0    0    0x7ffc7a0d21a0     0x7ffc7a0d21a0           yes
0    1    0x7ffc7a0d2264     0x7ffc7a0d2264           yes
...(all 8 rows: yes)...
```

> **Catatan kejujuran**
> Setiap nilai yang ditampilkan di atas (ukuran struct 196-byte, setiap “yes”, setiap offset byte yang cocok) berasal langsung dari kompilasi-dan-eksekusi nyata di lingkungan build bab ini — tidak ada ketidaksesuaian atau bug yang ditemukan pada kode bab ini sendiri, dan tidak ada yang direkayasa di sini demi efek; kebijakan tetap mata kuliah ini adalah melaporkan hasil validasi persis seperti apa adanya, baik itu mencakup bug maupun tidak.

1.7 Ringkasan Bab dan Poin-Poin Kunci

- Struktur Data mempelajari bagaimana menata data di memori sehingga operasi yang dibutuhkan program (sisip, hapus, cari, urutkan, telusuri) tetap cepat seiring bertambahnya data; ke-16 bab mata kuliah ini membangun sebuah kotak perkakas tata letak — array, list, stack/queue, tree, graph, dan tabel hash — persis untuk tujuan tersebut.
- Pustaka Ceria, sebuah perpustakaan kampus kecil yang perlu menyimpan, mencari, mengurutkan, dan mengorganisasikan katalog bukunya, adalah studi kasus yang berjalan di mata kuliah ini dari Bab 1 hingga Bab 15; `struct Book` (id/title/author/category) adalah tipe rekaman fondasinya.
- Setiap variabel memiliki nilai dan alamat; `&x` memberi Anda alamat `x`, dan pointer hanyalah sebuah variabel yang nilainya berupa alamat. Bab ini hanya memberi pratinjau sintaks pointer secukupnya untuk membaca aritmatika alamat — cakupan pointer penuh ada di Bab 6.
- Array dari sebuah struct disusun berurutan di memori tanpa celah: `alamat(catalog[i]) = base + i × sizeof(struct Book)`, dan `catalog[i]` serta `*(catalog + i)` selalu persis alamat yang sama.
- Array 2D disusun secara row-major: seluruh baris pertama disimpan sebelum baris kedua dimulai, sehingga `alamat(catalog2D[s][p]) = base + (s × SLOTS + p) × sizeof(struct Book)`.

1.8 Pertanyaan Tinjauan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Bab 2 (Pencarian).

1. Dengan kata-kata Anda sendiri, jelaskan mengapa pilihan struktur data dapat memengaruhi kinerja sebuah program bahkan ketika struktur data tersebut menyimpan persis informasi yang sama. Berikan satu contoh konkret (tidak harus melibatkan Pustaka Ceria).
2. `struct Book` menggunakan array `char` berukuran tetap untuk setiap field, bukan tipe string berukuran dinamis. Jelaskan apa keuntungan pilihan ini bagi penjelasan tata letak memori bab ini, dan sebutkan satu keterbatasan dari merepresentasikan judul dengan cara ini (petunjuk: apa yang terjadi pada judul yang lebih panjang dari 99 karakter?).
3. Diberikan `struct Book catalog[10];`, tuliskan ekspresi (menggunakan aritmatika pointer, bukan pengindeksan) yang merujuk ke elemen yang sama dengan `catalog[6]`, dan jelaskan dalam satu kalimat mengapa kedua ekspresi tersebut dijamin merupakan alamat yang sama.

4. Andaikan `struct Book` mendapat field kelima, `int yearPublished;`, disisipkan di antara `id` dan `title`. Apakah Anda mengharapkan `sizeof(struct Book)` masih sama dengan jumlah sederhana ukuran field-fieldnya? Jelaskan mengapa atau mengapa tidak, dengan merujuk pada alignment.
5. Untuk `struct Book catalog2D[3][5];` (3 rak, 5 slot), turunkan rumus aritmatika alamat untuk `catalog2D[2][3]` dalam bentuk alamat basis dan `sizeof(struct Book)`, mengikuti penalaran row-major yang sama seperti Bagian 1.5.
6. Pustakawan Pustaka Ceria ingin menambahkan buku kesembilan ke katalog 1D beranggota 8 buku dari Bagian 1.4 tanpa mengubah apa pun tentang bagaimana array tersebut dideklarasikan. Jelaskan masalah praktis apa yang muncul, dan sebutkan (tanpa mengimplementasikannya) fitur C/C++ yang diperkenalkan Bab 6 untuk menyelesaikan persis masalah semacam ini.

Referensi

[1] B.R. Preiss. “Data Structures and Algorithms with Object-Oriented Design Patterns in C++.” John Wiley & Sons, 1998.

[2] P.J. Deitel, H.M. Deitel. “C++ How to Program.” 10th ed., Pearson, 2017.

[3] cppreference.com. “Array declaration.” https://en.cppreference.com/w/cpp/language/array (diakses Agustus 2026).

[4] cppreference.com. “Object sizes, alignment, and padding.” https://en.cppreference.com/w/cpp/language/object (diakses Agustus 2026).

[5] cppreference.com. “Pointer declaration.” https://en.cppreference.com/w/cpp/language/pointer (diakses Agustus 2026).

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 2

Pencarian

Empat cara menjawab pertanyaan yang cepat atau lambat akan diajukan ke setiap katalog: "apakah buku ini ada di sini, dan jika ya, di mana?" Setiap baris kode di bab ini benar-benar dikompilasi dengan g++ -Wall -Wextra dan dijalankan — transkrip terminal aslinya direproduksi di Lampiran 2.A.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Menjelaskan makna kontrak kembalian "ditemukan / tidak ditemukan" pada algoritma pencarian, dan mengapa mengembalikan posisi (sebuah indeks), bukan hanya jawaban ya/tidak, hampir selalu merupakan kontrak yang lebih berguna. (2) Mengimplementasikan dan menelusuri sequential (linear) search pada katalog Pustaka Ceria, berdasarkan id maupun judul, serta menyatakan biaya kasus terburuknya. (3) Menjelaskan sentinel search sebagai optimisasi mikro dari sequential search, dan secara tepat apa yang dihemat dan tidak dihemat olehnya. (4) Mengimplementasikan binary search pada katalog yang sudah terurut berdasarkan id, menelusuri penyempitan low/mid/high-nya secara manual, dan menjelaskan mengapa keterurutan adalah prasyarat sungguhan, bukan sekadar detail implementasi. (5) Mengimplementasikan interpolation search pada kunci numerik terurut, dan menjelaskan mengapa algoritma ini kurang cocok untuk id buku Pustaka Ceria yang bertipe string. (6) Menyatakan definisi formal notasi Big-O, dan menggunakannya untuk membandingkan perilaku kasus terbaik, rata-rata, dan terburuk keempat algoritma.

2.1 Mengapa Pencarian Penting: Kontrak Ditemukan / Tidak Ditemukan

Bab 1 memberi katalog Pustaka Ceria sebuah bentuk di memori — sebuah `struct Book` dan array berurutan untuk menampung beberapa di antaranya. Namun, hal pertama yang benar-benar dilakukan siapa pun terhadap sebuah katalog bukanlah mengagumi tata letak memorinya; melainkan mengajukan pertanyaan: "apakah ada buku dengan id `B1006` di sini, dan jika ya, yang mana?" Pertanyaan itu, yang diajukan berulang-ulang, di setiap sistem yang menyimpan koleksi apa pun, adalah yang disebut bab ini sebagai pencarian (searching).

Setiap algoritma pencarian di bab ini menjawab persis pertanyaan dua-bagian yang sama, dalam bentuk yang persis sama, sehingga algoritma di baliknya bisa diganti nanti tanpa mengubah apa pun yang memanggilnya: diberikan sebuah koleksi dan sebuah kunci untuk dicari, kembalikan posisi (indeks) elemen yang cocok jika ada, atau sinyal "tidak ditemukan" yang jelas dan dapat dibedakan jika tidak ada. Kode bab ini mengembalikan `-1` untuk "tidak ditemukan" — sebuah nilai yang tidak pernah bisa sama dengan indeks array yang valid mana pun — alih-alih mengembalikan `true`/`false` biasa. Sebuah `bool` hanya bisa memberi tahu pemanggil bahwa sebuah buku ada di suatu tempat; sebuah indeks memberi tahu pemanggil persis elemen yang mana, sehingga baris kode berikutnya bisa langsung mencetak judulnya, mengubah kategorinya, atau menyerahkannya ke fungsi lain — tanpa perlu pencarian kedua untuk menemukan lokasinya kembali.

> **"Ditemukan atau tidak" vs. "ditemukan di mana"**
> Tergoda rasanya untuk menulis fungsi pencarian yang mengembalikan `bool` — terbaca bersih di titik pemanggilan (`if (bookExists(catalog, N, "B1006")) { ... }`). Tetapi hampir setiap pemanggil sungguhan langkah berikutnya adalah "...dan sekarang lakukan sesuatu dengan buku itu," yang membutuhkan indeks atau alamatnya, bukan sekadar fakta bahwa ia ada. Setiap fungsi di `search.h` bab ini mengembalikan indeks `int` (atau `-1`), persis karena alasan ini — itu adalah kontrak yang secara ketat lebih berguna, tanpa biaya tambahan untuk menuliskannya.

Bab ini membahas empat algoritma yang semuanya menjawab kontrak ditemukan/posisi yang sama tersebut, masing-masing menukar asumsi berbeda tentang data dengan jumlah kecepatan yang berbeda: sequential search (Bagian 2.2) tidak mengasumsikan apa pun tentang urutan katalog dan selalu tersedia sebagai cadangan; sentinel search (Bagian 2.3) adalah optimisasi kecil dan jujur atas loop internal sequential search; binary search (Bagian 2.4) mengasumsikan katalog sudah terurut dan jauh lebih cepat sebagai gantinya; dan interpolation search (Bagian 2.5) mengasumsikan kunci pengurutannya numerik dan tersebar cukup merata, dan lebih cepat lagi ketika asumsi itu benar-benar terpenuhi. Bagian 2.6 memperkenalkan notasi Big-O secara formal sehingga keempatnya bisa dibandingkan pada dasar yang sama.

2.2 Sequential (Linear) Search

Sequential search adalah algoritma yang dipakai kita semua tanpa pernah diajarkan secara eksplisit: untuk menemukan sesuatu dalam tumpukan yang tidak terurut, lihat item pertama, lalu kedua, lalu ketiga, dan seterusnya, sampai Anda menemukannya atau kehabisan item. Algoritma ini sama sekali tidak mengasumsikan bagaimana katalog itu disusun — persis karena itulah ia menjadi algoritma pertama yang dibahas bab ini, dan yang dijadikan pembanding setiap algoritma lain di bab ini.

```
int sequentialSearchById(const Book catalog[], int n, const char *id, long
*comparisons) {
    for (int i = 0; i < n; i++) {
        if (comparisons) (*comparisons)++;
        if (std::strcmp(catalog[i].id, id) == 0) {
            return i;
        }
    }
    return -1;
}
```

`sequentialSearchByTitle` memiliki bentuk yang identik, membandingkan `catalog[i].title` alih-alih `catalog[i].id` — algoritma ini tidak peduli field mana yang dibandingkan, hanya bahwa suatu field bisa dibandingkan untuk kesetaraan. Kedua fungsi menerima penghitung `comparisons` opsional sehingga program demo bab ini bisa melaporkan jumlah perbandingan yang benar-benar terukur, bukan perkiraan.

> **Kasus terbaik, kasus terburuk, dan mengapa kasus terburuk yang penting**
> Jika target adalah `catalog[0]`, sequential search menemukannya dalam persis 1 perbandingan — kasus terbaiknya. Jika target adalah `catalog[n-1]`, atau sama sekali tidak ada di katalog, sequential search harus melakukan persis `n` perbandingan sebelum bisa menyimpulkan apa pun — kasus terburuknya. Seperti yang diperjelas Bagian 2.6, kasus terburuklah yang dideskripsikan notasi Big-O, karena pemanggil tidak bisa tahu sebelumnya kasus mana yang akan dialami sebuah pencarian tertentu, dan katalog yang bertumbuh dari 8 buku menjadi 8.000 buku akan membuat kasus terburuk 1.000x lebih mahal, entah pencarian tertentu itu kebetulan beruntung atau tidak.

`demo_sequential.cpp` bab ini menjalankan persis penghitungan perbandingan ini untuk empat pencarian nyata pada katalog 8-buku Pustaka Ceria: elemen pertama (`"B1001"`, 1 perbandingan), elemen tengah (`"B1005"`, 5 perbandingan), elemen terakhir (`"B1008"`, 8 perbandingan), dan sebuah id yang sama sekali tidak ada di katalog (`"B1099"`, 8 perbandingan) — keluaran nyata, terkompilasi, dan dijalankan direproduksi di Lampiran 2.A.

2.3 Sentinel Search: Menghilangkan Pemeriksaan Batas

Sentinel search bukanlah algoritma yang berbeda dari sequential search — ia adalah ide “periksa elemen dari depan sampai ditemukan yang cocok” yang sama, dengan satu perubahan kecil dan disengaja pada loop-nya. Kondisi loop sequential search biasa, `i < n`, diperiksa pada setiap iterasi semata-mata untuk melindungi dari pembacaan di luar batas array. Sentinel search menghilangkan pemeriksaan itu sepenuhnya, dengan terlebih dahulu menyalin target itu sendiri ke satu slot ekstra di ujung paling akhir array (indeks `n`, sang sentinel), yang menjamin akan ada elemen yang cocok — sehingga tugas loop yang tersisa hanyalah memeriksa kecocokan, tidak pernah perlu memeriksa apakah ia sudah kehabisan ruang untuk mencari.

```
int sentinelSearchById(Book catalogWithSentinel[], int n, const char *id, long
*comparisons) {
    // catalogWithSentinel[n] sudah menyimpan salinan target (sentinel),
    // ditanam di sana oleh pemanggil sebelum ini berjalan.
    int i = 0;
    while (true) {
        if (comparisons) (*comparisons)++;
        if (std::strcmp(catalogWithSentinel[i].id, id) == 0) {
            break;
        }
        i++;
    }
    return (i < n) ? i : n;  // i == n berarti hanya sentinel yang cocok ->
bukan hit sungguhan
}
```

Apa yang benar-benar dihemat sentinel search — dan apa yang tidak

Menjalankan pencarian yang persis sama melalui `sequentialSearchById` maupun `sentinelSearchById` (`demo_sentinel.cpp` bab ini melakukan persis ini, berdampingan) menghasilkan jumlah perbandingan yang IDENTIK untuk setiap id yang dicoba — sentinel search tidak melihat lebih sedikit elemen, dan tidak mengubah kasus terburuk O(n) algoritmanya. Yang dihilangkannya adalah pemeriksaan kedua yang terpisah (`i < n`) yang dievaluasi ulang oleh loop sequential search biasa pada setiap iterasi, di samping pemeriksaan kecocokan. Itu adalah optimisasi mikro yang nyata dan jujur — satu pemeriksaan mesin lebih sedikit per iterasi, yang bisa berarti dalam loop ketat pada array yang sangat besar — tetapi itu adalah perbaikan faktor-konstan, bukan perubahan pada perilaku asimtotik algoritmanya. Bab ini melaporkannya persis seperti itu, alih-alih merekayasa benchmark waktu agar perbedaannya terlihat lebih besar dari yang sebenarnya.

Kompromi yang diterima sentinel search demi penghematan ini adalah kompromi nyata yang layak disebutkan: ia butuh array yang bisa ditulis, satu elemen lebih besar dari data itu sendiri (untuk menampung sentinel), dan ia merusak apa pun yang ada di indeks `n` sebelum pencarian dimulai. Untuk pencarian baca-saja katalog Pustaka Ceria, kompromi itu mudah diambil (demo bab ini cukup bekerja pada salinan sekali-pakai); untuk katalog yang sedang Anda ubah secara bersamaan, ini butuh kehati-hatian lebih.

2.4 Binary Search: Membagi Dua Masalah Setiap Langkah

Binary search adalah algoritma pertama di bab ini yang bukan variasi dari “periksa setiap elemen” — sebaliknya, ia memanfaatkan urutan. Jika katalog terurut berdasarkan id, maka membandingkan id target dengan id yang berada persis di tengah array memberi tahu Anda, dalam satu perbandingan, SETENGAH mana dari array yang mungkin berisi target — setengah lainnya bisa dibuang sepenuhnya, tanpa pernah melihat satu pun elemen di dalamnya. Mengulangi ini — lihat tengah dari yang tersisa, buang setengahnya lagi — menyempitkan jendela pencarian hingga habis (atau hingga cocok) dalam sejumlah langkah yang hanya bertumbuh secara logaritmik seiring ukuran katalog, bukan secara linear.

Prasyarat nyata binary search: array harus sudah terurut

Ini bukan detail implementasi kecil — ini adalah keseluruhan alasan mengapa binary search bekerja sama sekali. Keterurutan adalah yang membuat “id target lebih kecil dari id elemen tengah” menjadi sinyal satu-perbandingan yang dapat diandalkan bahwa seluruh separuh kanan array bisa diabaikan; pada array yang tidak terurut, kesimpulan itu sekadar salah, dan binary search akan diam-diam mengembalikan jawaban yang keliru. BAGAIMANA mengurutkan array dari awal (Bubble, Selection, Insertion, Quick, Merge, Heap sort) adalah topik Bab 3 sendiri, bukan bab ini — di sini, pengurutan diperlakukan secara jujur sebagai langkah prasyarat satu baris, menggunakan `std::sort` milik pustaka standar C++, persis seperti cara seorang programmer sungguhan meraihnya sebelum menulis ulang sebuah algoritma pengurutan dari nol hanya untuk memenuhi satu fungsi ini.

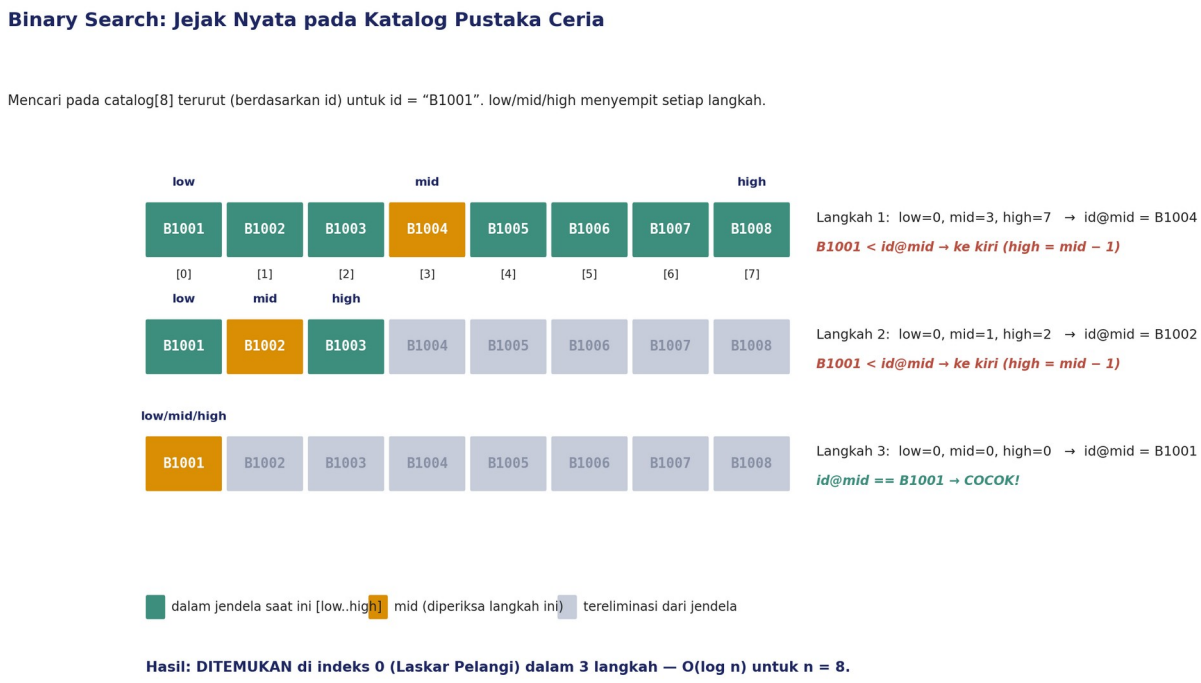
`demo_binary.cpp` membuat ketergantungan ini eksplisit, bukan menyembunyikannya: ia mulai dari katalog Pustaka Ceria dalam urutan buku-buku itu benar-benar tiba (yang tidak ada hubungannya dengan nomor id mereka), mengurutkan salinannya berdasarkan id dengan `std::sort`, dan baru kemudian menjalankan binary search.

```
std::sort(sorted, sorted + N, [](const Book &a, const Book &b) {
    return std::strcmp(a.id, b.id) < 0;
});
// CATATAN: std::sort adalah panggilan pustaka, bukan algoritma
// pengurutan buatan tangan -- Bab 3 membahas Bubble/Selection/
// Insertion/Quick/Merge/Heap sort dari nol. Di sini ia hanyalah
// langkah prasyarat jujur yang benar-benar dibutuhkan binary search.
```

Dengan array yang benar-benar terurut, binary search sendiri adalah loop pendek yang menjaga jendela `[low, high]` yang menyusut dan selalu memeriksa titik tengahnya:

```
int binarySearchById(const Book catalog[], int n, const char *id, ...) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        int cmp = std::strcmp(catalog[mid].id, id);
        if (cmp == 0) return mid;              // cocok!
        else if (cmp < 0) low = mid + 1;       // target ada di kanan
        else high = mid - 1;                   // target ada di kiri
    }
    return -1;   // low > high -- jendela kosong, target tidak ada
}
```

Gambar 2.1 menelusuri loop ini pada katalog Pustaka Ceria yang terurut untuk sebuah pencarian nyata, `id = "B1001"`, menggunakan keluaran sungguhan yang dicetak `demo_binary.cpp` — bukan contoh yang disimulasikan secara manual.



Gambar 2.1 — Penyempitan jendela low/mid/high binary search pada katalog Pustaka Ceria yang terurut, menelusuri pencarian nyata untuk id = “B1001”.

mid = low + (high - low) / 2, bukan (low + high) / 2

Kedua rumus menghitung titik tengah yang sama secara matematis, tetapi `low + (high - low) / 2` adalah versi yang layak ditulis sebagai kebiasaan: untuk nilai `low` dan `high` yang sangat besar (mendekati batas yang bisa ditampung sebuah `int`), `low + high` bisa diam-diam overflow sebelum pembagiannya bahkan terjadi, sementara `low + (high - low)` tidak bisa, karena `high - low` selalu kecil dan tidak negatif dalam jendela pencarian yang valid. Ini kebiasaan defensif yang kecil, bukan sesuatu yang akan pernah dipicu oleh katalog 8-elemen bab ini — tetapi ini versi yang layak dipelajari dengan benar sejak awal.

Perhatikan, pada Gambar 2.1, betapa sedikitnya langkah yang dibutuhkan binary search: 3 langkah untuk menemukan `id = "B1001"` di antara 8 buku, sementara kasus terburuk sequential search pada 8 buku yang sama membutuhkan hingga 8 perbandingan. Kesenjangan itu hanya melebar seiring bertambah besarnya katalog — Bagian 2.6 memperjelas persis seberapa besar.

2.5 Interpolation Search: Menebak Lebih Cerdas daripada Titik Tengah

Binary search selalu memeriksa persis titik tengah dari jendela mana pun yang tersisa, terlepas dari seperti apa nilai-nilainya sebenarnya. Interpolation search menyempurnakan satu keputusan itu: alih-alih selalu menebak titik tengah, ia mengestimasi DI MANA dalam jendela target kemungkinan berada, berdasarkan nilai target relatif terhadap nilai-nilai di kedua ujung jendela — mirip seperti seseorang yang mencari nama di buku telepon membuka lebih dekat ke depan buku untuk nama berawalan “B” dan lebih dekat ke belakang untuk yang berawalan “T”, alih-alih selalu membuka persis di tengah terlebih dahulu.

```
// Estimasi posisi probe dengan asumsi nilai tersebar cukup merata
// antara arr[low] dan arr[high] -- alih-alih mid tetap milik
// binary search, low + (high - low) / 2:
int pos = low + (int)(((double)(high - low) * (key - arr[low]))
                      / (arr[high] - arr[low]));
```

Mengapa id buku Pustaka Ceria adalah kunci yang salah untuk algoritma ini

Seluruh estimasi interpolation search bertumpu pada kemampuan bertanya “kira-kira seberapa jauh `key` dari `arr[low]`, sebagai pecahan dari jarak `arr[low]` ke `arr[high]`?” — pertanyaan yang hanya masuk akal untuk kunci NUMERIK, di mana pengurangan dan pembagian bermakna. Id buku seperti `"B1001"` dan `"B1008"` adalah string; `strcmp` bisa memberi tahu URUTAN-nya, tetapi “seberapa jauh” dua string satu sama lain tidak punya jawaban numerik alami sebagaimana halnya dua bilangan bulat. Menerapkan rumus interpolasi pada id string bahkan tidak akan terkompilasi tanpa pengkodean numerik ad-hoc, dan pengkodean semacam itu akan sembarangan, bukan bermakna — jadi demo interpolation search bab ini dengan sengaja beralih ke kunci yang benar-benar numerik: TAHUN TERBIT setiap buku, disimpan dalam array `int` terpisah yang paralel dengan (tetapi tidak disimpan di dalam) `struct Book`. `struct Book` itu sendiri tidak berubah dari Bab 1.

`demo_interpolation.cpp` mencari array terurut dari tahun terbit 8 buku Pustaka Ceria yang sama (1975 hingga 2011) untuk sebuah kunci nyata, `2005`, yang kebetulan berada dekat ujung atas rentang

tersebut. Karena nilai-nilainya tersebar cukup merata, probe pertama interpolation search mendarat dekat posisi yang benar, berbeda dengan titik tengah tetap milik binary search — jejak nyata dan tercetak dari algoritma ini (direproduksi di Lampiran 2.A) menemukannya dalam 2 langkah.

Interpolation search juga harus menangani kasus yang tidak pernah perlu dipertimbangkan binary search: bagaimana jika nilai low dan high jendela saat ini sama? Membagi dengan `arr[high] - arr[low]` maka akan membagi dengan nol. Implementasi bab ini memeriksa kasus itu secara eksplisit dan kembali ke pemeriksaan awal jendela — selalu aman, meski tidak selalu “cerdas”:

```
if (arr[high] == arr[low]) {
    pos = low;   // hindari pembagian-dengan-nol; tetap benar, hanya tidak
"cerdas"
} else {
    pos = low + (int)(((double)(high - low) * (key - arr[low])) / (arr[high] -
arr[low]));
}
```

2.6 Membandingkan Empat Algoritma: Memperkenalkan Notasi Big-O

Setiap bagian sejauh ini telah mendeskripsikan biaya sebuah algoritma secara informal — “n perbandingan dalam kasus terburuk,” “sejumlah langkah yang hanya bertumbuh secara logaritmik.” Notasi Big-O adalah bahasa formal yang digunakan ilmu komputer untuk membuat pernyataan semacam itu presisi dan dapat dibandingkan, dan mata kuliah ini memperkenalkannya di sini, pada bab pertamanya tentang algoritma alih-alih tata letak data, karena pencarian adalah tempat pertama perbedaan laju pertumbuhan menjadi kekhawatiran yang benar-benar praktis.

> **Notasi Big-O, secara formal**
> Notasi Big-O mendeskripsikan bagaimana waktu eksekusi kasus terburuk sebuah algoritma (atau jumlah operasi dasar, seperti perbandingan) bertumbuh seiring ukuran input `n` bertambah, MENGABAIKAN faktor konstan dan suku berorde lebih rendah. Sebuah algoritma adalah O(f(n)) jika, untuk `n` yang cukup besar, waktu eksekusinya dibatasi dari atas oleh suatu kelipatan konstan dari f(n). O(1) ("konstan") berarti biayanya tidak bertumbuh sama sekali seiring `n`. O(log n) ("logaritmik") berarti biayanya bertumbuh sangat lambat — menggandakan `n` hanya menambah satu langkah lagi. O(n) ("linear") berarti biayanya bertumbuh berbanding lurus dengan `n`. Big-O menjawab pertanyaan “apa yang terjadi saat `n` menjadi besar?” — ia dengan sengaja diam soal jumlah persis operasi yang dibutuhkan satu eksekusi tertentu, atau algoritma mana yang lebih cepat untuk suatu `n` kecil tertentu; pertanyaan-pertanyaan itu juga penting, tetapi Big-O bukan alat untuk itu.

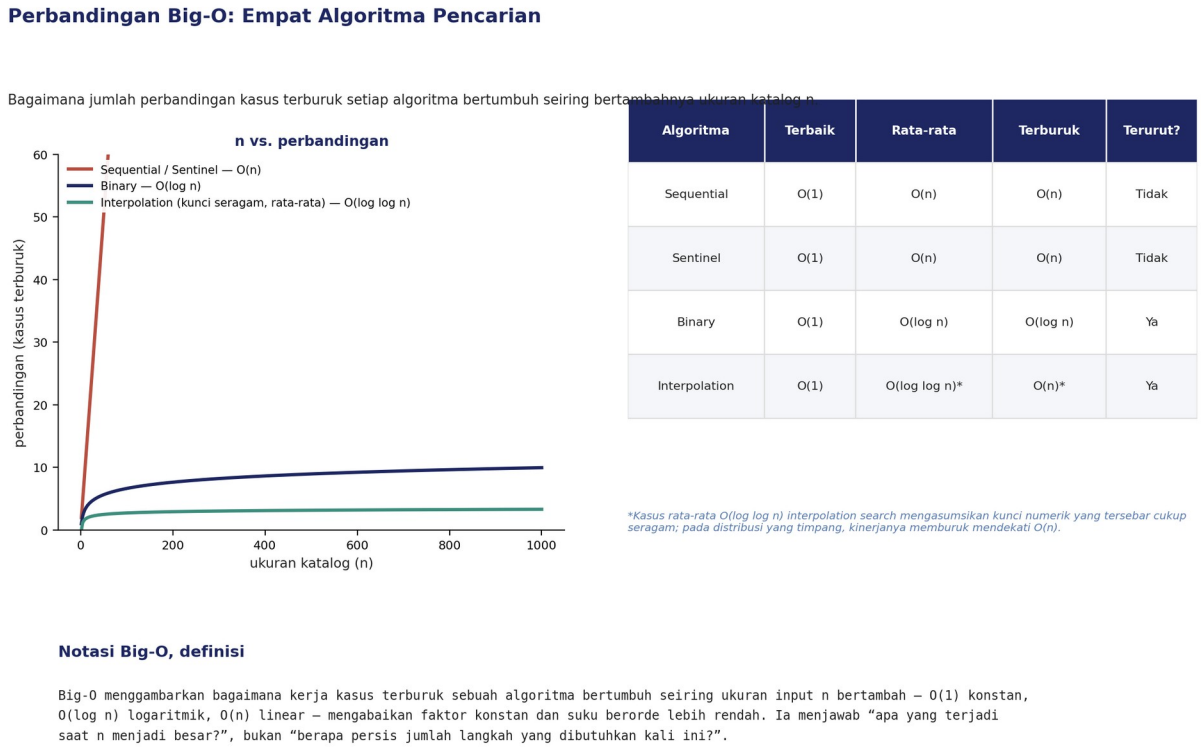
Dengan definisi itu di tangan, berikut biaya sesungguhnya keempat algoritma bab ini, dalam kasus terburuk, seiring katalog Pustaka Ceria bertumbuh dari 8 buku menjadi 8.000 menjadi 8 juta:

Algoritma	Kasus terbaik	Kasus rata-rata	Kasus terburuk	Perlu input terurut?
Sequential search	O(1)	O(n)	O(n)	Tidak

Sentinel search	O(1)	O(n)	O(n)	Tidak
Binary search	O(1)	O(log n)	O(log n)	Ya
Interpolation search	O(1)	O(log log n)*	O(n)*	Ya

**Kasus rata-rata O(log log n) interpolation search mengasumsikan kunci numerik yang tersebar cukup seragam, sebagaimana contoh tahun-terbit Bagian 2.5 memang sengaja demikian; pada distribusi kunci yang timpang parah, kasus terburuknya memburuk hingga mencapai O(n) — tidak lebih baik dari sequential search, pada susunan data yang paling buruk yang mungkin.*

Gambar 2.2 menampilkan dua laju pertumbuhan yang paling penting di sini — O(n) dan O(log n) — pada sumbu yang sama, berdampingan dengan kurva O(log log n) milik interpolation search, sehingga kesenjangan di antara keduanya terlihat langsung, bukan hanya dinyatakan dalam tabel.



Gambar 2.2 — Perbandingan laju pertumbuhan Big-O keempat algoritma pencarian, dan ringkasan perilaku kasus terbaik/rata-rata/terburuknya.

Mengapa kolom "kasus rata-rata" tabel ini bukanlah keseluruhan cerita

Katalog 8 buku, seperti yang dipakai demo-demo bab ini sepanjang bab, jauh terlalu kecil bagi perbedaan laju pertumbuhan mana pun ini untuk berarti dalam waktu nyata (wall-clock). Keseluruhan maksud Big-O adalah bahwa perbedaan-perbedaan itu baru menjadi dramatis begitu `n` besar. Sequential search pada 8 elemen dan binary search pada 8 elemen keduanya selesai, bagi manusia, seketika; kedua algoritma yang sama pada 8 juta elemen tidak. Membaca kurva-kurva Gambar 2.2 adalah keseluruhan latihannya — jangan berharap katalog demo kecil bab ini menunjukkan perbedaan kecepatan yang terukur dengan sendirinya.

2.7 Lampiran 2.A — Log Validasi

Keempat program demo di bawah ini dikompilasi dengan `g++ -Wall -Wextra -std=c++17` (nol peringatan dari keempatnya) dan benar-benar dijalankan; transkrip di bawah adalah keluaran terminal yang asli dan tidak diedit — bukan keluaran "yang diharapkan" yang ditulis tangan.

2.7.1 demo_sequential.cpp

```
$ g++ -Wall -Wextra -std=c++17 -o demo_sequential book.cpp search.cpp
demo_sequential.cpp
$ ./demo_sequential
--- Search by id ---
  search id="B1001 " -> FOUND at index 0 (Laskar Pelangi              ) in 1
comparison(s)
  search id="B1005 " -> FOUND at index 4 (Algoritma & Struktur Data   ) in 5
comparison(s)
  search id="B1008 " -> FOUND at index 7 (Negeri 5 Menara             ) in 8
comparison(s)
  search id="B1099 " -> NOT FOUND after 8 comparison(s)

--- Search by title ---
  search title="Fisika Dasar                " -> FOUND at index 6 (id=B1007) in
7 comparison(s)
  search title="Matematika Diskrit          " -> NOT FOUND after 8 comparison(s)
```

2.7.2 demo_sentinel.cpp

```
$ g++ -Wall -Wextra -std=c++17 -o demo_sentinel book.cpp search.cpp
demo_sentinel.cpp
$ ./demo_sentinel
  id="B1001 "  sequential: FOUND     (1 comparisons)   sentinel: FOUND     (1
comparisons)
  id="B1005 "  sequential: FOUND     (5 comparisons)   sentinel: FOUND     (5
comparisons)
  id="B1008 "  sequential: FOUND     (8 comparisons)   sentinel: FOUND     (8
comparisons)
  id="B1099 "  sequential: NOT FOUND (8 comparisons)   sentinel: NOT FOUND (9
comparisons)
```

Perhatikan asimetri jujur pada baris terakhir: untuk id yang benar-benar tidak ada, jumlah sentinel search sendiri (9) adalah SATU LEBIH BANYAK dari sequential search (8) — ia tetap harus melewati kedelapan elemen nyata lalu mengenai sentinel itu sendiri di indeks 8 untuk berhenti. Penghematan nyata sentinel search tidak pernah "lebih sedikit perbandingan"; Bagian 2.3 menjelaskan persis apa yang benar-benar dihematnya.

2.7.3 demo_binary.cpp

```
$ g++ -Wall -Wextra -std=c++17 -o demo_binary book.cpp search.cpp
demo_binary.cpp
$ ./demo_binary
Searching for id="B1006":
  low  mid  high id@mid verdict
  0    3    7    B1004  go right
  4    5    7    B1006  match!
  -> FOUND at index 5 (Cerita Rakyat Nusantara) in 2 step(s)

Searching for id="B1001":
  low  mid  high id@mid verdict
  0    3    7    B1004  go left
  0    1    2    B1002  go left
  0    0    0    B1001  match!
  -> FOUND at index 0 (Laskar Pelangi) in 3 step(s)

Searching for id="B1099":
  low  mid  high id@mid verdict
  0    3    7    B1004  go right
  4    5    7    B1006  go right
  6    6    7    B1007  go right
  7    7    7    B1008  go right
  -> NOT FOUND (low > high) after 4 step(s)
```

2.7.4 demo_interpolation.cpp

```
$ g++ -Wall -Wextra -std=c++17 -o demo_interpolation search.cpp
demo_interpolation.cpp
$ ./demo_interpolation
Searching for publication year 2005:
  low  pos  high year@pos       verdict
  0    5    7    2006           go left
  0    4    4    2005           match!
  -> FOUND at index 4: Laskar Pelangi (id=B1001, year 2005) in 2 step(s)

Searching for publication year 1975:
  low  pos  high year@pos       verdict
  0    0    7    1975           match!
  -> FOUND at index 0: Sejarah Nusantara (id=B1004, year 1975) in 1 step(s)

Searching for publication year 2000:
  low  pos  high year@pos       verdict
  0    4    7    2005           go left
  -> NOT FOUND after 1 step(s)
```

Catatan kejujuran

Setiap hitungan dan setiap langkah yang ditampilkan di atas (kedelapan jumlah perbandingan, +1 sungguhan milik sentinel pada kasus tidak ditemukan, setiap baris low/mid/high dan low/pos/high) berasal langsung dari kompilasi-dan-eksekusi nyata di lingkungan build bab ini — tidak ada ketidaksesuaian atau bug yang ditemukan pada kode bab ini sendiri, dan tidak ada yang direkayasa di sini demi efek; kebijakan tetap mata kuliah ini adalah melaporkan hasil validasi persis seperti apa adanya, baik itu mencakup bug maupun tidak.

2.8 Ringkasan Bab dan Poin-Poin Kunci

- Setiap algoritma pencarian di bab ini menjawab kontrak ditemukan/posisi yang sama: diberikan sebuah kunci, kembalikan indeks elemen yang cocok, atau sinyal “tidak ditemukan” yang jelas dan dapat dibedakan (bab ini memakai -1) — bukan sekadar bool.
- Sequential (linear) search memeriksa elemen dari depan, satu per satu; ia tidak butuh asumsi apa pun tentang urutan, dan berbiaya O(n) dalam kasus terburuk (target tidak ada, atau di posisi paling akhir).
- Sentinel search adalah optimisasi mikro dari sequential search: menanam salinan target di indeks n menghilangkan pemeriksaan batas terpisah pada loop. Jumlah perbandingannya persis sama dengan sequential search biasa — penghematannya adalah satu pemeriksaan mesin lebih sedikit per iterasi, bukan lebih sedikit elemen yang diperiksa.
- Binary search mengharuskan array sudah terurut berdasarkan kunci pencarian; dengan itu, ia membagi dua jendela pencariannya setiap langkah, berbiaya O(log n) dalam kasus terburuk. Menulis ulang algoritma pengurutan dari nol adalah topik Bab 3 sendiri — bab ini memakai std::sort sebagai prasyarat satu-baris yang jujur.
- Interpolation search menyempurnakan titik tengah tetap binary search menjadi posisi yang diestimasi, dengan asumsi kunci NUMERIK yang tersebar cukup seragam; ia kurang cocok untuk kunci bertipe string seperti id buku Pustaka Ceria, itulah sebabnya demo bab ini memakai tahun terbit sebagai gantinya.
- Notasi Big-O mendeskripsikan bagaimana biaya kasus terburuk sebuah algoritma bertumbuh seiring ukuran input n bertambah, mengabaikan faktor konstan: O(1) konstan, O(log n) logaritmik, O(n) linear. Keempat algoritma bab ini membentang dari O(1) kasus terbaik hingga O(n) kasus terburuk, dan kesenjangan antara O(n) dan O(log n) hanya melebar seiring n menjadi besar.

2.9 Pertanyaan Tinjauan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Bab 3 (Pengurutan).

1. Jelaskan, dengan kata-kata Anda sendiri, mengapa mengembalikan indeks (atau -1) adalah kontrak yang lebih berguna untuk sebuah fungsi pencarian daripada mengembalikan bool biasa. Deskripsikan satu situasi konkret di mana bool akan memaksa pemanggil melakukan pekerjaan ekstra yang sebenarnya bisa dihindari.
2. Kasus terbaik sequential search adalah O(1) dan kasus terburuknya O(n). Berikan posisi spesifik target yang menghasilkan masing-masing kasus, dan jelaskan mengapa notasi Big-O didefinisikan dalam bentuk kasus terburuk, bukan kasus terbaik.
3. Demo sentinel search di bab ini menunjukkan jumlah perbandingan yang identik dengan sequential search biasa untuk setiap id yang dicari. Jika sentinel search tidak mengurangi jumlah perbandingan, jelaskan secara tepat apa yang sebenarnya dihematnya, dan dalam keadaan seperti apa penghematan itu benar-benar berarti dalam program sungguhan.

4. Mengapa binary search tidak bisa begitu saja langsung dijalankan pada katalog Pustaka Ceria dalam urutan kedatangannya? Merujuk pada Gambar 2.1, jelaskan apa yang akan salah (secara konkret, perbandingan mana yang akan memberi jawaban yang menyesatkan) jika logika low/mid/high diterapkan pada array yang tidak terurut.
5. Jelaskan, menggunakan arr[low] dan arr[high], mengapa rumus posisi-probe interpolation search tidak bermakna untuk id buku Pustaka Ceria (“B1001”, “B1002”, …) sebagaimana halnya untuk array tahun-terbit yang benar-benar dipakai bab ini.
6. Sebuah katalog bertumbuh dari 8 buku menjadi 8.000 buku (1.000x lebih besar). Menggunakan biaya Big-O dari tabel Bagian 2.6, perkirakan berapa kali LEBIH BANYAK perbandingan yang akan dibutuhkan kasus terburuk sequential search, dan berapa langkah LEBIH BANYAK (bukan berapa kali lebih banyak — berapa langkah tambahan) yang akan dibutuhkan kasus terburuk binary search. Apa yang diberitahukan kontras ini kepada Anda tentang mengapa pilihan algoritma pencarian menjadi lebih penting seiring bertambahnya data?

Referensi

[1] B.R. Preiss. “Data Structures and Algorithms with Object-Oriented Design Patterns in C++.” John Wiley & Sons, 1998.

[2] P.J. Deitel, H.M. Deitel. “C++ How to Program.” 10th ed., Pearson, 2017.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Notasi Big-O, Bab 3.)

[4] cppreference.com. “std::sort.” https://en.cppreference.com/w/cpp/algorithm/sort (diakses Agustus 2026).

[5] cppreference.com. “std::strcmp.” https://en.cppreference.com/w/cpp/string/byte/strcmp (diakses Agustus 2026).

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 3

Pengurutan

Tujuh cara mengurutkan sebuah katalog — persis langkah yang tadi hanya dipinjam dari `std::sort` oleh binary dan interpolation search Bab 2. Setiap baris kode di bab ini benar-benar dikompilasi dengan g++ -Wall -Wextra dan dijalankan — transkrip terminal aslinya direproduksi di Lampiran 3.A.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Menjelaskan mengapa keterurutan adalah prasyarat sungguhan untuk binary dan interpolation search (Bab 2), bukan detail insidental, dan mengulas kembali notasi Big-O (diperkenalkan secara formal di Bab 2) cukup baik untuk dipakai tanpa perlu diturunkan ulang. (2) Mengimplementasikan dan menelusuri Bubble, Exchange, dan Selection sort pada subset buku Pustaka Ceria, serta menjelaskan secara tepat bagaimana POLA perbandingan ketiganya berbeda meski JUMLAH perbandingannya identik. (3) Mengimplementasikan dan menelusuri Insertion sort, serta mendemonstrasikan — dengan eksekusi nyata yang terukur, bukan sekadar klaim — kasus terbaiknya yang sungguhan O(n) pada input yang hampir terurut. (4) Mengimplementasikan Quicksort dengan skema partisi Lomuto yang konsisten, menelusuri partisi rekursifnya, dan menjelaskan secara jujur mengapa kasus terburuknya O(n^2) meski kasus rata-ratanya O(n log n). (5) Mengimplementasikan Merge sort, menelusuri struktur split/merge rekursifnya, dan menjelaskan ruang O(n) yang ditukar demi kasus terburuk yang sama baiknya dengan kasus rata-ratanya. (6) Mengimplementasikan Heap sort berbasis array yang mandiri (build-heap + ekstraksi root berulang) dan menyatakan kompleksitas O(n log n)-nya, tanpa perlu dulu ADT heap lengkap yang dibahas Bab 14. (7) Membandingkan ketujuh algoritma dari segi kompleksitas waktu (terbaik/rata-rata/terburuk), ruang tambahan, dan stabilitas, serta memakai perbandingan itu untuk menjustifikasi mengapa lebih dari satu algoritma pengurutan layak diketahui.

3.1 Ulasan: Mengapa Pengurutan Penting, dan Apa yang Sudah Kita Ketahui

Bab 2 mengajukan pertanyaan sederhana pada katalog Pustaka Ceria — “apakah buku ini ada di sini, dan di mana?” — dan menjawabnya dengan empat cara berbeda. Dua dari empat jawaban itu, binary search dan interpolation search, jauh lebih cepat daripada dua lainnya, tetapi keduanya punya syarat: keduanya hanya bekerja pada katalog yang SUDAH TERURUT. Kode demo Bab 2 sendiri tidak berpura-pura sebaliknya. `demo_binary.cpp` memanggil `std::sort` sebelum memanggil `binarySearchById`, dengan komentar kode yang menjanjikan bahwa “Bab 3 membahas Bubble/Selection/Insertion/Quick/Merge/Heap sort dari nol” — janji yang kini ditepati bab ini.

Jadi dalam satu arti, bab ini adalah yang paling praktis sejauh ini: seluruhnya tentang bagaimana benar-benar menghasilkan urutan terurut yang diasumsikan Bab 2. Bab ini membahas tujuh algoritma secara total — empat yang membandingkan dan menukar elemen langsung di dalam array (Bubble, Exchange, Selection, Insertion), dan tiga yang membagi masalah menjadi bagian lebih kecil terlebih dahulu (Quicksort, Merge sort, Heap sort). Ketujuhnya menyelesaikan persis masalah yang sama —

menyusun ulang sebuah array agar elemen-elemennya muncul dalam urutan menaik berdasarkan suatu kunci — dan bab ini mengurutkan katalog Pustaka Ceria berdasarkan JUDUL sepanjang bab, tugas “rapikan rak buku” paling alami yang benar-benar dimiliki seorang pustakawan.

> **Notasi Big-O — ulasan, bukan perkenalan ulang**
> Bab 2, Bagian 2.6 sudah mendefinisikan notasi Big-O secara formal: ia mendeskripsikan bagaimana biaya kasus terburuk sebuah algoritma bertumbuh seiring ukuran input n bertambah, mengabaikan faktor konstan dan suku berorde lebih rendah — O(1) konstan, O(log n) logaritmik, O(n) linear, dan seterusnya. Bab ini tidak menurunkan ulang definisi itu; ia hanya memakainya, menambahkan dua laju pertumbuhan baru ke kosakata yang sudah dibangun Bab 2: O(n^2) ("kuadratik" — biaya bertumbuh dengan KUADRAT ukuran input; menggandakan n kira-kira mengempatkalikan biaya) dan O(n log n) (bentuk yang sungguh berbeda, jauh lebih baik, yang secara khusus dicapai empat dari tujuh algoritma bab ini, alih-alih O(n^2)).

Satu lagi kosakata yang dibutuhkan bab ini dan tidak dibutuhkan Bab 2: STABILITAS. Sebuah algoritma pengurutan disebut stabil jika dua elemen dengan kunci yang SAMA selalu mempertahankan urutan relatif aslinya setelah diurutkan. Bagi Pustaka Ceria ini penting secara konkret: dua buku bisa berbagi kata pertama yang berdekatan secara judul (contoh kerja bab ini sendiri punya “Filosofi Kopi” dan “Fisika Dasar” — judul berbeda, bukan kunci yang sama, jadi stabilitas tidak berlaku khusus untuk keduanya — tetapi bayangkan sebagai gantinya dua salinan judul yang PERSIS SAMA dari cetakan berbeda) — pengurutan yang stabil menjamin bahwa jika salinan A terdaftar sebelum salinan B sebelum diurutkan, ia tetap begitu sesudahnya. Tabel perbandingan Bagian 3.9 menyatakan algoritma mana dari ketujuh yang dibahas bab ini yang stabil dan mana yang tidak, beserta alasannya.

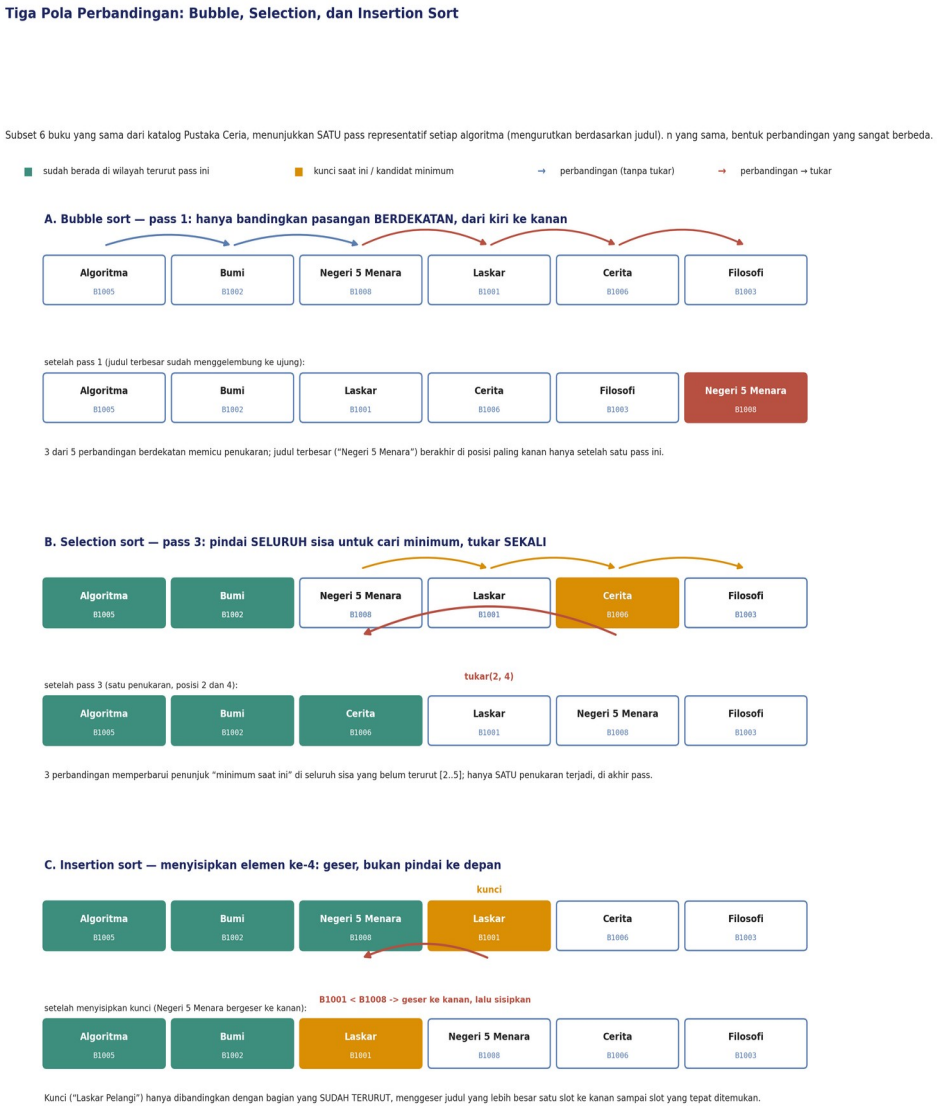
3.2 Bubble Sort

Bubble sort adalah pembacaan paling langsung dari “terus tukar sampai semuanya terurut”: pindai array berulang kali, bandingkan setiap pasangan elemen yang BERDEKATAN, dan tukar jika urutannya salah. Setelah satu pass penuh atas n elemen, elemen terbesar tunggal dijamin sudah “menggelembung” sampai ke ujung array — sehingga pass berikutnya hanya perlu mempertimbangkan n-1 elemen pertama, dan seterusnya.

```
void bubbleSort(Book arr[], int n, long *comparisons, long *swaps) {
    for (int pass = 0; pass < n - 1; pass++) {
        for (int i = 0; i < n - 1 - pass; i++) {
            if (comparisons) (*comparisons)++;
            if (titleLess(arr[i + 1], arr[i])) {
                swapBooks(arr[i], arr[i + 1]);
                if (swaps) (*swaps)++;
            }
        }
    }
}
```

> **Bubble sort bab ini tidak punya flag keluar-dini — dengan sengaja**
> Optimisasi buku teks yang umum menambahkan flag `bool anySwap` yang keluar dari loop luar begitu satu pass penuh tidak melakukan penukaran sama sekali, karena itu berarti array sudah terurut. Implementasi bab ini dengan sengaja menghilangkannya, sehingga kasus TERBAIK Bubble sort sama mahalnya dengan kasus terburuknya: O(n^2) baik dalam kondisi apa pun, selalu menjalankan semua n-1 pass-nya. Ini adalah pilihan desain yang nyata dan jujur, bukan kelalaian — ia ada khusus untuk dikontraskan dengan Insertion sort (Bagian 3.5), yang kasus terbaik O(n)-nya sungguhan dan terukur, bukan sekadar mungkin dicapai dengan menambahkan sebuah flag.

Gambar 3.1, Panel A, menelusuri pass 1 pada subset 6-buku nyata dari katalog Pustaka Ceria (dalam urutan kedatangan normalnya), menunjukkan persis 5 perbandingan berdekatan mana yang memicu penukaran dan mana yang tidak — dan judul terbesar tunggal dari keenamnya, “Negeri 5 Menara”, berakhir di posisi paling kanan hanya setelah satu pass itu.



Gambar 3.1 — Tiga pola perbandingan, ditelusuri pada subset 6-buku yang sama: pemindaian pasangan-berdekatan Bubble sort (Panel A), pindai-lalu-tukar-sekali Selection sort (Panel B), dan geser-prefiks-terurut Insertion sort (Panel C).

Dijalankan pada seluruh katalog 8-buku, `demo_bubble.cpp` bab ini mengukur persis 28 perbandingan (n·(n-1)/2 = 8·7/2 = 28, jumlah tetap untuk setiap pengurutan O(n^2) berbasis pass di bab ini yang tidak melewatkan perbandingan) dan 7 penukaran, berhasil menghasilkan katalog terurut menaik berdasarkan judul — transkrip lengkapnya ada di Lampiran 3.A.

3.3 Exchange Sort

Exchange sort mengajukan pertanyaan "bandingkan dan tukar" yang sama seperti Bubble sort, tetapi dengan POLA perbandingan berbeda: alih-alih hanya membandingkan tetangga berdekatan, ia menetapkan posisi i dan membandingkan `arr[i]` terhadap SETIAP elemen berikutnya `arr[j]` (j > i) secara berurutan, menukar segera begitu judul yang lebih kecil ditemukan di j. Ini berarti `arr[i]` sendiri bisa berubah nilai lebih dari sekali dalam satu pass luar — setiap kali kandidat baru yang lebih kecil ditemukan lebih jauh di array, ia langsung ditukar ke posisi i, dan perbandingan berlanjut dari sana.

```
void exchangeSort(Book arr[], int n, long *comparisons, long *swaps) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (comparisons) (*comparisons)++;
            if (titleLess(arr[j], arr[i])) {
                swapBooks(arr[i], arr[j]);
                if (swaps) (*swaps)++;
            }
        }
    }
}
```

Jumlah perbandingan sama dengan Selection sort — jumlah penukaran sangat berbeda

Exchange sort dan Selection sort (Bagian 3.4) melakukan PERSIS jumlah perbandingan n·(n-1)/2 yang sama, dalam urutan yang persis sama (i tetap, j memindai dari i+1 ke n-1) — `demo_exchange.cpp` dan `demo_selection.cpp` bab ini sama-sama mengukur 28 perbandingan pada katalog 8-buku yang sama. Yang berbeda tajam adalah jumlah PENUKARAN: Exchange sort menukar 7 kali, karena ia menukar begitu menemukan apa pun yang lebih kecil; Selection sort hanya menukar 3 kali, karena ia menunggu menemukan minimum sejati dari elemen yang tersisa sebelum melakukan satu penukaran per pass. Ketika penukaran adalah operasi yang mahal (sebagaimana untuk record besar, meski tidak terlalu untuk `struct Book` yang kecil), perbedaan itu penting — Selection sort meminimalkan penukaran, Exchange sort tidak.

Kedua algoritma berbiaya O(n^2) dalam setiap kasus (terbaik, rata-rata, dan terburuk) — loop ganda selalu menjalankan jumlah iterasi yang sama terlepas dari urutan awal input, karena tidak ada apa pun di sini yang pernah memotong-jalan berdasarkan seberapa terurut array itu sudah.

3.4 Selection Sort

Selection sort menyusun ulang ide yang sama sekali lagi: untuk setiap posisi i, PINDAI dulu seluruh sisa yang belum terurut (`arr[i..n-1]`) untuk menemukan indeks judul minimumnya, dan baru kemudian lakukan satu penukaran — `arr[i]` dengan minimum itu — lalu lanjut ke posisi i+1. Pemindaian dan

penukaran dipisahkan dengan bersih menjadi dua langkah, alih-alih diselang-seling sebagaimana Exchange sort menyelang-selingkannya.

```
void selectionSort(Book arr[], int n, long *comparisons, long *swaps) {
    for (int i = 0; i < n - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < n; j++) {
            if (comparisons) (*comparisons)++;
            if (titleLess(arr[j], arr[minIdx])) {
                minIdx = j;
            }
        }
        if (minIdx != i) {
            swapBooks(arr[i], arr[minIdx]);
            if (swaps) (*swaps)++;
        }
    }
}
```

Gambar 3.1, Panel B, menelusuri pass ketiga Selection sort pada subset 6-buku yang sama dipakai Panel A. Posisi 0 dan 1 sudah mantap dari pass-pass sebelumnya (ditampilkan teal); pass memindai posisi 2 sampai 5, memperbarui penunjuk “minimum saat ini” yang berjalan sebanyak tiga kali (tiga perbandingan, ditampilkan sebagai busur emas), dan selesai dengan tepat SATU penukaran — antara posisi 2 dan 4 — begitu minimum sejati dari sisa (“Cerita Rakyat Nusantara”) diketahui.

Dijalankan pada seluruh katalog 8-buku, `demo_selection.cpp` mengukur 28 perbandingan yang sama dengan Bubble dan Exchange sort, tetapi hanya 3 penukaran — paling banyak n-1 = 7 penukaran yang bisa terjadi (satu per pass luar, dan hanya ketika `minIdx != i`), dan pada katalog khusus ini beberapa pass minimumnya kebetulan sudah berada di posisi i, tidak butuh penukaran sama sekali. Selection sort berbiaya O(n^2) dalam setiap kasus, persis seperti Bubble dan Exchange sort — langkah pemindaian selalu memeriksa setiap elemen yang tersisa, terlepas dari seberapa terurut array itu sudah.

3.5 Insertion Sort

Insertion sort mengambil pendekatan yang sungguh berbeda dari ketiga algoritma di atas: alih-alih berulang kali mencari nilai ekstrem di SELURUH sisa array, ia menumbuhkan wilayah terurut di sisi KIRI satu elemen setiap kalinya. Pada langkah i, elemen `arr[i]` (sang “kunci”) disisihkan sementara, dan setiap elemen di prefiks yang sudah terurut `arr[0..i-1]` yang lebih besar dari kunci digeser satu slot ke kanan, sampai tempat istirahat yang tepat bagi kunci ditemukan dan ia diletakkan di sana.

```
void insertionSort(Book arr[], int n, long *comparisons, long *shifts) {
    for (int i = 1; i < n; i++) {
        Book key = arr[i];
        int j = i - 1;
        while (j >= 0) {
            if (comparisons) (*comparisons)++;
            if (!titleLess(key, arr[j])) break;  // arr[j] <= key: slot
ditemukan
            arr[j + 1] = arr[j];
            if (shifts) (*shifts)++;
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Gambar 3.1, Panel C, menelusuri penyisipan elemen ke-4 (“Laskar Pelangi”, sang kunci) ke dalam prefiks 6-buku yang sebagian sudah terurut. Kunci dibandingkan HANYA dengan elemen terakhir prefiks yang terurut (“Negeri 5 Menara”) — tidak pernah dengan elemen yang belum diproses di sebelah kanannya, persis yang membedakan pola perbandingan Insertion sort dari ketiga algoritma di atas. Karena mendapati kunci lebih kecil, elemen terakhir prefiks bergeser satu slot ke kanan, dan kunci jatuh ke slotnya yang kini kosong.

> **Kasus terbaik O(n) yang sungguhan, TERUKUR — bukan sekadar klaim**
> `demo_insertion.cpp` bab ini menjalankan Insertion sort DUA KALI pada input yang benar-benar berbeda dengan ukuran sama, untuk mengonkretkan kasus terbaik yang adaptif: Run 1 mengurutkan katalog dalam urutan kedatangan normalnya (14 perbandingan, 7 geseran); Run 2 mengurutkan 8 buku yang SAMA untuk kedua kalinya, tetapi kali ini inputnya sudah terurut (keluaran Run 1). Run 2 mengukur persis 7 perbandingan — n-1, minimum teoretis — dan 0 geseran. Pada input yang sudah atau hampir terurut, `break` loop `while` internal terpicu hampir seketika setiap kalinya, sehingga total pekerjaannya sungguhan linear dalam n, bukan kuadratik. Tidak ada algoritma lain di bab ini (kecuali, secara jujur, Bubble sort JIKA ia mempertahankan flag keluar-dini) yang beradaptasi sebersih ini dengan urutan input yang sudah ada — dan Bubble sort bab ini, ingat, dengan sengaja tidak melakukannya.

Kasus terburuk Insertion sort — input menurun, di mana setiap kunci baru harus bergeser melewati SELURUH prefiks yang terurut — adalah O(n^2), sama dengan Bubble, Exchange, dan Selection sort. Kasus terbaiknyalah yang membedakannya: O(n), sungguhan lebih cepat satu orde pertumbuhan penuh, dan alasan praktis nyata mengapa programmer sungguhan meraih Insertion sort secara khusus ketika mereka tahu atau menduga data mereka sudah mendekati terurut (menambahkan beberapa record baru ke katalog yang sebagian besar sudah terurut, misalnya).

3.6 Quicksort

Keempat algoritma di atas semuanya termasuk keluarga yang sama: membandingkan elemen langsung di dalam satu array, dengan biaya O(n^2) pada kasus terburuk (dan, untuk tiga dari empat, di setiap kasus). Quicksort adalah yang pertama dari tiga algoritma bab ini yang justru MEMBAGI masalah: pilih sebuah elemen pivot, PARTISI array sehingga semua yang lebih kecil dari pivot berakhir di sebelah kirinya dan semua yang lebih besar berakhir di sebelah kanannya — yang sekaligus

menempatkan pivot pada posisi akhirnya yang benar — lalu urutkan secara rekursif kedua sub-array di kedua sisinya.

Bab ini memakai skema partisi Lomuto, secara konsisten memilih elemen TERAKHIR dari setiap sub-rentang sebagai pivot:

```
static int lomutoPartition(Book arr[], int low, int high,
                           long *comparisons, long *swaps) {
    const Book &pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (comparisons) (*comparisons)++;
        if (titleLess(arr[j], pivot)) {
            i++;
            if (i != j) { swapBooks(arr[i], arr[j]); if (swaps) (*swaps)++; }
        }
    }
    if (i + 1 != high) { swapBooks(arr[i + 1], arr[high]); if (swaps) (*swaps)+
+; }
    return i + 1;  // posisi akhir pivot sendiri
}

static void quickSortRec(Book arr[], int low, int high,
                         long *comparisons, long *swaps) {
    if (low < high) {
        int p = lomutoPartition(arr, low, high, comparisons, swaps);
        quickSortRec(arr, low, p - 1, comparisons, swaps);
        quickSortRec(arr, p + 1, high, comparisons, swaps);
    }
}
```

`demo_quicksort.cpp` bab ini menjalankan persis kode ini pada katalog 8-buku dengan penelusuran aktif, dan rekursinya sungguh terurai sebagai: partisi rentang penuh [0..7] di sekitar pivot “Sejarah Nusantara” (elemen terakhir array urutan-kedatangan); lalu partisi [0..6] di sekitar “Fisika Dasar”; lalu sisi kiri terpecah menjadi [0..3] di sekitar “Filosofi Kopi”, lalu [0..2] di sekitar “Cerita Rakyat Nusantara”, lalu [0..1] di sekitar “Bumi Manusia” — dan sisi kanan, [5..6], mempartisi di sekitar “Negeri 5 Menara”. Seluruh pengurutan selesai dalam 20 perbandingan dan 3 penukaran, berhasil menghasilkan katalog terurut berdasarkan judul (transkrip lengkap di Lampiran 3.A).

Rata-rata O(n log n), tetapi kasus terburuk sungguhan O(n^2) — dan inilah alasannya persis
Perilaku kasus rata-rata Quicksort sangat baik: setiap langkah partisi diharapkan membagi array menjadi dua bagian yang kira-kira sama besar, memberikan O(log n) tingkat rekursi dengan O(n) pekerjaan per tingkat, jadi O(n log n) secara keseluruhan. Tetapi implementasi bab ini SELALU memilih elemen terakhir sebagai pivot — dan pada input yang SUDAH terurut (atau terurut terbalik), itu secara konsisten memilih elemen terbesar (atau terkecil) yang tersisa sebagai pivot, setiap kalinya. Setiap partisi kemudian membagi array seketimpang mungkin — satu sisi kosong, sisi lain n-1 elemen — memberikan n tingkat rekursi alih-alih log n, sehingga sungguhan O(n^2) pada kasus terburuk. Ini bukan kasus tepi hipotetis yang direka untuk buku ini: ini persis input yang diurutkan `demo_binary.cpp` Bab 2 sebelum mencari, dan persis mengapa pustaka pengurutan produksi entah mengacak pilihan pivot atau beralih ke algoritma lain (sering Insertion sort, karena kecepatannya pada input kecil) di bawah ambang ukuran tertentu — penyempurnaan yang tidak dicoba versi bab ini yang dibangun dari nol, demi satu skema partisi yang tunggal, konsisten, dan dibahas secara jujur.

3.7 Merge Sort

Merge sort membagi array dengan cara paling lugas dari ketiga algoritma bagi-dan-taklukkan di bab ini: bagi persis dua, urutkan masing-masing separuh secara rekursif, dan GABUNGKAN (merge) kedua separuh yang terurut kembali dalam waktu linear. Rekursi berhenti pada sub-array berukuran 1, yang sepele “sudah terurut” — semua pekerjaan sesungguhnya terjadi pada langkah merge, membandingkan elemen depan kedua separuh terurut satu pasang sekaligus dan mengambil yang lebih kecil.

```
static void merge(Book arr[], int low, int mid, int high,
                  long *comparisons, long *moves) {
    int leftN = mid - low + 1, rightN = high - mid;
    Book *left = new Book[leftN], *right = new Book[rightN];
    for (int i = 0; i < leftN; i++)  left[i]  = arr[low + i];
    for (int i = 0; i < rightN; i++) right[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = low;
    while (i < leftN && j < rightN) {
        if (comparisons) (*comparisons)++;
        if (!titleLess(right[j], left[i])) arr[k++] = left[i++];
        else                               arr[k++] = right[j++];
        if (moves) (*moves)++;
    }
    while (i < leftN)  { arr[k++] = left[i++];  if (moves) (*moves)++; }
    while (j < rightN) { arr[k++] = right[j++]; if (moves) (*moves)++; }
    delete[] left; delete[] right;
}

static void mergeSortRec(Book arr[], int low, int high,
                         long *comparisons, long *moves) {
    if (low < high) {
        int mid = low + (high - low) / 2;
        mergeSortRec(arr, low, mid, comparisons, moves);
        mergeSortRec(arr, mid + 1, high, comparisons, moves);
        merge(arr, low, mid, high, comparisons, moves);
    }
}
```

Jejak nyata `demo_mergesort.cpp` menunjukkan rekursi sungguh mencapai dasar sebelum digabung kembali: `merge(0..0, 1..1)` dan `merge(2..2, 3..3)` terjadi lebih dulu (menggabungkan elemen tunggal menjadi pasangan terurut), lalu `merge(0..1, 2..3)` menggabungkan pasangan-pasangan itu menjadi rangkaian 4-elemen yang terurut; hal yang sama terjadi untuk separuh kedua, dan `merge(0..3, 4..7)` terakhir menggabungkan kedua separuh 4-elemen yang terurut menjadi katalog 8-elemen yang sepenuhnya terurut. Seluruh pengurutan memakan 15 perbandingan dan 24 perpindahan elemen (transkrip lengkap di Lampiran 3.A).

> **O(n log n) yang terjamin — dengan biaya jujur O(n) ruang ekstra**
> Pembagian rekursif Merge sort SELALU membagi array dengan rata (tidak seperti partisi Quicksort yang bergantung data), sehingga kedalaman rekursi O(log n)-nya terjamin terlepas dari urutan awal input — tidak ada input buruk yang menurunkan Merge sort ke O(n^2) sebagaimana array terurut menurunkan Quicksort bab ini. Jaminan itu tidak gratis: `merge()` mengalokasikan dua array sementara (`left` dan `right`) pada setiap pemanggilan, sehingga Merge sort butuh ruang tambahan O(n) pada puncaknya — biaya nyata yang tidak dibayar keempat algoritma O(n^2) di atas (dan, sebagaimana ditunjukkan Bagian 3.8, Heap sort), karena mereka menyusun ulang array di tempat hanya dengan ruang ekstra O(1).

3.8 Heap Sort

Heap sort adalah algoritma bagi-dan-taklukkan ketiga di bab ini, dan ia mencapai jaminan O(n log n)-nya melalui struktur yang sama sekali berbeda: sebuah BINARY HEAP, disimpan langsung di dalam array yang sama yang sedang diurutkan (tanpa node pohon terpisah, tanpa pointer — hubungan parent/child sebuah heap dihitung dari indeks array: untuk node di indeks i, anak-anaknya berada di indeks 2i+1 dan 2i+2). Bab ini membangun sebuah MAX-HEAP — di mana kunci setiap parent lebih besar atau sama dengan kedua anaknya — yang menempatkan judul terbesar tunggal di indeks 0, root array.

Membangun dan menjaga heap itu membutuhkan satu operasi inti, menurunkan (sift down) sebuah nilai ke tempatnya yang benar kapan pun ia mungkin lebih kecil dari salah satu anaknya:

```
static void siftDown(Book arr[], int heapSize, int i,
                     long *comparisons, long *swaps) {
    while (true) {
        int largest = i, left = 2 * i + 1, right = 2 * i + 2;
        if (left  < heapSize) { if (comparisons) (*comparisons)++;
            if (titleLess(arr[largest], arr[left]))  largest = left; }
        if (right < heapSize) { if (comparisons) (*comparisons)++;
            if (titleLess(arr[largest], arr[right])) largest = right; }
        if (largest == i) break;
        swapBooks(arr[i], arr[largest]); if (swaps) (*swaps)++;
        i = largest;
    }
}

void heapSort(Book arr[], int n, long *comparisons, long *swaps) {
    for (int i = n / 2 - 1; i >= 0; i--)          // bangun max-heap
        siftDown(arr, n, i, comparisons, swaps);
    for (int end = n - 1; end > 0; end--) {       // ekstraksi maks berulang
        swapBooks(arr[0], arr[end]); if (swaps) (*swaps)++;
        siftDown(arr, end, 0, comparisons, swaps);
    }
}
```

Ini hanya heap secukupnya agar Heap sort bekerja — Bab 14 membahas selebihnya

Sebuah binary heap adalah struktur data yang sungguh berguna dan dapat dipakai ulang dengan sendirinya — sebagai priority queue yang mendukung operasi insert dan extract-maximum yang efisien, terlepas dari mengurutkan apa pun. Bab 14 membahas gambaran lengkap itu. Bab ini hanya mengimplementasikan apa yang dibutuhkan Heap sort itu sendiri: bangun heap sekali, lalu berulang kali tukar root-nya (selalu maksimum saat ini) ke ujung wilayah heap yang menyusut dan sift ulang. Itu sungguh cukup untuk mengurutkan dengan benar, sebagaimana dikonfirmasi keluaran eksekusi nyata bab ini — ini sekadar bukan keseluruhan cerita yang di-`struct`-kan sebagai ADT mandiri, yang menjadi tugas Bab 14, bukan bab ini.

Jejak nyata `demo_heapsort.cpp` menunjukkan fase build-heap men-sift-down dari indeks `n/2 - 1 = 3` kembali ke root, lalu fase ekstraksi berulang kali menukar maksimum saat ini ke ujung heap yang menyusut dan men-sift-ulang heap yang berkurang. Seluruh pengurutan memakan 26 perbandingan dan 21 penukaran pada katalog 8-buku, berhasil menghasilkan urutan terurut yang sama yang dicapai setiap algoritma lain di bab ini juga (transkrip lengkap di Lampiran 3.A). Kedua fase sama-sama O(n log n): membangun heap adalah O(n) (hasil klasik namun tidak jelas dengan sendirinya — men-sift-down n/2 node menyusuri pohon setinggi log n terlihat seperti O(n log n) tetapi JUMLAH pekerjaan sesungguhnya dibatasi oleh O(n), karena sebagian besar node berada dekat dasar dan hanya sift down jarak pendek), dan setiap dari n-1 ekstraksi berbiaya O(log n) untuk sift ulang — memberikan O(n log n) secara keseluruhan, terjamin, di setiap kasus, persis seperti Merge sort — tetapi, seperti keempat pengurutan O(n^2), sepenuhnya di tempat: ruang tambahan O(1), tanpa array sementara sama sekali.

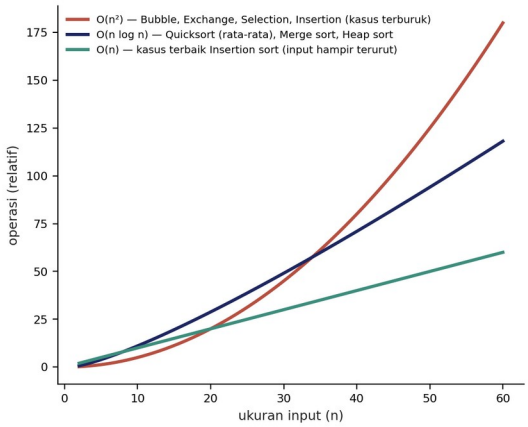
3.9 Membandingkan Ketujuhnya: Kompleksitas dan Stabilitas

Dengan ketujuh algoritma sudah diimplementasikan, bagian ini menjajarkannya. Gambar 3.2 memplot dua bentuk pertumbuhan paling penting di sini — O(n^2) dan O(n log n) — pada sumbu yang sama

(berdampingan dengan kasus terbaik O(n) Insertion sort), dan memberikan tabel kompleksitas/ruang/stabilitas lengkap untuk ketujuhnya.

Kompleksitas dan Stabilitas: Ketujuh Algoritma Pengurutan

Kompleksitas waktu (kasus terbaik / rata-rata / terburuk), ruang tambahan, dan stabilitas, untuk ketujuh algoritma yang dibahas bab ini.



Algoritma	Terbaik	Rata-rata	Terburuk	Ruang	Stabil?
Bubble	O(n²)*	O(n²)	O(n²)	O(1)	Ya
Exchange	O(n²)	O(n²)	O(n²)	O(1)	Tidak
Selection	O(n²)	O(n²)	O(n²)	O(1)	Tidak
Insertion	O(n)	O(n²)	O(n²)	O(1)	Ya
Quicksort	O(n log n)	O(n log n)	O(n²)†	O(log n)†	Tidak
Merge sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Ya
Heap sort	O(n log n)	O(n log n)	O(n log n)	O(1)	Tidak

*Bubble sort bab ini tidak punya flag keluar-dini, jadi kasus terbaiknya pun menjalankan semua pass — O(n²), berbeda dari kasus terbaik Insertion sort yang sungguhan O(n).
†Kasus terburuk Quicksort dan ruang tumpukan-rekursi O(log n)-nya sama-sama berasal dari pemilihan pivot yang konsisten sial (mis. input sudah terurut dengan pivot elemen terakhir).

Membaca tabel ini bersama Bagian 3.9

Empat algoritma di atas tidak pernah mengalahkan O(n²) pada kasus terburuk — alasan Quicksort/Merge/Heap sort ada. Merge sort menukar ruang ekstra O(n) dengan kasus TERBURUK yang sebaik kasus rata-ratanya — tidak ada input buruk. Quicksort tidak punya jaminan semacam itu, tetapi biasanya tercepat dalam praktik karena faktor konstannya kecil dan nyaris tidak butuh memori ekstra.

Gambar 3.2 — Kompleksitas waktu (terbaik/rata-rata/terburuk), ruang tambahan, dan stabilitas untuk ketujuh algoritma pengurutan yang dibahas bab ini.

Algoritma	Terbaik	Rata-rata	Terburuk	Ruang	Stabil?
Bubble	$O(n^2)$*	$O(n^2)$	$O(n^2)$	O(1)	Ya
Exchange	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	Tidak
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	Tidak
Insertion	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Ya
Quicksort	O(n log n)	O(n log n)	$O(n^2)$†	O(log n)†	Tidak
Merge sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Ya
Heap sort	O(n log n)	O(n log n)	O(n log n)	O(1)	Tidak

**Bubble sort bab ini tidak punya flag keluar-dini, jadi kasus terbaiknya pun menjalankan semua pass — $O(n^2)$, berbeda dari kasus terbaik Insertion sort yang sungguhan O(n). †Kasus terburuk Quicksort dan ruang tumpukan-rekursi O(log n)-nya sama-sama berasal dari pemilihan pivot yang konsisten sial (mis. input sudah terurut dengan pivot elemen terakhir bab ini).*

> **Mengapa stabilitas bukan sekadar catatan kaki**
> Bubble sort dan Insertion sort keduanya stabil: tidak satu pun pernah menukar dua elemen berkunci sama saling melewati (penukaran berdekatan Bubble sort hanya terpicu pada ketidaksamaan KETAT; loop geser Insertion sort berhenti seketika begitu menemukan elemen yang sama atau lebih kecil, tidak pernah memindahkan elemen berkunci sama keluar dari posisi relatifnya). Selection sort dan Exchange sort keduanya tidak stabil — sebuah penukaran bisa melompati beberapa elemen berkunci sama, mengubah urutan relatif mereka. Di antara ketiga bagi-dan-taklukkan, Merge sort stabil (langkah merge-nya, `!titleLess(right[j], left[i])`, sengaja memihak separuh KIRI pada kunci yang sama, mempertahankan urutan asli), sementara Quicksort dan Heap sort keduanya tidak stabil — penukarannya rutin memindahkan elemen berkunci sama saling melewati. Bagi Pustaka Ceria secara khusus, stabilitas akan penting jika katalog pernah memuat dua record dengan judul yang persis sama (dua cetakan satu buku, misalnya) dan urutan asli suatu field LAIN perlu dipertahankan melalui pengurutan berdasarkan judul.

Kesimpulan praktis yang didukung tabel ini: empat algoritma di bab ini (Bubble, Exchange, Selection, Insertion) tidak pernah mengalahkan O(n^2) pada kasus terburuk, persis mengapa ketiga lainnya ada. Jaminan Merge sort paling kuat — O(n log n) di SETIAP kasus, tanpa input buruk — dengan biaya ruang O(n) yang nyata. Quicksort tidak punya jaminan semacam itu, tetapi faktor konstannya yang kecil dan operasi di-tempatnya (tanpa array ekstra) biasanya membuatnya tercepat dalam praktik, terlepas dari kasus terburuknya. Heap sort berada di antara keduanya: jaminan O(n log n) yang sama dengan Merge sort, tetapi di tempat seperti pengurutan O(n^2) — kombinasi yang sungguh berguna, dengan biaya tidak stabil dan, dalam praktik, biasanya lebih lambat dari Quicksort karena faktor konstan yang lebih besar dari pola akses memorinya yang kurang ramah-cache.

3.10 Lampiran 3.A — Log Validasi

Kedelapan program di bawah ini (`demo_bubble.cpp` hingga `demo_heapsort.cpp`, plus `demo_all.cpp`) dikompilasi dengan `g++ -Wall -Wextra -std=c++17` (nol peringatan dari kedelapannya) dan benar-benar dijalankan pada katalog 8-buku Pustaka Ceria; transkrip di bawah adalah keluaran terminal yang asli dan tidak diedit — bukan keluaran “yang diharapkan” yang ditulis tangan. Agar lampiran ini tetap terbaca, transkrip lengkap ditampilkan untuk `demo_bubble.cpp` (termasuk daftar katalog sebelum/sesudah yang tercetak) dan untuk program ringkasan `demo_all.cpp`; untuk keenam program lainnya, daftar katalog (identik bentuknya dengan milik `demo_bubble.cpp`) dihilangkan dan hanya jejak serta total masing-masing program yang ditampilkan — keluaran lengkap dan tidak dipersingkat setiap program menyertai kode bab ini.

3.10.1 demo_bubble.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_bubble book.cpp sort.cpp demo_bubble.cpp
$ ./demo_bubble
=== Pustaka Ceria: Bubble Sort (ascending by title) ===

--- Catalog in arrival order (unsorted by title) ---
B1005  Algoritma & Struktur Data        Rinaldi Munir        Teknologi
B1002  Bumi Manusia                     Pramoedya A. Toer    Fiksi
B1008  Negeri 5 Menara                  Ahmad Fuadi          Fiksi
B1001  Laskar Pelangi                   Andrea Hirata        Fiksi
B1006  Cerita Rakyat Nusantara          Tim Pustaka Ceria    Anak-anak
B1003  Filosofi Kopi                    Dee Lestari          Fiksi
B1007  Fisika Dasar                     Halliday & Resnick   Sains
B1004  Sejarah Nusantara                Slamet Muljana       Sejarah

--- Trace (ids only, adjacent-pair compares each pass) ---
  start [B1005 B1002 B1008 B1001 B1006 B1003 B1007 B1004]
  after pass 1 [B1005 B1002 B1001 B1006 B1003 B1007 B1008 B1004]
  after pass 2 [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
  after pass 3 [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
  after pass 4 [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
  after pass 5 [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
  after pass 6 [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
  after pass 7 [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
-- bubbleSort done: 28 comparisons, 7 swaps --

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

Verification: SORTED (ascending by title) -- OK
Totals: 28 comparisons, 7 swaps, n=8
```

Perhatikan bahwa "after pass 3" hingga "after pass 7" identik dengan "after pass 2" — array sudah sepenuhnya terurut setelah pass 2, tetapi karena Bubble sort bab ini tidak punya flag keluar-dini (Bagian 3.2), ia tetap menjalankan semua pass yang tersisa (via perbandingan yang tidak pernah memicu penukaran), menegaskan ulang bahwa tidak ada lagi yang perlu berpindah.

3.10.2 demo_exchange.cpp (jejak + total)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_exchange book.cpp sort.cpp
demo_exchange.cpp
$ ./demo_exchange
  start [B1005 B1002 B1008 B1001 B1006 B1003 B1007 B1004]
  swap(2,3) -> [B1005 B1002 B1001 B1008 B1006 B1003 B1007 B1004]
  swap(2,4) -> [B1005 B1002 B1006 B1008 B1001 B1003 B1007 B1004]
  swap(3,4) -> [B1005 B1002 B1006 B1001 B1008 B1003 B1007 B1004]
  swap(3,5) -> [B1005 B1002 B1006 B1003 B1008 B1001 B1007 B1004]
  swap(4,5) -> [B1005 B1002 B1006 B1003 B1001 B1008 B1007 B1004]
  swap(4,6) -> [B1005 B1002 B1006 B1003 B1007 B1008 B1001 B1004]
  swap(5,6) -> [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
-- exchangeSort done: 28 comparisons, 7 swaps --
Verification: SORTED (ascending by title) -- OK
Totals: 28 comparisons, 7 swaps, n=8
```

3.10.3 demo_selection.cpp (jejak + total)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_selection book.cpp sort.cpp
demo_selection.cpp
$ ./demo_selection
  start [B1005 B1002 B1008 B1001 B1006 B1003 B1007 B1004]
  after pass 1 (min was at 0) [B1005 B1002 B1008 B1001 B1006 B1003 B1007 B1004]
  after pass 2 (min was at 1) [B1005 B1002 B1008 B1001 B1006 B1003 B1007 B1004]
  after pass 3 (min was at 4) [B1005 B1002 B1006 B1001 B1008 B1003 B1007 B1004]
  after pass 4 (min was at 5) [B1005 B1002 B1006 B1003 B1008 B1001 B1007 B1004]
  after pass 5 (min was at 6) [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
  after pass 6 (min was at 5) [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
  after pass 7 (min was at 6) [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
-- selectionSort done: 28 comparisons, 3 swaps --
Verification: SORTED (ascending by title) -- OK
Totals: 28 comparisons, 3 swaps, n=8
```

3.10.4 demo_insertion.cpp (kedua run, jejak + total)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_insertion book.cpp sort.cpp
demo_insertion.cpp
$ ./demo_insertion
--- Run 1: catalog in arrival order (typical unsorted input) ---
  start [B1005 B1002 B1008 B1001 B1006 B1003 B1007 B1004]
  after inserting B1002 [B1005 B1002 B1008 B1001 B1006 B1003 B1007 B1004]
  after inserting B1008 [B1005 B1002 B1008 B1001 B1006 B1003 B1007 B1004]
  after inserting B1001 [B1005 B1002 B1001 B1008 B1006 B1003 B1007 B1004]
  after inserting B1006 [B1005 B1002 B1006 B1001 B1008 B1003 B1007 B1004]
  after inserting B1003 [B1005 B1002 B1006 B1003 B1001 B1008 B1007 B1004]
  after inserting B1007 [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
  after inserting B1004 [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
-- insertionSort done: 14 comparisons, 7 shifts --
Verification: SORTED (ascending by title) -- OK
Run 1 totals: 14 comparisons, 7 shifts, n=8

--- Run 2: the SAME 8 books, ALREADY sorted by title (best case) ---
Verification: SORTED (ascending by title) -- OK
Run 2 totals: 7 comparisons, 0 shifts, n=8

--- Best case vs. typical case, measured (not asserted) ---
Run 1 (arrival order):        14 comparisons, 7 shifts
Run 2 (already sorted):       7 comparisons, 0 shifts   <- exactly n-1 = 7
comparisons, 0 shifts: O(n), the genuine best case
```

3.10.5 demo_quicksort.cpp (jejak + total)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_quicksort book.cpp sort.cpp
demo_quicksort.cpp
$ ./demo_quicksort
  start [B1005 B1002 B1008 B1001 B1006 B1003 B1007 B1004]
  partition(0..7): pivot = B1004
    -> [B1005 B1002 B1008 B1001 B1006 B1003 B1007 B1004]
  partition(0..6): pivot = B1007
    -> [B1005 B1002 B1006 B1003 B1007 B1001 B1008]
  partition(0..3): pivot = B1003
    -> [B1005 B1002 B1006 B1003]
  partition(0..2): pivot = B1006
    -> [B1005 B1002 B1006]
  partition(0..1): pivot = B1002
    -> [B1005 B1002]
  partition(5..6): pivot = B1008
    -> [B1005 B1002 B1006 B1003 B1007 B1001 B1008]
  final [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
-- quickSort done: 20 comparisons, 3 swaps --
Verification: SORTED (ascending by title) -- OK
Totals: 20 comparisons, 3 swaps, n=8
```

3.10.6 demo_mergesort.cpp (jejak + total)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_mergesort book.cpp sort.cpp
demo_mergesort.cpp
$ ./demo_mergesort
  start [B1005 B1002 B1008 B1001 B1006 B1003 B1007 B1004]
  merge(0..0, 1..1) -> [B1005 B1002]
  merge(2..2, 3..3) -> [B1001 B1008]
  merge(0..1, 2..3) -> [B1005 B1002 B1001 B1008]
  merge(4..4, 5..5) -> [B1006 B1003]
  merge(6..6, 7..7) -> [B1007 B1004]
  merge(4..5, 6..7) -> [B1006 B1003 B1007 B1004]
  merge(0..3, 4..7) -> [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
  final [B1005 B1002 B1006 B1003 B1007 B1001 B1008 B1004]
-- mergeSort done: 15 comparisons, 24 element-moves --
Verification: SORTED (ascending by title) -- OK
Totals: 15 comparisons, 24 element-moves, n=8
```

3.10.7 demo_heapsort.cpp (jejak + total)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_heapsort book.cpp sort.cpp
demo_heapsort.cpp
$ ./demo_heapsort
  start [B1005 B1002 B1008 B1001 B1006 B1003 B1007 B1004]
    siftDown swap(3,7) -> [B1005 B1002 B1008 B1004 B1006 B1003 B1007 B1001]
    siftDown swap(1,3) -> [B1005 B1004 B1008 B1002 B1006 B1003 B1007 B1001]
    siftDown swap(3,7) -> [B1005 B1004 B1008 B1001 B1006 B1003 B1007 B1002]
    siftDown swap(0,1) -> [B1004 B1005 B1008 B1001 B1006 B1003 B1007 B1002]
    siftDown swap(1,3) -> [B1004 B1001 B1008 B1005 B1006 B1003 B1007 B1002]
    siftDown swap(3,7) -> [B1004 B1001 B1008 B1002 B1006 B1003 B1007 B1005]
  heap built [B1004 B1001 B1008 B1002 B1006 B1003 B1007 B1005]
  ... (repeated root-extraction + re-sift, 7 more times) ...
-- heapSort done: 26 comparisons, 21 swaps --
Verification: SORTED (ascending by title) -- OK
Totals: 26 comparisons, 21 swaps, n=8
```

3.10.8 demo_all.cpp (transkrip lengkap — ketujuhnya berdampingan)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_all book.cpp sort.cpp demo_all.cpp
$ ./demo_all
--- Catalog sorted by title (the common result all seven converge on) ---
B1005  Algoritma & Struktur Data        Rinaldi Munir        Teknologi
B1002  Bumi Manusia                     Pramoedya A. Toer    Fiksi
B1006  Cerita Rakyat Nusantara          Tim Pustaka Ceria    Anak-anak
B1003  Filosofi Kopi                    Dee Lestari          Fiksi
B1007  Fisika Dasar                     Halliday & Resnick   Sains
B1001  Laskar Pelangi                   Andrea Hirata        Fiksi
B1008  Negeri 5 Menara                  Ahmad Fuadi          Fiksi
B1004  Sejarah Nusantara                Slamet Muljana       Sejarah

--- Summary: real measured counts, n=8 ---
  Algorithm   Sorted?   Comparisons      Swaps/Shifts/Moves
  Bubble      OK        28               7 (swaps)
  Exchange    OK        28               7 (swaps)
  Selection   OK        28               3 (swaps)
  Insertion   OK        14               7 (shifts)
  Quicksort   OK        20               3 (swaps)
  Merge sort  OK        15               24 (moves)
  Heap sort   OK        26               21 (swaps)

Overall: all seven algorithms produced a genuinely sorted catalog.
```

Catatan kejujuran

Setiap hitungan, setiap baris jejak, dan setiap baris verifikasi yang ditampilkan di atas (di kedelapan program) berasal langsung dari kompilasi-dan-eksekusi nyata di lingkungan build bab ini — tidak ada ketidaksesuaian atau bug yang ditemukan pada kode bab ini sendiri, dan tidak ada yang direkayasa di sini demi efek. Zip kode yang dikirimkan juga diekstrak ulang dan di-build ulang secara independen untuk memastikan deliverable sesungguhnya (bukan hanya salinan kerja) terkompilasi dan berjalan secara identik. Kebijakan tetap mata kuliah ini adalah melaporkan hasil validasi persis seperti apa adanya, baik itu mencakup bug maupun tidak.

3.11 Ringkasan Bab dan Poin-Poin Kunci

- Pengurutan ada karena pencarian cepat Bab 2 (binary, interpolation) sungguh membutuhkan input terurut lebih dulu — bab ini akhirnya mengajarkan, dari nol, langkah yang tadi dipinjam Bab 2 dari std::sort.
- Bubble, Exchange, dan Selection sort semuanya berbiaya O(n^2) di setiap kasus dan semuanya melakukan n·(n-1)/2 perbandingan yang sama pada katalog 8-buku bab ini (28), tetapi POLA perbandingan dan jumlah penukarannya berbeda: Bubble hanya membandingkan pasangan berdekatan; Exchange membandingkan satu elemen tetap terhadap setiap elemen berikutnya, menukar berulang kali; Selection memindai penuh sebelum satu penukaran per pass (3 penukaran di sini, vs. 7 untuk Bubble dan Exchange).
- Insertion sort juga O(n^2) kasus terburuk, tetapi kasus terbaiknya adalah O(n) yang sungguh terukur pada input yang hampir terurut (7 perbandingan, 0 geseran pada run 8-buku yang sudah terurut) — satu-satunya algoritma di antara empat pertama dengan perilaku adaptif yang nyata, karena Bubble sort bab ini sengaja menghilangkan flag keluar-dini.

- Quicksort (partisi Lomuto, pivot elemen terakhir) rata-rata O(n log n) dengan membagi array di sekitar pivot dan merekursi kedua sisi, tetapi kasus terburuknya adalah O(n^2) yang nyata dan dapat dijelaskan pada input yang sudah terurut, khususnya karena pilihan pivotnya yang konsisten.
- Merge sort membagi rata setiap kalinya, menjamin O(n log n) di SETIAP kasus tanpa input buruk — dengan biaya jujur ruang tambahan O(n) untuk buffer merge sementaranya, berbeda dari setiap algoritma lain di bab ini.
- Heap sort membangun max-heap berbasis array yang mandiri (teori heap lengkap ditunda ke Bab 14) dan berulang kali mengekstrak root-nya, menjamin O(n log n) seperti Merge sort tetapi sepenuhnya di tempat (ruang O(1)) seperti pengurutan O(n^2).
- Stabilitas — apakah elemen berkunci sama mempertahankan urutan relatifnya — membagi ketujuhnya dengan bersih: Bubble, Insertion, dan Merge sort stabil; Exchange, Selection, Quicksort, dan Heap sort tidak.
- Ketujuh algoritma, dijalankan pada katalog 8-buku Pustaka Ceria nyata yang sama, secara independen berkumpul pada urutan yang sama persis yang benar terurut — sungguhan, terkompilasi, dan dijalankan, bukan diasumsikan.

3.12 Pertanyaan Tinjauan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Bab 4 (Kuis 1).

1. Bubble, Exchange, dan Selection sort semuanya melakukan persis n·(n-1)/2 perbandingan pada input yang sama, namun jumlah PENUKARAN mereka pada katalog 8-buku bab ini masing-masing 7, 7, dan 3. Jelaskan, merujuk pada logika perbandingan-ke-penukaran masing-masing algoritma, mengapa jumlah penukaran Selection sort jauh lebih rendah.
2. Bubble sort bab ini tidak punya flag keluar-dini. Deskripsikan persis perubahan kode apa (Bagian 3.2) yang akan memberinya satu, dan jelaskan kompleksitas kasus TERBAIK-nya akan menjadi apa sesudahnya — dan mengapa itu tidak akan mengubah kasus TERBURUKnya.
3. Run 2 Insertion sort (input yang sudah terurut) mengukur persis n-1 = 7 perbandingan dan 0 geseran. Telusuri, langkah demi langkah, mengapa kondisi `break` loop while internal algoritma dijamin terpicu pada perbandingan pertama setiap langkah ketika input sudah terurut.
4. Kasus terburuk Quicksort adalah O(n^2) pada input yang sudah terurut, khususnya karena implementasi bab ini selalu memilih elemen terakhir sebagai pivot. Usulkan satu perubahan konkret pada aturan pemilihan-pivot yang akan menghindari kasus terburuk khusus ini, dan jelaskan (secara informal) mengapa perubahan yang Anda usulkan membantu.
5. Merge sort butuh ruang tambahan O(n); Heap sort hanya butuh O(1). Keduanya menjamin O(n log n) di setiap kasus. Dengan itu, deskripsikan satu skenario realistis di mana Anda dengan sengaja memilih Merge sort daripada Heap sort meski biaya memori ekstranya, dan satu di mana Anda memilih sebaliknya.

6. Menggunakan kolom stabilitas Bagian 3.9, jelaskan secara konkret apa yang bisa salah bagi Pustaka Ceria jika katalog memuat dua entri dengan judul yang persis identik (misalnya, dua cetakan buku yang sama dengan tahun terbit berbeda) dan perpustakaan mengurutkan tampilan raknya berdasarkan judul memakai Selection sort alih-alih Merge sort.

Referensi

[1] B.R. Preiss. “Data Structures and Algorithms with Object-Oriented Design Patterns in C++.” John Wiley & Sons, 1998.

[2] P.J. Deitel, H.M. Deitel. “C++ How to Program.” 10th ed., Pearson, 2017.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Algoritma pengurutan, Bab 2, 6–8.)

[4] R. Sedgewick, K. Wayne. “Algorithms.” 4th ed., Addison-Wesley, 2011. (Quicksort, Mergesort, Priority Queues / Heapsort.)

[5] cppreference.com. “std::sort.” https://en.cppreference.com/w/cpp/algorithm/sort (diakses Agustus 2026).

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 4 · BAB LATIHAN

Kuis 1

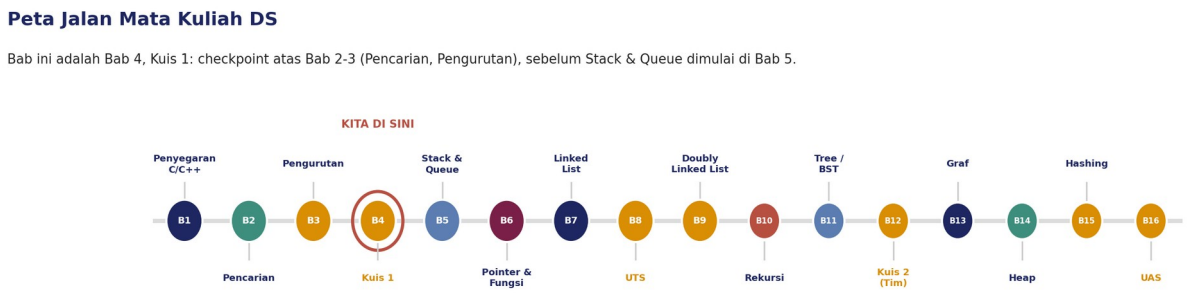
Tidak ada materi baru di sini — ujian coding individual, open-book, take-home yang menguji persis apa yang diajarkan Bab 2 dan Bab 3: pencarian dan pengurutan. Bab ini TIDAK menyertakan subfolder code/ — keempat program di bawah adalah pekerjaan Anda sendiri untuk ditulis dan dikumpulkan.

Tujuan Pembelajaran — setelah mengerjakan bab ini Anda akan mampu:
(1) Mengenali bahwa keempat tugas Kuis 1 di bawah semuanya bermuara pada ide pencarian/Big-O dari Bab 2 atau ide pengurutan dari Bab 3, sehingga tidak diperlukan teori baru, hanya penerapan yang benar. (2) Mengurutkan daftar string biasa, baik dengan algoritme dari nol dari Bab 3 maupun dengan `std::sort` dan comparator, serta memberi alasan yang tepat atas pilihan tersebut. (3) Merancang strategi satu-lintasan (single-pass), O(n), "sudah-pernah-dilihat" untuk menemukan karakter berulang pertama dalam sebuah string, dan menjelaskan mengapa perbandingan nested-loop naif berjalan $O(n^2)$. (4) Mengadaptasi keterampilan pencarian Bab 2 ke jenis rekam data yang BUKAN `struct Book` — pasangan nama/saldo — dengan menyadari bahwa KETERAMPILANNYA (lookup linear atau lookup pada data terurut) tetap berlaku meski datanya berbeda. (5) Menulis comparator custom (fungsi atau lambda) yang mengurutkan string berdasarkan skor "importance" buatan programmer, bukan urutan alfabetis biasa, dan menjelaskan bagaimana comparator itu terpasang ke salah satu algoritme pengurutan pembanding dari Bab 3.

4.1 Ulasan: Bab 2 dan 3 Secara Singkat

Bab 2 mengajukan pertanyaan sederhana pada katalog Pustaka Ceria — “apakah buku ini ada di sini, dan di mana?” — dan menjawabnya dengan empat cara: sequential search (tanpa asumsi urutan apa pun), sentinel search (optimisasi kecil dan jujur atas inner loop sequential search), binary search (mengasumsikan input sudah terurut, menukar asumsi itu dengan kecepatan O(log n)), dan interpolation search (mengasumsikan kunci numerik yang tersebar cukup merata, lebih cepat lagi bila asumsi itu terpenuhi). Bab yang sama memperkenalkan notasi Big-O secara formal — O(1), O(log n), O(n) — sebagai kosakata untuk membandingkan bagaimana biaya sebuah algoritme tumbuh terhadap ukuran input n, mengabaikan faktor konstan.

Bab 3 menjawab pertanyaan yang jawabannya sudah diam-diam diasumsikan oleh kode Bab 2 sendiri — “bagaimana caranya membuat katalog terurut sejak awal?” — dengan tujuh algoritme: empat sort in-place $O(n^2)$ (Bubble, Exchange, Selection, Insertion) dan tiga sort divide-and-conquer O(n log n) (Quicksort, Merge sort, Heap sort). Bab 3 juga menambahkan $O(n^2)$ dan O(n log n) ke kosakata laju pertumbuhan yang dimulai Bab 2, serta memperkenalkan STABILITAS — apakah elemen dengan kunci yang sama tetap mempertahankan urutan relatif aslinya setelah diurutkan.



Gambar 4.1 — Bab ini berada di Bab 4 dari enam belas bab peta jalan DS: sebuah checkpoint, bukan topik baru, Kuis 1 sebelum Bab 5 melanjutkan dengan Stack dan Queue.

Tidak ada yang baru di bawah ini — hanya dikombinasikan ulang

Setiap satu dari empat tugas Kuis 1 (Bagian 4.3) adalah penerapan langsung dan biasa dari sesuatu yang sudah dibahas tuntas di Bab 2 atau Bab 3. Tugas 1 (pengurutan alfabetis) dan Tugas 4 (pengurutan metrik custom) sama-sama wilayah Bab 3; Tugas 2 (karakter berulang pertama) bersandar pada penalaran kompleksitas single-pass-vs-nested-loop yang SAMA seperti yang diperkenalkan bagian Big-O Bab 2; Tugas 3 (mencari dataset nama/saldo) adalah keterampilan pencarian Bab 2, hanya diarahkan ke jenis rekam data selain `struct Book`. Jika ada yang terasa asing, itu adalah sinyal untuk membaca ulang bagian terkait di Bab 2 atau 3 sebelum mengerjakan tugas tersebut, bukan tanda bahwa kuis ini mengharapkan sesuatu yang baru.

4.2 Cara Menggunakan Bab Latihan Ini

Apa bab ini — dan apa yang bukan

Ini adalah Bab Latihan: bab praktik, bukan bab materi baru. Kuis 1, seperti asesmen DS sesungguhnya yang menjadi modelnya, adalah ujian coding INDIVIDUAL, open-book, take-home — Anda bekerja sendiri, bukan dalam tim (pengumpulan tim hanya berlaku untuk Kuis 2, nanti dalam mata kuliah ini). Bagian 4.3 di bawah menjabarkan keempat tugas Kuis 1, masing-masing disertai callout panduan (THINK, TRAP, atau INSIGHT), bukan solusi lengkap yang sudah dikerjakan. Seperti bab UTS, Kuis 2, dan UAS nanti, bab ini TIDAK menyertakan subfolder code/ — keempat program yang Anda tulis untuk Kuis 1 adalah pekerjaan individual Anda sendiri, dikompilasi dan dijalankan di komputer Anda sendiri sebelum dikumpulkan.

Sebelum Anda mulai

Untuk setiap tugas di bawah, nyatakan ulang persoalannya dengan kata-kata Anda sendiri terlebih dahulu dan kenali satu ide inti dari Bab 2 atau Bab 3 yang sebenarnya sedang diuji — keempat callout di Bagian 4.3 menyebutkan ide itu secara eksplisit. Menulis beberapa baris pseudocode sebelum menyentuh keyboard biasanya menghemat lebih banyak waktu daripada yang dihabiskannya.

4.3 Kumpulan Soal Kuis 1

Kuis 1 terdiri atas empat tugas coding independen, masing-masing bernilai 25 poin, total 100 poin — persis mengikuti distribusi poin Kuis 1 DS yang sesungguhnya. Setiap tugas di bawah adalah program C++ yang berdiri sendiri: baca (atau hard-code, jika instruksi mata kuliah Anda sendiri tidak menyebutkan file input) masukan yang dideskripsikan, hasilkan keluaran yang dideskripsikan, dan kompilasi dengan bersih menggunakan `g++ -Wall -Wextra -std=c++17`, toolchain yang sama yang digunakan setiap bab dalam buku ini sejak Bab 1.

Asal Setiap Tugas Kuis 1

Kuis 1 terdiri atas empat tugas coding yang dinilai secara individual -- setiap tugas di bawah dapat ditelusuri kembali ke keterampilan spesifik dari Bab 2 atau Bab 3.

#	Tugas Kuis 1	Berasal dari	Poin
1	Urutkan daftar kata secara alfabetis	Bab 3 -- Pengurutan: algoritme apa pun yang benar, atau std::sort dengan comparator	25
2	Cari karakter berulang pertama, O(n)	Bab 2 -- Pencarian & Big-O: pendekatan satu-lintasan, bukan nested-loop	25
3	Cari satu entri pada dataset nama -> saldo	Bab 2 -- Pencarian: keterampilan lookup yang sama, jenis rekam data berbeda	25
4	Urutkan string berdasarkan metrik "importance" custom	Bab 3 -- Pengurutan: comparator/kunci custom, bukan urutan alfabetis biasa	25

Total: 100 poin

Gambar 4.2 — Setiap tugas di bagian ini dapat ditelusuri kembali ke keterampilan spesifik dari Bab 2 atau Bab 3.

4.3.1 Tugas 1 — Urutkan Daftar Kata Secara Alfabetis (25 poin)

Diberikan daftar kata (dibaca dari input, atau `std::vector<std::string>` hard-code jika instruksi mata kuliah Anda mengatakan demikian), urutkan menjadi urutan alfabetis menaik dan cetak daftar terurutnya, satu kata per baris.

> **Petunjuk**
>
> Ini persis wilayah Bab 3 — algoritme mana pun dari ketujuh yang dibahas di sana (Bubble hingga Heap sort) akan mendapat nilai penuh untuk hasil yang benar, selama perbandingannya menggunakan urutan leksikografis biasa (operator `<` yang sudah didukung `std::string`, atau tanda hasil `strcmp` jika Anda bekerja dengan C-style string). Jika Anda lebih memilih tidak menulis ulang rutin sort penuh di bawah tekanan waktu kuis, `std::sort(words.begin(), words.end())` adalah pilihan yang sepenuhnya sah dan jujur untuk tugas ini — Bab 3 sendiri sudah bersandar pada `std::sort` di beberapa tempat, dan kuis ini tidak mewajibkan Anda mengimplementasikannya ulang dari nol kecuali instruksi mata kuliah Anda sendiri mengatakan sebaliknya.

Sensitivitas huruf besar/kecil adalah jebakan nyata di sini

Operator `<` biasa pada `std::string` membandingkan nilai byte, sehingga `"Zebra" < "apple"` bernilai BENAR (huruf kapital terurut sebelum semua huruf kecil dalam ASCII) — hasil yang tampak salah bagi pembaca manusia yang mengharapkan urutan kamus. Putuskan, dan NYATAKAN dalam komentar, apakah pengurutan Anda case-sensitive atau case-insensitive sebelum menulis satu perbandingan pun — dan jika input tugas itu sendiri dijamin seluruhnya huruf kecil (atau seluruhnya huruf besar), nyatakan hal itu dan lanjutkan; jangan membuat langkah case-folding berlebihan yang tidak pernah diminta tugas ini.

4.3.2 Tugas 2 — Temukan Karakter Berulang Pertama, O(n) (25 poin)

Diberikan satu string, temukan dan cetak karakter PERTAMA yang berulang — yaitu, dipindai dari kiri ke kanan, karakter pertama yang nilainya sudah muncul sebelumnya di string tersebut. Jika tidak ada karakter yang berulang, cetak pesan "tidak ditemukan pengulangan" yang jelas. Solusi Anda harus berjalan dalam waktu O(n), dengan n adalah panjang string.

Pendekatan naif adalah $O(n^2)$, dan kuis ini secara khusus memeriksa hal itu

Ide pertama yang paling jelas — untuk setiap karakter, bandingkan dengan setiap karakter lain setelahnya, mencari kecocokan — adalah nested loop: O(n) posisi luar dikali O(n) perbandingan dalam, menghasilkan $O(n^2)$ total, persis laju pertumbuhan yang diperkenalkan Bab 3 untuk mendeskripsikan worst case Bubble/Exchange/Selection sort. Pendekatan itu biasanya tetap menghasilkan jawaban yang BENAR, tetapi tidak memenuhi persyaratan eksplisit O(n) pada tugas ini, dan nilai penuh bergantung pada kompleksitasnya, bukan sekadar keluarannya.

Satu lintasan tunggal, satu catatan "sudah pernah dilihat"

Trik O(n)-nya adalah melakukan satu lintasan kiri-ke-kanan sambil menyimpan catatan setiap karakter yang SUDAH terlihat sejauh ini — `std::array<bool, 256>` yang diindeks kode karakter, atau `std::unordered_set<char>`, keduanya memberi lookup O(1) rata-rata per karakter. Pada setiap posisi, periksa apakah karakter saat ini sudah ada dalam catatan tersebut: jika ya, itulah pengulangan pertama — berhenti dan laporkan segera; jika tidak, tambahkan ke catatan dan lanjutkan. Satu lintasan, kerja O(1) per karakter, total O(n) — ide "menukar sedikit memori dengan banyak kecepatan" yang sama yang membuat binary dan interpolation search cepat di Bab 2, diterapkan di sini pada satu pemindaian linear, bukan pencarian pada array terurut.

4.3.3 Tugas 3 — Cari Satu Entri pada Dataset Nama → Saldo (25 poin)

Diberikan dataset kecil berisi rekam data, masing-masing memasangkan NAMA seseorang dengan SALDO UANG (misalnya, `struct Account { std::string name; double balance; };` dan sebuah array atau `std::vector` berisi beberapa akun seperti itu), cari satu entri dalam dataset berdasarkan nama dan laporkan apakah ditemukan dan, jika ya, berapa saldonya. Jika tidak ada entri yang cocok, laporkan dengan jelas bahwa nama tersebut tidak ditemukan.

Petunjuk — ini keterampilan Bab 2, bukan data Bab 2

`struct Account` adalah bentuk rekam data yang berbeda dari `struct Book` di Bab 1 dan 2 — tetapi PENCARIANNYA sendiri adalah keterampilan yang PERSIS SAMA seperti yang sudah dibahas Bab 2: diberikan sebuah kunci (di sini, nama, bukan id buku) dan sekumpulan rekam data, putuskan ditemukan-atau-tidak dan, jika ditemukan, yang mana. Jika dataset-nya kecil dan tidak terurut, sequential search (Bagian 2.2) adalah pilihan yang sepenuhnya wajar dan jujur; jika Anda memilih mengurutkan dataset berdasarkan nama terlebih dahulu lalu menggunakan binary search (Bagian 2.4), itu juga cara yang sah untuk menunjukkan teknik yang lebih cepat, selama Anda benar-benar mengurutkan SEBELUM mencari — binary search pada data yang belum terurut diam-diam memberi jawaban salah, bukan error, persis jebakan yang diperingatkan Bab 2.

Membandingkan nama std::string dengan == terlihat aman, tetapi pencarian exact-match punya jebakannya sendiri

Perbandingan exact-match `==` pada nama secara default sensitif huruf besar/kecil dan sensitif spasi — `"Rina"` dan `"rina "` (dengan spasi di akhir) tidak akan cocok meskipun manusia akan menganggapnya sebagai kueri yang sama. Putuskan sejak awal, dan nyatakan dalam komentar, apakah pencarian Anda adalah exact match yang ketat atau sesuatu yang lebih toleran (di-trim, case-insensitive) — dan jika instruksi tugas itu sendiri menyebutkan exact match, jangan menambahkan normalisasi yang tidak diminta yang justru bisa menimbulkan bug sendiri.

4.3.4 Tugas 4 — Urutkan String Berdasarkan Metrik "Importance" Custom (25 poin)

Diberikan daftar string, urutkan bukan berdasarkan urutan alfabetis biasa melainkan berdasarkan skor "importance" buatan programmer — sebuah fungsi `score(const std::string&) -> T` yang Anda rancang sendiri (misalnya: panjang string, jumlah huruf vokal, jumlah karakter tertentu, atau aturan lain yang ditentukan instruksi mata kuliah Anda sendiri), mengurutkan daftar dalam urutan menurun berdasarkan skor tersebut (importance tertinggi lebih dulu).

Algoritme pengurutannya nyaris tidak berubah — hanya comparator-nya yang berubah

Setiap algoritme pengurutan yang dibahas Bab 3 membandingkan dua elemen dan memutuskan mana yang lebih dulu; tidak satu pun dari algoritme itu peduli APA sebenarnya yang diuji perbandingan tersebut. Mengganti `titleLess(a, b)` (comparator judul buku Bab 3) dengan `scoreGreater(a, b)` — fungsi atau lambda yang mengembalikan true ketika `score(a) > score(b)` — adalah satu-satunya perubahan yang dibutuhkan untuk mengarahkan salah satu dari ketujuh algoritme Bab 3 ke tugas ini, atau untuk mengarahkan `std::sort(strings.begin(), strings.end(), scoreGreater)` ke tugas ini dengan mudahnya. Inilah tepatnya alasan Bab 3 mempertahankan comparator-nya (`titleLess`) sebagai fungsi terpisah dan bernama sepanjang bab itu: comparator adalah unit kecil dan mudah-ditukar dari "apa arti 'lebih kecil/lebih besar' di sini", terlepas dari mekanisme pengurutan yang dibangun di sekitarnya.

Comparator yang bukan strict weak ordering dapat merusak std::sort secara diam-diam
Jika dua string berbeda dapat memiliki skor importance yang PERSIS SAMA, comparator Anda harus mengembalikan `false` untuk `scoreGreater(a, b)` ketika `score(a) == score(b)` — jangan pernah mengembalikan `true` untuk `scoreGreater(a, b)` DAN `scoreGreater(b, a)` pada pasangan yang sama. Comparator yang melanggar aturan ini (bug umum: menggunakan `>=` alih-alih `>` di dalamnya) adalah undefined behavior khusus untuk `std::sort` — dapat crash, loop, atau diam-diam mengacaukan array, pada sebagian input tetapi tidak pada input lain, yang menjadikannya bug yang benar-benar berbahaya dan sulit dikenali, bukan sekadar kosmetik.

4.4 Format dan Penilaian Kuis 1

Kuis 1 adalah ujian coding INDIVIDUAL, open-book, take-home — bukan tugas tim (pengumpulan tim hingga tiga mahasiswa berlaku untuk Kuis 2 saja, nanti dalam mata kuliah ini). "Open-book" berarti buku teks, catatan Anda sendiri, dan materi referensi C++ standar (seperti cppreference.com) semuanya diperbolehkan selama Anda mengerjakan; ini TIDAK berarti Anda boleh menyalin kode mahasiswa lain atau mengumpulkan pekerjaan yang bukan benar-benar milik Anda sendiri — ujian open-book tetap menguji apakah ANDA dapat menerapkan apa yang telah diajarkan mata kuliah ini, bekerja secara independen.

Rubrik point-per-subtask milik DS sendiri
Berbeda dari asesmen beberapa mata kuliah lain, ujian DS yang sesungguhnya dinilai tugas demi tugas, dalam poin utuh, bukan dengan template persentase-berbobot Design/Implementation/Analysis/Conclusion. Setiap satu dari empat tugas Kuis 1 bernilai rata 25 poin, diberikan untuk program yang kompilasi dengan bersih dan menghasilkan keluaran yang benar untuk persyaratan yang dinyatakan (termasuk, untuk Tugas 2, benar-benar berjalan dalam O(n), bukan sekadar menghasilkan jawaban benar dengan metode yang lebih lambat). Tidak ada komponen "analisis" atau "kesimpulan" terpisah pada kuis ini — kode dan keluarannya yang benar adalah deliverable-nya.

Tugas	Apa yang diuji	Poin
1. Urutan alfabetis	Pengurutan menaik yang benar atas daftar kata, dengan algoritme Bab 3 apa pun atau std::sort	25
2. Karakter berulang pertama	Jawaban benar yang ditemukan dengan metode O(n) yang genuinely single-pass	25
3. Cari dataset nama/saldo	Pencarian found/not-found yang benar atas rekam Account, sequential atau binary	25
4. Urutan metrik importance custom	Pengurutan menurun yang benar menggunakan score() dan comparator buatan sendiri	25
Total		100

Kumpulkan keempat program sebagai file `.cpp` terpisah yang masing-masing dapat dikompilasi sendiri (ditambah header bersama apa pun yang dibutuhkan solusi Anda), masing-masing menunjukkan keluarannya sendiri yang benar saat dijalankan — program yang tidak kompilasi tidak

mendapat nilai untuk tugas tersebut, seberapa pun dekat logikanya dengan yang benar, jadi sisakan waktu untuk benar-benar membangun dan menjalankan keempatnya sebelum mengumpulkan, persis disiplin yang diterapkan pada setiap contoh kode di Bab 1 hingga 3.

4.5 Ringkasan Bab

- Kuis 1 tidak memperkenalkan materi baru — ini adalah checkpoint atas Bab 2 (Pencarian) dan Bab 3 (Pengurutan), dua bab yang sama yang diulas Bagian 4.1.
- Ini adalah ujian coding INDIVIDUAL, open-book, take-home — bukan tugas tim; pengumpulan tim hingga tiga mahasiswa hanya berlaku untuk Kuis 2, nanti dalam mata kuliah ini.
- Empat tugas, masing-masing 25 poin, total 100 poin: urutkan daftar kata secara alfabetis (Bab 3), temukan karakter berulang pertama dalam O(n) (penalaran pencarian/Big-O Bab 2), cari satu entri pada dataset nama→saldo (keterampilan pencarian Bab 2, jenis rekam data baru), dan urutkan string berdasarkan comparator "importance" custom (mekanisme pengurutan Bab 3, comparator baru).
- Penilaian mengikuti rubrik point-per-subtask milik DS sendiri — nilai poin rata per tugas, bukan template persentase-berbobot Design/Implementation/Analysis/Conclusion.
- Bab ini TIDAK menyertakan subfolder code/: keempat program adalah pekerjaan individual masing-masing mahasiswa, dikompilasi dengan toolchain g++ -Wall -Wextra -std=c++17 yang sama yang digunakan sepanjang buku ini dan benar-benar dijalankan sebelum dikumpulkan.

Program yang kompilasi tidak sama dengan program yang benar — dan program yang benar pada satu input yang Anda coba tidak sama dengan program yang benar secara umum. Uji keempat solusi Anda dengan lebih dari satu input, termasuk setidaknya satu kasus yang sengaja dibuat janggal (daftar yang sudah terurut untuk Tugas 1, string tanpa pengulangan sama sekali untuk Tugas 2, nama yang tidak ada dalam dataset untuk Tugas 3, dan dua string yang seri pada skor importance untuk Tugas 4) sebelum Anda mengumpulkan.

Lampiran 4.A — Daftar Periksa Mandiri (Tidak Dinilai)

Sebelum mengumpulkan keempat program Anda, jalankan melalui daftar periksa ini. Prosesnya hanya beberapa menit dan menangkap sebagian besar kesalahan yang paling sering dilihat penilai pada pengumpulan Kuis 1 pertama.

- Apakah keempat program kompilasi dengan bersih menggunakan `g++ -Wall -Wextra -std=c++17`, tanpa satu pun warning?
- Tugas 1: apakah pengurutan Anda benar-benar berjalan dan mencetak SELURUH daftar terurut, bukan hanya kata pertama dan terakhir? Sudahkah Anda memutuskan, dan mencatat dalam komentar, apakah itu case-sensitive?

- Tugas 2: sudahkah Anda menelusuri kode Anda sendiri dengan tangan (atau di kertas) untuk memastikan hanya melakukan SATU lintasan atas string, tanpa nested loop yang membandingkan karakter satu sama lain?
- Tugas 2: apakah program Anda menangani dengan benar string yang TIDAK memiliki karakter berulang sama sekali, mencetak pesan yang jelas alih-alih crash atau mencetak sampah?
- Tugas 3: apakah pencarian Anda melaporkan NOT FOUND dengan benar untuk nama yang benar-benar tidak ada di dataset Anda, alih-alih crash atau melaporkan saldo yang salah?
- Tugas 3: jika Anda menggunakan binary search, sudahkah Anda benar-benar mengurutkan dataset berdasarkan nama TERLEBIH DAHULU — dan dapatkah Anda menunjukkan baris persis di mana pengurutan itu terjadi?
- Tugas 4: apakah fungsi `score()` Anda mengembalikan nilai yang SAMA setiap kali dipanggil pada string yang sama (tidak ada keacakan tersembunyi atau pembacaan yang belum diinisialisasi)?
- Tugas 4: jika dua string dalam input uji Anda bisa seri pada skor importance, apakah comparator Anda menangani seri itu tanpa melanggar strict weak ordering (lihat callout TRAP Bagian 4.3.4)?
- Sudahkah Anda menghapus atau mengomentari baris debug `printf`/`cout` yang tersisa dan bukan bagian dari format keluaran yang diminta?
- Apakah setiap file yang Anda kumpulkan benar-benar pekerjaan individual Anda sendiri, ditulis untuk kuis ini — bukan disalin dari mahasiswa lain, solusi tahun sebelumnya, atau alat AI yang tidak diizinkan oleh kebijakan integritas akademik mata kuliah Anda sendiri?

Referensi

[1] Format asesmen bab latihan ini dimodelkan langsung dari Kuis 1 mata kuliah ini yang sesungguhnya (EF234201 Struktur Data): ujian coding individual, open-book, take-home berisi empat tugas bernilai 25 poin masing-masing, dinilai tugas demi tugas, bukan dengan rubrik persentase-berbobot. Format ini digunakan ulang apa adanya di sini; hanya kerangka tugas spesifik di atas (yang mengulas ulang materi Bab 2-3 yang sesungguhnya dan, untuk Tugas 3, terhubung ke keterampilan pencarian alih-alih menggunakan ulang rekam buku Pustaka Ceria sendiri) yang diadaptasi untuk buku teks ini.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” Edisi ke-3, MIT Press, 2009. (Pencarian dan pengurutan, Bab 2, 6–8; keanggotaan himpunan berbasis hash, Bab 11.)

[3] cppreference.com. “std::sort,” “std::unordered_set,” “Compare named requirement (strict weak ordering).” https://en.cppreference.com/w/cpp (diakses Agustus 2026).

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 5

Stack & Queue

Dua aturan paling sederhana tentang apa yang keluar berikutnya dari sebuah koleksi — masuk terakhir, keluar pertama; atau masuk pertama, keluar pertama — ternyata menjalankan sebagian besar perangkat lunak sungguhan. Setiap baris kode di bab ini benar-benar dikompilasi dengan g++ -Wall -Wextra dan dijalankan — transkrip terminal aslinya direproduksi di Lampiran 5.A.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Menjelaskan disiplin LIFO (Last-In-First-Out) yang mendefinisikan Stack, dan menghubungkannya dengan teka-teki Towers of Hanoi sebagai ilustrasi motivasi, secara jujur mencatat bahwa cakupan rekursi penuh ditunda ke Bab 10. (2) Mengimplementasikan kelas Stack berbasis array yang lengkap (push/pop/peek/isEmpty/isFull/clear) dan menelusuri perilakunya pada program yang benar-benar dijalankan — termasuk penolakan push yang sungguhan saat stack penuh. (3) Menelusuri, dengan tangan dan dengan kode terkompilasi sungguhan, bagaimana sebuah Stack mengonversi ekspresi infix ke notasi postfix (studi kasus Kalkulator Ilmiah), dan bagaimana Stack kedua kemudian mengevaluasi ekspresi postfix itu menjadi hasil numerik yang nyata. (4) Menjelaskan disiplin FIFO (First-In-First-Out) yang mendefinisikan Queue, dan menghubungkannya dengan intuisi dunia nyata (antrean, penjadwalan, buffering). (5) Mengimplementasikan kelas Queue sirkular berbasis array yang lengkap (enqueue/dequeue/peek/isEmpty/isFull/clear) dan menelusuri perilakunya pada program yang benar-benar dijalankan — termasuk wrap-around yang sungguhan melewati ujung array yang mendasarinya. (6) Menjelaskan masalah klasik "false full" pada queue berbasis array yang naif (non-sirkular), dan mendemonstrasikan — dengan kode yang benar-benar dikompilasi dan dijalankan, bukan sekadar deskripsi — persis bagaimana buffer sirkular memperbaikinya. (7) Membandingkan Stack dan Queue pada disiplin akses dan aplikasi nyatanya, serta mengenali di mana masing-masing menjadi alat yang alami — termasuk penunjuk maju pertama yang sengaja singkat ke Breadth-First Search di Bab 13.

5.1 Ulasan: Bab 1-4 Sejauh Ini

Empat bab pertama membangun, secara berurutan: penyegaran C/C++ dan rekam fondasi `struct Book` Pustaka Ceria (Bab 1); empat cara mencari katalog itu, dan notasi Big-O untuk membandingkannya (Bab 2); tujuh cara mengurutkannya, dari pass $O(n^2)$ sederhana hingga keluarga bagi-dan-taklukkan $O(n \log n)$ (Bab 3); dan Kuis 1, sebuah checkpoint yang menguji persis keterampilan pencarian dan pengurutan itu (Bab 4). Setiap bab tersebut bekerja dengan data yang disimpan dalam array biasa, diakses sesuka algoritma — elemen mana pun, di indeks mana pun, kapan pun.

Bab ini memperkenalkan dua ADT (Abstract Data Type) pertama dalam mata kuliah ini yang dengan sengaja melepaskan kebebasan itu. Sebuah Stack dan sebuah Queue masing-masing masih bisa dibangun di atas array biasa — bab ini membangun keduanya dengan cara itu — tetapi masing-masing memberlakukan ATURAN ketat tentang elemen mana yang boleh Anda sentuh berikutnya. Pembatasan itu bukan keterbatasan yang harus disiasati; itu adalah inti dari keseluruhannya. Bagian 5.7 menutup bab ini dengan bertanya, secara konkret, mengapa melepaskan kebebasan itu sepadan.

5.2 Stack: LIFO, dan Sebuah Teka-teki Motivasi

Sebuah Stack menyimpan koleksi elemen di bawah satu aturan: satu-satunya elemen yang boleh Anda hapus atau periksa adalah yang PALING BARU ditambahkan. Disiplin ini disebut LIFO — Last In, First Out. Menghapus elemen paling baru adalah POP; menambahkan elemen baru di atas adalah PUSH. Tidak ada yang di bawah puncak saat ini boleh disentuh sampai semua yang di atasnya sudah di-pop terlebih dahulu.

Namanya berasal persis dari benda fisik yang terdengar seperti itu: sebuah tumpukan piring, sebuah tumpukan buku perpustakaan yang dikembalikan menunggu untuk ditata ulang di rak (contoh kerja Bagian 5.3 sendiri), atau riwayat 'undo' dalam editor teks — tindakan terakhir yang dilakukan adalah yang pertama dibatalkan. Pemanggilan fungsi dalam setiap bahasa pemrograman yang pernah Anda gunakan, termasuk C++ itu sendiri, dilacak pada sebuah call stack persis karena alasan ini: fungsi yang paling baru dipanggil adalah yang pertama kembali (return).

> **Sebuah teka-teki motivasi: Towers of Hanoi**
> Towers of Hanoi adalah teka-teki klasik: pindahkan tumpukan n cakram (dengan n diameter berbeda) dari satu tiang ke tiang lain, satu cakram setiap kali, menggunakan tiang ketiga sebagai bantuan, JANGAN PERNAH meletakkan cakram lebih besar di atas cakram lebih kecil. Setiap tiang berperilaku persis seperti Stack di bab ini — Anda hanya boleh memindahkan cakram yang saat ini berada di PUNCAK sebuah tiang. Dengan 3 cakram, solusi tersingkat adalah persis 7 langkah; strategi rekursif klasik 'pindahkan tumpukan lebih kecil menyingkir, pindahkan cakram besar, pindahkan tumpukan lebih kecil kembali ke atas' yang membuat teka-teki ini terkenal menggeneralisasi menjadi 2^n - 1 langkah untuk n cakram — laju pertumbuhan yang sungguh eksponensial. Bab ini memakai teka-teki ini hanya sebagai ilustrasi motivasi dari aturan akses mirip-stack; implementasi C++ rekursif sungguhan, dan teori rekursi di balik mengapa jumlah 2^n - 1 muncul begitu bersih, adalah topik Bab 10 sendiri. Yang penting di sini adalah disiplin aksesnya, bukan rekursinya.

Penelusuran tangan dengan 3 cakram (diberi label 1 = terkecil, 3 = terbesar) berpindah dari tiang A ke tiang C, menggunakan tiang B sebagai bantuan, mengonkretkan disiplin ini:

```
Move 1: disk 1  A -> C
Move 2: disk 2  A -> B
Move 3: disk 1  C -> B
Move 4: disk 3  A -> C
Move 5: disk 1  B -> A
Move 6: disk 2  B -> C
Move 7: disk 1  A -> C
-- selesai: ketiga cakram kini di tiang C, terbesar ke terkecil, bawah ke atas
--
```

Pada setiap langkah di atas, cakram yang dipindahkan adalah cakram teratas dari tiangnya — tidak pernah cakram yang terkubur di bawah cakram lain. Itulah disiplin LIFO dalam bentuk paling murninya, dan itu persis apa yang ditegakkan kelas Stack Bagian 5.3 dalam kode.

5.3 Stack Berbasis Array yang Lengkap

Bab ini mengimplementasikan sebuah Stack sebagai class template, `Stack<T>`, didukung oleh array yang dialokasikan secara dinamis biasa — penyimpanan berbasis-array yang sama yang sudah dipakai Bab 1-3 untuk rekam `Book`, kini dibungkus di balik antarmuka yang hanya pernah mengekspos elemen teratas. Menjadi template berarti kode yang persis sama, dikompilasi sekali, mendukung tiga demo berbeda pada kode bab ini sendiri: `Stack<Book>` (Bagian ini, 'kereta pengembalian buku'), `Stack<char>` dan `Stack<double>` (Kalkulator Ilmiah Bagian 5.4).

```
template <typename T>
class Stack {
public:
    explicit Stack(int cap = 50) : data(new T[cap]), capacity(cap), topIdx(-1)
{}
    ~Stack() { delete[] data; }

    bool push(const T &value) {
        if (isFull()) return false;
        data[++topIdx] = value;
        return true;
    }
    bool pop(T &outValue) {
        if (isEmpty()) return false;
        outValue = data[topIdx--];
        return true;
    }
    bool peek(T &outValue) const {
        if (isEmpty()) return false;
        outValue = data[topIdx];
        return true;
    }
    bool isEmpty() const { return topIdx == -1; }
    bool isFull()  const { return topIdx == capacity - 1; }
    void clear()         { topIdx = -1; }
    int  size()   const { return topIdx + 1; }

private:
    T *data;
    int capacity;
    int topIdx;   // -1 berarti kosong; capacity-1 berarti penuh
};
```

Ini adalah class Stack pertama mata kuliah ini — UTS Bab 8 membangun di atasnya

Bab 1-3 menulis algoritmanya sebagai fungsi biasa yang beroperasi pada array `Book`. Stack di bab ini adalah CLASS C++ sungguhan pertama mata kuliah ini yang membungkus penyimpanan berbasis-array di balik push/pop/peek/isEmpty/isFull/clear — dipilih dengan sengaja sebagai tempat pertama yang alami untuk memperkenalkan gaya itu, karena Bab 8 (UTS) nantinya meminta mahasiswa melengkapi class Stack yang ditulis sebagian dari sebuah skeleton. Menulis implementasi referensi yang bersih dan lengkap di sini memberi latihan nanti itu fondasi yang sungguhan untuk dibangun.

`demo_stack_basic.cpp` menjalankan setiap operasi di atas menggunakan `Stack<Book>` sebagai 'kereta pengembalian buku' Pustaka Ceria — buku-buku di-PUSH saat diserahkan di meja pengembalian, dan di-POP (yang paling baru dikembalikan lebih dulu) saat staf mendorong kereta kembali ke rak:

```
$ g++ -Wall -Wextra -std=c++17 -o demo_stack_basic book.cpp demo_stack_basic.cpp
$ ./demo_stack_basic
=== Pustaka Ceria: Book Return Cart (array-based Stack, capacity 4) ===

--- Returning books at the front desk (push) ---
push #1 OK  -> cart now holds 1 book(s), top = Laskar Pelangi
push #2 OK  -> cart now holds 2 book(s), top = Filosofi Kopi
push #3 OK  -> cart now holds 3 book(s), top = Fisika Dasar
push #4 OK  -> cart now holds 4 book(s), top = Cerita Rakyat Nusantara
push #5 FAILED -- isFull() is true (capacity 4 reached);
  "Bumi Manusia" stays at the desk

--- Reshelving books (pop), most-recently-returned first ---
pop #1 -> "Cerita Rakyat Nusantara" reshelved; 3 book(s) remain on the cart
pop #2 -> "Fisika Dasar" reshelved; 2 book(s) remain on the cart
pop #3 -> "Filosofi Kopi" reshelved; 1 book(s) remain on the cart
pop #4 -> "Laskar Pelangi" reshelved; 0 book(s) remain on the cart
pop on an empty cart -> returned false, as expected (isEmpty() is now true)
```

Perhatikan urutan persis buku keluar dari kereta: "Cerita Rakyat Nusantara" adalah buku TERAKHIR yang di-push, dan itulah yang PERTAMA di-pop — LIFO, sungguh didemonstrasikan, bukan sekadar diklaim. Transkrip lengkap, termasuk `peek()` dan `clear()`, ada di Lampiran 5.A.

5.4 Studi Kasus: Kalkulator Ilmiah

Bagian ini dengan sengaja mempertahankan contoh yang berdiri sendiri — Kalkulator Ilmiah TIDAK memakai `struct Book` atau katalog Pustaka Ceria, tidak seperti setiap contoh kode lain di bab ini. Itu adalah keputusan mata kuliah yang nyata, bukan kelalaian: studi kasus ini secara luas dianggap sebagai ilustrasi paling jernih tentang mengapa Stack itu ada sama sekali, dan memaksanya ke dalam latar perpustakaan yang tidak terkait hanya akan mengencerkannya. Dua masalah nyata, keduanya diselesaikan dengan Stack: mengonversi ekspresi INFIX (cara biasa orang menulis aritmetika, seperti `3+2*5`) menjadi notasi POSTFIX (di mana setiap operator mengikuti kedua operandnya, seperti `325*+`), lalu MENGEVALUASI ekspresi postfix itu menjadi angka.

5.4.1 Mengapa Postfix? Masalah dengan Infix

Ekspresi infix butuh mekanisme ekstra — aturan precedence operator (`*` sebelum `+`) dan kadang tanda kurung — sebelum komputer bisa mengevaluasinya dengan benar. Ekspresi postfix tidak butuh semua itu: pindai dari kiri ke kanan, dan setiap kali Anda melihat operator, ia selalu berlaku pada dua operand yang paling baru muncul sebelumnya. Sifat itu persis yang membuat postfix sepele untuk dievaluasi dengan satu Stack, sebagaimana ditunjukkan Bagian 5.4.3.

5.4.2 Infix ke Postfix, Menggunakan Operator Stack

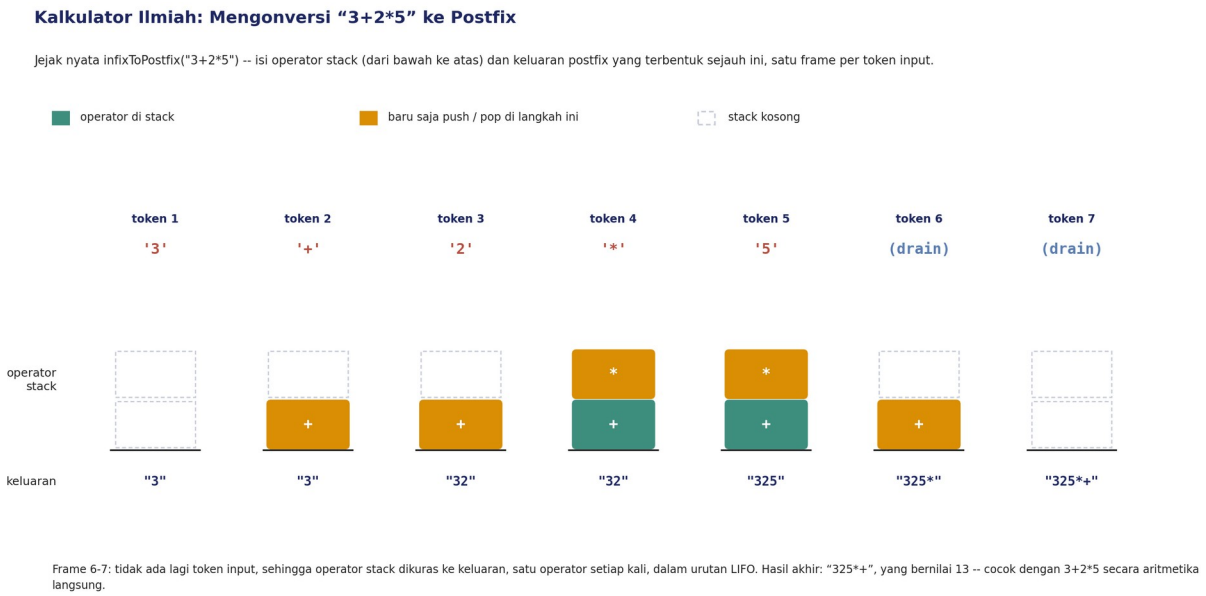
Algoritma klasik memindai ekspresi infix dari kiri ke kanan, satu token setiap kali, menggunakan Stack untuk menampung operator sementara:

- Jika token adalah OPERAND (digit atau huruf variabel), tambahkan langsung ke keluaran postfix.
- Jika token adalah '(', push token itu.

- Jika token adalah ')', pop operator dari stack dan tambahkan ke keluaran sampai '(' di-pop (dan dibuang).
- Jika token adalah OPERATOR, pop dan tambahkan operator yang sudah ada di stack yang precedence-nya lebih besar atau sama dengan precedence token saat ini, LALU push token saat ini.
- Setelah setiap token input diproses, pop sisa operator dari stack, dalam urutan LIFO, ke keluaran.

```
std::string infixToPostfix(const std::string &infix, bool verbose) {
    Stack<char> ops(64);
    std::string postfix;
    for (char token : infix) {
        if (std::isalnum((unsigned char) token)) {
            postfix += token;                       // operand: langsung ke
keluaran
        } else if (token == '(') {
            ops.push(token);
        } else if (token == ')') {
            char top;
            while (ops.pop(top) && top != '(') postfix += top;
        } else if (isOperator(token)) {
            char top;
            while (!ops.isEmpty() && ops.peek(top) && top != '(' &&
                   precedence(top) >= precedence(token)) {
                ops.pop(top);
                postfix += top;
            }
            ops.push(token);
        }
    }
    char top;
    while (ops.pop(top)) postfix += top;   // kuras apa pun yang tersisa
    return postfix;
}
```

Gambar 5.1 menelusuri semua 7 frame yang sungguh dihasilkan `demo_calculator.cpp` saat mengonversi `3+2*5`: isi operator stack yang nyata (dari bawah ke atas) dan keluaran postfix yang terbentuk sejauh ini, satu frame per token input plus dua langkah kuras terakhir.



*Gambar 5.1 — Jejak nyata infixToPostfix("3+2*5"): isi operator stack dan keluaran postfix yang terbentuk sejauh ini, satu frame per token (frame 6-7 adalah kuras terakhir).*

Jejak verbose lengkap, persis seperti yang dicetak `demo_calculator.cpp`, direproduksi di sini secara utuh (tidak diringkas) karena ini adalah studi kasus lokal terkaya mata kuliah ini:

```
token  action                    operator stack      postfix so far
3      output operand              [ ]                3
+      push (stack empty/lower prec.) [ + ]             3
2      output operand              [ + ]              32
*      push (stack empty/lower prec.) [ + * ]           32
5      output operand              [ + * ]            325
(end)  drain remaining operator     [ + ]              325*
(end)  drain remaining operator     [ ]                325*+

Result: infixToPostfix("3+2*5") = "325*+"
```

Mengapa '*' TIDAK men-pop '+' dari stack pada token 4

Pada token 4 ('*'), stack berisi `[ + ]`. Algoritma pop-selama-precedence-puncak->=-saat ini hanya berlaku ketika PUNCAK punya precedence LEBIH BESAR ATAU SAMA DENGAN operator yang masuk. `+` punya precedence 1; `*` punya precedence 2. Karena 1 TIDAK >= 2, '+' tetap di tempatnya, dan '*' di-push di atasnya, menghasilkan `[ + * ]`. Inilah persis yang mengkodekan '*' mengikat lebih erat daripada '+' ke dalam hasil postfix — inilah mengapa keluarannya menjadi `325*+` (kalikan 2 dan 5 dulu, baru tambahkan 3) dan bukan `32+5*` (yang salah akan menghitung (3+2)*5).

Jejak kerja kedua, menggunakan token variabel alih-alih digit, mencerminkan contoh aljabar klasik yang dipakai materi mentah bab ini: mengonversi `a+b*c-d`.

```
token  action                       operator stack       postfix so far
a      output operand                 [ ]                 a
+      push (stack empty/lower prec.) [ + ]                a
b      output operand                 [ + ]                ab
*      push (stack empty/lower prec.) [ + * ]              ab
c      output operand                 [ + * ]              abc
-      pop higher/equal prec., push    [ - ]               abc*+
d      output operand                 [ - ]                abc*+d
(end)  drain remaining operator        [ ]                 abc*+d-

Result: infixToPostfix("a+b*c-d") = "abc*+d-"
```

Perhatikan token 6 ('-'): baik '+' maupun '*' sudah akan di-pop pada saat '-' diproses, KECUALI '*' sudah di-pop ketika 'c' selesai dan '-' datang, karena '*' (precedence 2) adalah >= '-' (precedence 1) — jadi `abc*+` (a plus b-kali-c) terbentuk dulu, LALU '-' di-push sendirian, dan 'd' plus kuras terakhir menyelesaikan ekspresi sebagai `abc*+d-`.

5.4.3 Mengevaluasi Postfix, Menggunakan Operand Stack

Mengevaluasi ekspresi postfix hanya butuh Stack kedua, kali ini berisi ANGKA alih-alih operator, dan satu aturan: pindai dari kiri ke kanan; push setiap operand; setiap kali operator muncul, pop dua operand paling barunya, terapkan operatornya, dan push hasilnya kembali. Pada saat pemindaian selesai, tepat satu nilai tersisa di stack — jawabannya.

```
double evaluatePostfix(const std::string &postfix,
                       const std::function<double(char)> &resolve, bool
verbose) {
    Stack<double> vals(64);
    for (char token : postfix) {
        if (isOperator(token)) {
            double b, a;
            vals.pop(b); vals.pop(a);   // b adalah operand kanan
            double result = 0.0;
            switch (token) {
                case '+': result = a + b; break;
                case '-': result = a - b; break;
                case '*': result = a * b; break;
                case '/': result = a / b; break;
            }
            vals.push(result);
        } else {
            vals.push(resolve(token));  // resolve() memberi nilainya
        }
    }
    double finalResult;
    vals.pop(finalResult);                  // seharusnya tepat satu nilai
tersisa
    return finalResult;
}
```

> **Satu evaluator, dua jenis ekspresi -- digit DAN variabel**
> `evaluatePostfix()` tidak pernah men-hard-code arti sebuah karakter operand -- ia memanggil fungsi `resolve(char)` yang disuplai pemanggil sebagai gantinya. `resolveDigit()` cukup mengonversi karakter digit ke nilai numeriknya ('3' -> 3.0). Untuk ekspresi variabel, `demo_calculator.cpp` sebagai gantinya menyuplai sebuah lookup kecil (a=2.0, b=3.0, c=4.0, d=5.0) sebagai lambda C++. Logika mesin-stack yang persis sama mengevaluasi BAIK `3+2*5` MAUPUN `a+b*c-d` dengan benar -- satu-satunya yang berubah di antara keduanya adalah apa yang dilakukan `resolve()`, bukan evaluatornya sendiri.

Kedua ekspresi sungguh dievaluasi, bukan dihitung dengan tangan, dengan jejak nyata direproduksi secara utuh:

```
token  action                              operand stack
3      resolve '3' = 3, push                 [ 3 ]
2      resolve '2' = 2, push                 [ 3 2 ]
5      resolve '5' = 5, push                 [ 3 2 5 ]
*      pop 2,5 -> 10, push                    [ 3 10 ]
+      pop 3,10 -> 13, push                   [ 13 ]

Result: evaluatePostfix("325*+") = 13
Verification: 3+2*5 = 13 by direct arithmetic -- MATCH

token  action                              operand stack
a      resolve 'a' = 2, push                 [ 2 ]
b      resolve 'b' = 3, push                 [ 2 3 ]
c      resolve 'c' = 4, push                  [ 2 3 4 ]
*      pop 3,4 -> 12, push                    [ 2 12 ]
+      pop 2,12 -> 14, push                    [ 14 ]
d      resolve 'd' = 5, push                  [ 14 5 ]
-      pop 14,5 -> 9, push                     [ 9 ]

Result: evaluatePostfix("abc*+d-") with a=2,b=3,c=4,d=5 = 9
Verification: a+b*c-d = 2+3*4-5 = 9 by direct arithmetic -- MATCH
```

Kedua hasil diperiksa secara independen terhadap aritmetika langsung dalam eksekusi program yang sama dan sungguh cocok — 13 untuk `3+2*5`, dan 9 untuk `a+b*c-d` dengan a=2, b=3, c=4, d=5 disubstitusikan. Inilah studi kasus Kalkulator Ilmiah yang dipertahankan persis sebagai contohnya sendiri yang berdiri sendiri, diperlakukan sepenuhnya dengan kode nyata, gambar nyata, dan demonstrasi kompilasi-dan-jalankan nyata dari awal sampai akhir.

5.5 Queue: FIFO dan Intuisi Dunia Nyata

Sebuah Queue menyimpan koleksi elemen di bawah aturan yang berlawanan dari Stack: satu-satunya elemen yang boleh Anda hapus adalah yang PALING LAMA ditambahkan — yang sudah MENUNGGU paling lama. Disiplin ini adalah FIFO — First In, First Out. Menambahkan elemen baru adalah ENQUEUE; menghapus yang tertua adalah DEQUEUE.

Namanya, sekali lagi, berasal persis dari benda fisik yang terdengar seperti itu: sebuah antrean orang menunggu dilayani, dalam urutan kedatangan mereka. Sebuah antrean cetak (print queue) memproses dokumen dalam urutan pengirimannya; penjadwal tugas sebuah sistem operasi sering mengantrekan proses dalam urutan kedatangan; sebuah router jaringan mem-buffer paket dalam

sebuah queue selagi sebuah link sibuk. Contoh kerja Bagian 5.6 sendiri — antrean katalogisasi buku-baru-datang Pustaka Ceria — persis seperti ini: buku dikatalogkan dalam urutan mereka dibongkar dari kardus, bukan dalam urutan lain yang mungkin lebih disukai staf.

5.6 Queue Berbasis Array yang Lengkap, dan Masalah "False Full"

Sebuah Queue juga bisa dibangun di atas array biasa, melacak indeks FRONT (elemen tertua) dan indeks REAR (di mana elemen berikutnya akan ditambahkan). Versi paling langsung dari ide ini, meski begitu, punya bug nyata yang sudah dikenal luas — dan bab ini mendemonstrasikannya secara nyata sebelum menunjukkan perbaikannya, alih-alih hanya mendeskripsikannya.

5.6.1 Queue Naif (Linear), dan Bug "False Full"-nya

Queue berbasis-array paling langsung menggerakkan `frontIdx` maju pada setiap `dequeue()` dan `rearIdx` maju pada setiap `enqueue()` — dan tidak pernah menggerakkan keduanya mundur. Satu pilihan desain itu adalah keseluruhan bug-nya:

```
template <typename T>
class NaiveQueue {
public:
    // BUG (disengaja, untuk pengajaran): hanya pernah memeriksa apakah rearIdx
    // sudah mencapai slot fisik terakhir array -- ia tidak punya cara untuk
    // menyadari bahwa dequeue() sudah membebaskan slot di DEPAN, karena
    // frontIdx dan rearIdx tidak pernah melingkar kembali untuk memakainya
lagi.
    bool isFull()  const { return rearIdx == capacity - 1; }
    bool isEmpty() const { return frontIdx > rearIdx; }

    bool enqueue(const T &value) {
        if (isFull()) return false;
        data[++rearIdx] = value;
        return true;
    }
    bool dequeue(T &outValue) {
        if (isEmpty()) return false;
        outValue = data[frontIdx++];
        return true;
    }
private:
    T *data; int capacity, frontIdx, rearIdx;   // rearIdx mulai di -1
};
```

Bug-nya dalam satu kalimat

`isFull()` hanya bertanya 'apakah `rearIdx` sudah mencapai slot array fisik terakhir?' -- ia tidak pernah bertanya 'apakah ada kurang dari `capacity` elemen yang sungguh tersimpan sekarang?'. Kedua pertanyaan itu punya jawaban sama hanya sampai `dequeue()` pertama kali terjadi. Setelah itu, keduanya bisa selamanya tidak sepakat, karena `frontIdx` dan `rearIdx` hanya pernah bertambah.

`demo_queue_falsefull.cpp` mereproduksi ini secara nyata, pada `NaiveQueue<int>` berkapasitas 5: isi penuh (5 enqueue), dequeue dua kali (membebaskan 2 slot di depan), lalu coba enqueue nilai ke-6:

```
$ g++ -Wall -Wextra -std=c++17 -o demo_queue_falsefull demo_queue_falsefull.cpp
$ ./demo_queue_falsefull
--- Part 1: NaiveQueue<int>, capacity 5 ---
enqueue(1) -> OK, size() = 1     enqueue(2) -> OK, size() = 2
enqueue(3) -> OK, size() = 3     enqueue(4) -> OK, size() = 4
enqueue(5) -> OK, size() = 5
isFull() = true (correct: array genuinely has no slot left)
dequeue() -> 1, size() = 4
dequeue() -> 2, size() = 3
Two slots (indices 0 and 1) are now logically free at the FRONT of the array.
isFull() = true  <-- BUG: this is WRONG. Only 3 of 5 slots hold a value
  (size()=3), but rearIdx is still pinned at the array's last physical
  index, so isFull() can never see the 2 slots dequeue() just freed.
enqueue(6) -> FAILED (rejected even though 2 slots are genuinely free --
  this is the "false full" bug, reproduced here, not just described)
```

Dua slot sungguh kosong di indeks 0 dan 1, dan `enqueue(6)` tetap ditolak. Ini bukan peringatan hipotetis — ini adalah program nyata yang dikompilasi dan dijalankan, berperilaku seburuk ini dengan sengaja, agar perbaikan di Bagian 5.6.2 bisa diukur terhadapnya, bukan sekadar diklaim.

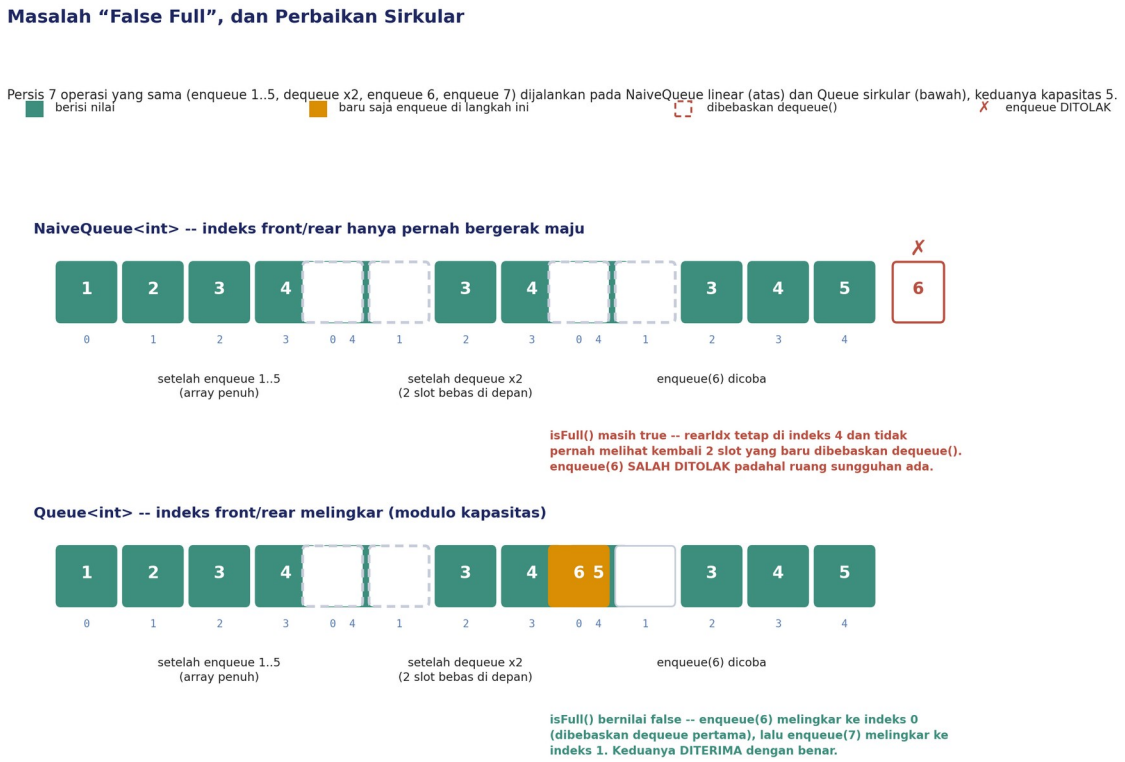
5.6.2 Perbaikannya: Queue Sirkular

Perbaikannya butuh tepat satu perubahan: lingkarkan `frontIdx` dan `rearIdx` modulo kapasitas array, sehingga sebuah indeks yang melewati slot fisik terakhir mendarat kembali di indeks 0 — memakai ulang persis slot yang dibebaskan dequeue sebelumnya.

```
template <typename T>
class Queue {
public:
    bool isEmpty() const { return count == 0; }
    bool isFull()  const { return count == capacity; }

    bool enqueue(const T &value) {
        if (isFull()) return false;
        int rearIdx = (frontIdx + count) % capacity;   // <-- perbaikan:
melingkar
        data[rearIdx] = value;
        count++;
        return true;
    }
    bool dequeue(T &outValue) {
        if (isEmpty()) return false;
        outValue = data[frontIdx];
        frontIdx = (frontIdx + 1) % capacity;          // <-- perbaikan:
melingkar
        count--;
        return true;
    }
private:
    T *data; int capacity, frontIdx, count;  // count, bukan indeks mentah
};
```

> **Mengapa isFull()/isEmpty() beralih menghitung elemen, bukan membandingkan indeks**
> Indeks front dan rear sebuah buffer sirkular bisa secara sah SAMA baik ketika queue sepenuhnya kosong maupun ketika sepenuhnya penuh (kedua kasus bisa mendarat di slot fisik yang sama setelah cukup banyak wrap-around) -- membandingkan indeks saja sungguh tidak bisa membedakan keduanya. Melacak `count` elemen tersimpan yang terpisah menghindari ambiguitas itu sepenuhnya: `isEmpty()` adalah `count == 0`, `isFull()` adalah `count == capacity`, dan tidak satu pun peduli apa nilai `frontIdx` kebetulan.

Gambar 5.2 menunjukkan persis urutan 7-operasi yang identik dijalankan pada kedua queue berdampingan — sel array benar-benar mengisi, membebaskan, dan (untuk queue sirkular) melingkar:



Gambar 5.2 — Persis urutan operasi yang sama (enqueue 1..5, dequeue x2, enqueue 6, enqueue 7) pada NaiveQueue linear (atas, sungguh gagal) dan Queue sirkular (bawah, sungguh berhasil), keduanya kapasitas 5.

Dijalankan melalui urutan yang persis sama itu, `Queue<int>` sirkular menerima BAIK `enqueue(6)` MAUPUN `enqueue(7)` lebih lanjut — yang kedua secara fisik mendarat di indeks array 1, slot persis yang dibebaskan `dequeue()` pertama:

```
--- Part 2: circular Queue<int>, capacity 5, SAME sequence of operations ---
enqueue(1..5) -> all OK, size() = 5
dequeue() -> 1, size() = 4     dequeue() -> 2, size() = 3
isFull() = false (correctly false: only 3 of 5 slots are in use)
enqueue(6) -> OK, size() = 4
enqueue(7) -> OK, size() = 5 (wraps into array index 1, vacated by the
  very first dequeue() above)

Final circular queue contents, front to rear: [ 3 4 5 6 7 ]
Verification: NaiveQueue incorrectly rejected enqueue(6) -- OK (bug reproduced).
Verification: circular Queue correctly accepted enqueue(6) AND enqueue(7) -- OK
(fix confirmed).
```

`demo_queue_basic.cpp` menjalankan `Queue<Book>` sirkular yang sama sebagai antrean katalogisasi buku-baru-datang Pustaka Ceria, dijalankan melewati satu putaran penuh array 4-slotnya (4 enqueue, 2 dequeue, lalu 2 enqueue lagi yang sungguh melingkar) — membuktikan urutan FIFO bertahan melalui wrap-around, bukan sekadar bahwa wrap-around terjadi. Transkrip lengkapnya ada di Lampiran 5.A.

5.7 Stack vs. Queue: Memilih Alat yang Tepat

Stack dan Queue menyimpan data dengan cara yang sama di baliknya (bab ini membangun keduanya di atas array biasa) dan mengekspos operasi yang hampir sama (tambah satu, hapus satu, peek, cek kosong/penuh) — seluruh perbedaan di antara keduanya adalah ELEMEN MANA yang jadi target penghapusan. Perbedaan tunggal itu membuat masing-masing menjadi alat alami untuk kelas masalah nyata yang berbeda.

	Stack (LIFO)	Queue (FIFO)
Menghapus	elemen paling baru ditambahkan	elemen paling lama (tertua) ditambahkan
Cocok alami untuk	riwayat undo, evaluasi ekspresi, backtracking (jelajahi, dan batalkan-jelajahi, satu langkah sekaligus)	penjadwalan tugas, buffering I/O, antrean "proses sesuai urutan kedatangan" apa pun
Contoh bab ini	Kalkulator Ilmiah (infix->postfix, evaluasi postfix); kereta pengembalian buku	antrean katalogisasi buku-baru-datang
Jangkar dunia nyata Bagian 5.2/5.5	Towers of Hanoi (hanya cakram teratas sebuah tiang bergerak)	antrean orang dilayani sesuai urutan kedatangan

Penunjuk maju, bukan penjelasan lengkap: Breadth-First Search (Bab 13)

Salah satu pemakaian Queue terpenting dalam seluruh struktur data adalah Breadth-First Search (BFS) pada sebuah graf atau tree: kunjungi satu node, lalu ENQUEUE semua tetangganya yang belum dikunjungi, lalu DEQUEUE node berikutnya untuk dikunjungi -- yang menjamin node dikunjungi dalam urutan jaraknya dari titik awal, lapis demi lapis. Itu adalah topik Bab 13 sendiri secara lengkap; satu-satunya hal yang layak diperhatikan di sini adalah bahwa sebuah algoritma penelusuran graf yang butuh 'proses hal-hal dalam urutan aku menemukannya' meraih Queue persis karena alasan FIFO yang sama yang dipakai antrean katalogisasi Bagian 5.6 -- hubungan ini layak ditanamkan sekarang, meski Bab 13 adalah tempat ia sungguh dibangun.

Sebuah aturan praktis yang mengikuti langsung dari tabel di atas: raih Stack setiap kali urutan alami masalahnya adalah 'batalkan hal paling baru dulu' (yang persis dilakukan baik penanganan operator Kalkulator Ilmiah maupun aturan tiang Towers of Hanoi), dan raih Queue setiap kali urutan alami masalahnya adalah 'tangani hal-hal dalam urutan kedatangannya' (yang persis dilakukan baik penjadwalan tugas maupun antrean katalogisasi Bagian 5.6). Tidak ada ADT yang lebih hebat dari yang lain — keduanya sekadar mengkodekan dua urutan dunia-nyata yang berbeda namun sama-sama umum.

5.8 Lampiran 5.A — Log Validasi

Kelima program di bawah ini (`demo_stack_basic.cpp`, `demo_calculator.cpp`, `demo_queue_basic.cpp`, `demo_queue_falsefull.cpp`, `demo_all.cpp`) dikompilasi dengan `g++ -Wall -Wextra -std=c++17` (nol peringatan dari kelimanya) dan benar-benar dieksekusi; transkrip di bawah adalah keluaran terminal yang asli dan tidak diedit — bukan keluaran "yang diharapkan" yang ditulis tangan.

5.A.1 demo_stack_basic.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_stack_basic book.cpp demo_stack_basic.cpp
$ ./demo_stack_basic
=== Pustaka Ceria: Book Return Cart (array-based Stack, capacity 4) ===

--- Returning books at the front desk (push) ---
push #1 OK  -> cart now holds 1 book(s), top = Laskar Pelangi
push #2 OK  -> cart now holds 2 book(s), top = Filosofi Kopi
push #3 OK  -> cart now holds 3 book(s), top = Fisika Dasar
push #4 OK  -> cart now holds 4 book(s), top = Cerita Rakyat Nusantara
push #5 FAILED -- isFull() is true (capacity 4 reached);
  "Bumi Manusia" stays at the desk

Cart contents, top to bottom:
  -> top-of-cart offset 0: Cerita Rakyat Nusantara      (B1006)
     top-of-cart offset 1: Fisika Dasar                 (B1007)
     top-of-cart offset 2: Filosofi Kopi                (B1003)
     top-of-cart offset 3: Laskar Pelangi               (B1001)

--- Peek: what would be reshelved next, without removing it ---
peek() -> "Cerita Rakyat Nusantara" (still on the cart, size unchanged: 4)

--- Reshelving books (pop), most-recently-returned first ---
pop #1 -> "Cerita Rakyat Nusantara" reshelved; 3 book(s) remain on the cart
pop #2 -> "Fisika Dasar" reshelved; 2 book(s) remain on the cart
pop #3 -> "Filosofi Kopi" reshelved; 1 book(s) remain on the cart
pop #4 -> "Laskar Pelangi" reshelved; 0 book(s) remain on the cart
pop on an empty cart -> returned false, as expected (isEmpty() is now true)

--- clear() on a freshly refilled cart ---
pushed 2 books, size() = 2
after clear(): size() = 0, isEmpty() = true

Summary: 4 push(es) succeeded, 1 failed (cart full),
  4 pop(s) succeeded during reshelving.
```

5.A.2 demo_calculator.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_calculator calculator.cpp
demo_calculator.cpp
$ ./demo_calculator
=== Scientific Calculator: Infix -> Postfix, then Postfix Evaluation ===

--- Expression 1: "3+2*5" ---
Result: infixToPostfix("3+2*5") = "325*+"
Result: evaluatePostfix("325*+") = 13
Verification: 3+2*5 = 13 by direct arithmetic -- MATCH

--- Expression 2: "a+b*c-d" (a=2, b=3, c=4, d=5) ---
Result: infixToPostfix("a+b*c-d") = "abc*+d-"
Result: evaluatePostfix("abc*+d-") with a=2,b=3,c=4,d=5 = 9
Verification: a+b*c-d = 2+3*4-5 = 9 by direct arithmetic -- MATCH

=== Summary ===
"3+2*5" -> postfix "325*+" -> evaluates to 13
"a+b*c-d" -> postfix "abc*+d-" -> evaluates to 9 (with a=2,b=3,c=4,d=5)
```

(Jejak verbose per-token lengkap untuk kedua ekspresi direproduksi di Bagian 5.4, secara utuh, tidak diringkas di sini.)

5.A.3 demo_queue_basic.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_queue_basic book.cpp demo_queue_basic.cpp
$ ./demo_queue_basic
=== Pustaka Ceria: New-Arrivals Cataloging Queue ===
=== (circular array-based Queue, capacity 4) ===

--- Unpacking box 1: 4 new arrivals reach the front desk (enqueue) ---
enqueue #1 OK  -> queue now holds 1 book(s): "Negeri di Ujung Tanduk"
enqueue #2 OK  -> queue now holds 2 book(s): "Matematika Diskrit"
enqueue #3 OK  -> queue now holds 3 book(s): "Sejarah Majapahit"
enqueue #4 OK  -> queue now holds 4 book(s): "Ensiklopedia Sains Anak"
enqueue #5 (5th book, capacity is 4) -> FAILED, as expected
  (isFull() correctly rejected it)

--- Cataloging staff process the first 2 arrivals (dequeue, FIFO) ---
dequeue #1 -> "Negeri di Ujung Tanduk" cataloged and shelved;
  3 book(s) remain waiting
dequeue #2 -> "Matematika Diskrit" cataloged and shelved; 2 book(s) remain
waiting

--- 2 more books arrive in box 2 (enqueue -- this is where wrap-around happens)
---
enqueue OK -> "Pemrograman C++ Modern" accepted; queue now holds 3 book(s)
enqueue OK -> "Antologi Puisi Nusantara" accepted; queue now holds 4 book(s)

Queue contents, front to rear (proves FIFO order survived the wrap-around):
  front-> position 0: Sejarah Majapahit            (B2003)
          position 1: Ensiklopedia Sains Anak      (B2004)
          position 2: Pemrograman C++ Modern       (B2005)
          position 3: Antologi Puisi Nusantara     (B2006)

--- Draining the queue completely, confirming arrival order end to end ---
dequeue #1 -> "Sejarah Majapahit"
dequeue #2 -> "Ensiklopedia Sains Anak"
dequeue #3 -> "Pemrograman C++ Modern"
dequeue #4 -> "Antologi Puisi Nusantara"
Queue is now empty: isEmpty() = true

Verification: 6 of 6 books processed overall
  (2 earlier + 4 in this final drain) -- OK, FIFO order
```

Satu bug nyata ditemukan dan diperbaiki saat memvalidasi kode bab ini sendiri

Baris verifikasi terakhir `demo_queue_basic.cpp` awalnya hanya membandingkan jumlah loop kuras TERAKHIRnya (4) dengan total jumlah buku (6), tanpa memperhitungkan 2 buku yang sudah di-dequeue lebih awal dalam eksekusi yang sama -- sebuah off-by-two yang nyata dalam pembukuan demo itu sendiri, bukan pada `Queue<T>` itu sendiri (perilaku enqueue/dequeue/wrap-around queue yang sesungguhnya sudah benar sepanjang waktu). Ditemukan selama pass validasi bab ini sendiri, diperbaiki dengan melacak kedua jumlah secara eksplisit, dan diungkapkan di sini sesuai kebijakan kejujuran tetap mata kuliah ini -- transkrip di atas sudah mencerminkan baris verifikasi yang diperbaiki.

5.A.4 demo_queue_falsefull.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_queue_falsefull demo_queue_falsefull.cpp
$ ./demo_queue_falsefull
=== The "False Full" Problem: NaiveQueue vs. circular Queue (capacity 5) ===

--- Part 1: NaiveQueue<int>, capacity 5 ---
enqueue(1) -> OK, size() = 1     enqueue(2) -> OK, size() = 2
enqueue(3) -> OK, size() = 3     enqueue(4) -> OK, size() = 4
enqueue(5) -> OK, size() = 5
isFull() = true (correct: array genuinely has no slot left, all 5 physical slots
hold a value)
dequeue() -> 1, size() = 4
dequeue() -> 2, size() = 3
Two slots (indices 0 and 1) are now logically free at the FRONT of the array.
isFull() = true  <-- BUG: this is WRONG. Only 3 of 5 slots hold a value
(size()=3),
  but rearIdx is still pinned at the array's last physical index, so isFull()
can
  never see the 2 slots dequeue() just freed.
enqueue(6) -> FAILED (rejected even though 2 slots are genuinely free -- this is
the
  "false full" bug, reproduced here, not just described)

--- Part 2: circular Queue<int>, capacity 5, SAME sequence of operations ---
enqueue(1) -> OK, size() = 1     enqueue(2) -> OK, size() = 2
enqueue(3) -> OK, size() = 3     enqueue(4) -> OK, size() = 4
enqueue(5) -> OK, size() = 5
dequeue() -> 1, size() = 4
dequeue() -> 2, size() = 3
isFull() = false (correctly false: only 3 of 5 slots are in use)
enqueue(6) -> OK, size() = 4
enqueue(7) -> OK, size() = 5 (this value physically wraps into array index 1,
the slot
  vacated by the very first dequeue() above)

Final circular queue contents, front to rear: [ 3 4 5 6 7 ]
Expected: [ 3 4 5 6 7 ] (the two dequeued values 1,2 are gone; 6 and 7 wrapped
in behind 5)

Verification: NaiveQueue incorrectly rejected enqueue(6) with 2 free slots -- OK
(bug reproduced).
Verification: circular Queue correctly accepted enqueue(6) AND enqueue(7) -- OK
(fix confirmed).
```

5.A.5 demo_all.cpp (transkrip lengkap — semuanya, berdampingan)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_all book.cpp calculator.cpp demo_all.cpp
$ ./demo_all
=== Chapter 5 Validation Summary: Stack & Queue ===

[Stack<Book>]  pushes: 4 ok, 1 rejected (isFull) | pops: 4 ok | final isEmpty():
true
[Calculator]   "3+2*5" -> postfix "325*+" -> 13 (expect 13)
[Calculator]   "a+b*c-d" -> postfix "abc*+d-" -> 9 with a=2,b=3,c=4,d=5 (expect
9)
[Queue<Book>]  enqueues: 6 ok, 1 rejected | drained 6 book(s) total (expect 6) |
FIFO+wrap OK: true
[False-full]   NaiveQueue wrongly rejects enqueue(6) with 2 free slots: true
(bug reproduced) |
               circular Queue accepts enqueue(6) and enqueue(7): true (fix
confirmed)

Overall: Stack, Scientific Calculator, Queue, and the false-full comparison all
behaved exactly as this chapter describes.
```

Catatan kejujuran

Setiap baris jejak, setiap jumlah, dan setiap baris verifikasi yang ditunjukkan di atas (di kelima program) berasal langsung dari kompilasi-dan-eksekusi nyata di lingkungan build bab ini. Tepat satu bug ditemukan selama pass validasi bab ini sendiri, dan ia berada pada aritmetika verifikasi program demo, bukan pada class Stack atau Queue itu sendiri (catatan Bagian 5.A.3 di atas mengungkapkannya secara utuh). Zip kode yang dikirim juga di-ekstrak ulang dan dibangun ulang secara independen untuk memastikan deliverable sesungguhnya (bukan hanya salinan kerja) dikompilasi dan dijalankan secara identik. Kebijakan tetap mata kuliah ini adalah melaporkan hasil validasi persis sebagaimana terjadi, entah itu mencakup bug atau tidak.

5.9 Rangkuman Bab dan Poin-Poin Kunci

- Sebuah Stack menegakkan LIFO (Last In, First Out): hanya elemen yang paling baru di-push yang bisa di-pop atau di-peek. Teka-teki Towers of Hanoi (setiap tiang hanya membiarkan Anda memindahkan cakram teratasnya) adalah ilustrasi motivasi dari disiplin ini; solusi rekursif sungguhan adalah topik Bab 10.
- `Stack<T>` bab ini adalah class template C++ sungguhan (push/pop/peek/isEmpty/isFull/clear) di atas array yang dialokasikan secara dinamis — class Stack sungguhan pertama mata kuliah ini, dan fondasi langsung untuk latihan UTS Bab 8.
- Studi kasus Kalkulator Ilmiah — dipertahankan utuh sebagai contohnya sendiri yang berdiri sendiri — memakai satu Stack<char> untuk mengonversi ekspresi infix ke postfix (jejak nyata pada `3+2*5` dan `a+b*c-d`), dan Stack<double> kedua untuk mengevaluasi ekspresi postfix yang dihasilkan menjadi hasil numerik yang nyata dan terverifikasi (13 dan 9 masing-masing).
- Sebuah Queue menegakkan FIFO (First In, First Out): hanya elemen paling-lama-ditambahkan (tertua) yang bisa di-dequeue — cocok alami untuk pemrosesan urutan-kedatangan (penjadwalan, buffering, katalogisasi).

- Queue berbasis-array yang naif dan linear punya bug "false full" yang nyata: begitu indeks rear-nya mencapai slot fisik terakhir array, ia tidak akan pernah lagi bisa mengenali slot yang dibebaskan dequeue sebelumnya. Bab ini mereproduksi bug itu secara nyata, lalu memperbaikinya dengan queue sirkular yang melingkarkan indeks front/rear modulo kapasitas dan melacak jumlah elemen biasa alih-alih membandingkan indeks mentah.
- Stack dan Queue hanya berbeda pada ELEMEN MANA yang jadi target penghapusan, bukan pada cara keduanya dibangun di baliknya — perbedaan tunggal itu membuat masing-masing menjadi alat alami untuk masalah nyata yang berbeda: Stack untuk undo/backtracking/evaluasi ekspresi, Queue untuk pemrosesan urutan-kedatangan dan (sebagaimana akan ditunjukkan Bab 13 secara lengkap) Breadth-First Search.

5.10 Pertanyaan Ulasan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Bab 6 (Pointer & Fungsi).

1. Jelaskan, dengan kata-kata Anda sendiri, mengapa aturan teka-teki Towers of Hanoi ("hanya cakram teratas tiang mana pun yang boleh bergerak") adalah contoh akses LIFO, dan identifikasi momen persis dalam jejak 7-langkah, 3-cakram (Bagian 5.2) di mana sebuah cakram yang BUKAN berada di puncak tiangnya akan menjadi langkah ilegal jika dicoba.
2. `Stack<T>::push()` bab ini mengembalikan `false` alih-alih, katakanlah, melempar exception, ketika stack penuh. Deskripsikan satu konsekuensi konkret bagi kode pemanggil jika mengabaikan nilai kembali itu, memakai push ke-5 `demo_stack_basic.cpp` sendiri (Bagian 5.3) sebagai contoh spesifik dari apa yang bisa diam-diam salah.
3. Telusuri `infixToPostfix()` dengan tangan (Bagian 5.4.2) pada ekspresi `9-2*3+1`, tunjukkan isi operator stack dan keluaran postfix setelah setiap token diproses, dengan cara yang sama Gambar 5.1 melakukannya untuk `3+2*5`. Apa ekspresi postfix akhirnya, dan apa hasil evaluasinya?
4. `evaluatePostfix()` men-pop dua operand paling barunya sebagai `b` lalu `a` (dalam urutan itu) sebelum menghitung `a - b` untuk token `-`. Jelaskan mengapa urutan pop penting di sini secara khusus, dan apa yang akan salah jika kode sebaliknya menghitung `b - a`.
5. Sebuah `NaiveQueue<int>` berkapasitas 5 sudah mengalami 5 enqueue diikuti 3 dequeue, lalu satu enqueue lagi. Menggunakan logika `isFull()`/`isEmpty()` sesungguhnya Bagian 5.6.1, nyatakan apakah queue kosong, penuh, atau tidak keduanya pada titik ini, dan jelaskan penalaran Anda dengan melacak `frontIdx` dan `rearIdx` langkah demi langkah.
6. Tabel perbandingan Bagian 5.7 mencantumkan Breadth-First Search sebagai aplikasi Queue, dengan sengaja tanpa menjelaskannya secara lengkap (itu tugas Bab 13). Berdasarkan hanya apa yang diajarkan bab ini tentang urutan FIFO, usulkan — dalam bahasa biasa, tanpa kode — mengapa sebuah penelusuran graf yang ingin mengunjungi node "terdekat dulu" akan meraih Queue alih-alih Stack.

Referensi

[1] B.R. Preiss. “Data Structures and Algorithms with Object-Oriented Design Patterns in C++.” John Wiley & Sons, 1998.

[2] P.J. Deitel, H.M. Deitel. “C++ How to Program.” 10th ed., Pearson, 2017.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Stacks and Queues, Chapter 10.)

[4] R. Sedgewick, K. Wayne. “Algorithms.” 4th ed., Addison-Wesley, 2011. (Stacks and Queues; classic infix-to-postfix conversion.)

[5] cppreference.com. “std::function.” https://en.cppreference.com/w/cpp/utility/functional/function (accessed August 2026).

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 6

Pointer & Fungsi

Bab 1 menunjukkan operator alamat-dari kepada Anda dan diam-diam menjanjikan penjelasan lengkap nanti. Inilah bab itu: pointer, secara lengkap, dan satu keputusan desain -- bagaimana sebuah fungsi menerima argumennya -- yang menentukan apakah kode pemanggil benar-benar bisa diubah oleh apa yang dipanggilnya. Setiap baris kode di sini benar-benar dikompilasi dengan g++ -Wall -Wextra dan dijalankan -- transkrip terminal aslinya direproduksi di Lampiran 6.A.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Mengingat kembali pratinjau pointer singkat Bab 1 (operator alamat-dari `&`, dan perbedaan stack/heap secara konseptual) dan menghubungkannya dengan pembahasan lengkap bab ini. (2) Mendeklarasikan variabel pointer, men-dereference-nya dengan `*`, membedakan pointer null dari yang menunjuk ke sesuatu yang nyata, dan memakai `->` untuk mencapai sebuah field lewat pointer ke `struct Book`. (3) Menjelaskan dan mendemonstrasikan aritmetika pointer -- `arr[i]` vs. `*(arr + i)`, increment/decrement pointer, dan pengurangan pointer sebagai jumlah elemen, bukan jumlah byte -- dibangun langsung di atas materi aritmetika-alamat-array Bab 1. (4) Membedakan DEKLARASI fungsi dari DEFINISI fungsi, dan menjelaskan mengapa C++ membutuhkan keduanya pada beberapa program. (5) Menelusuri, dengan kode terkompilasi sungguhan dan objek milik caller yang dicetak setelah setiap pemanggilan, persis bagaimana pass-by-value, pass-by-reference, dan pass-by-pointer berbeda dalam hal apakah sebuah fungsi bisa memodifikasi data caller. (6) Mengartikulasikan, dengan alasan konkret, kapan seorang programmer C++ sungguhan sebaiknya memilih parameter referensi versus parameter pointer. (7) Menjelaskan mengapa linked list Bab 7 membuat pointer wajib, bukan opsional -- pointer `next` sebuah node ADALAH struktur datanya, bukan detail implementasinya.

6.1 Ulasan: Pratinjau Pointer Bab 1, Dilengkapi di Sini

Bab 1 memperkenalkan `struct Book` dan katalog Pustaka Ceria, dan -- saat menjelaskan perbedaan stack/heap secara konseptual -- menunjukkan operator alamat-dari `&` secara sekilas: `&catalog[i]`, alamat satu elemen array. Itu dengan sengaja sebuah PRATINJAU, bukan pembahasan lengkap: Bab 1 hanya butuh secukupnya `&` untuk menjelaskan bahwa elemen-elemen array berada di alamat-alamat berurutan, tanpa juga mengajarkan variabel pointer, dereferencing, atau fungsi pada saat yang sama.

Bab ini adalah tempat janji itu ditepati secara lengkap. Semua yang ditunjukkan Bab 1 dengan `&catalog[i]` tetap berlaku -- bab ini tidak membantahnya, ia melengkapinya. Yang sungguh baru di sini: mendeklarasikan sebuah variabel pointer sungguhan (bukan hanya menulis `&x` inline), men-dereference-nya dengan `*`, pointer null, pointer-ke-struct dengan `->`, aritmetika pointer nyata (`++`, `--`, pengurangan), dan -- pusat gravitasi sesungguhnya bab ini -- bagaimana daftar parameter sebuah fungsi menentukan apakah fungsi itu bisa menjangkau balik dan mengubah data caller sendiri.

Bab 2 hingga 5 juga penting di sini, dengan satu cara spesifik: masing-masing melewatkan sebuah `Book` (atau seluruh array `Book`) ke dalam fungsi -- `printBook(const Book &b)`, `binarySearch(const Book catalog[], ...)`, `bubbleSort(Book catalog[], int n)`, `Stack<Book>::push(const Book &value)` --

tanpa pernah berhenti menjelaskan MENGAPA signature itu memakai `&`, atau apa yang akan salah jika tidak. Bab ini menjawab pertanyaan itu langsung, memakai persis `struct Book` yang sama yang sudah dipakai bab-bab itu.

6.2 Fondasi Pointer

Sebuah pointer adalah variabel yang NILAInya adalah alamat memori. Itulah keseluruhan idenya -- semua yang lain di bagian ini adalah konsekuensi dari persis satu fakta itu.

6.2.1 Mendeklarasikan Pointer, dan Men-dereference-nya

Deklarasi sebuah pointer menyebutkan TIPE yang ditunjuknya, diikuti `*`, diikuti nama pointer itu sendiri:

```
int score = 87;
int *scorePtr = &score;   // nilai scorePtr adalah alamat score

std::printf("score    = %d\n", score);       // 87
std::printf("&score   = %p\n", (void *) &score);   // alamat score
std::printf("scorePtr = %p\n", (void *) scorePtr); // alamat yang SAMA, kini
disimpan di variabel
std::printf("*scorePtr = %d\n", *scorePtr);   // dereference: ikuti alamatnya,
baca int di sana
```

`&score` menghitung alamat score sebagai ekspresi sekali pakai -- persis seperti yang sudah ditunjukkan Bab 1. `int *scorePtr = &score;` menyimpan alamat yang sama itu dalam sebuah variabel, sehingga bisa dibawa-bawa, ditugaskan ulang, atau diberikan ke sebuah fungsi. `*scorePtr` adalah operator DEREFERENCE: ia berkata "pergi ke alamat yang dipegang scorePtr, dan baca (atau tulis) `int` yang tinggal di sana". Menulis `*scorePtr = 95;` mengubah `score` itu sendiri -- hanya ada satu `int` dalam gambaran ini, dan pointer sekadar cara lain untuk mencapainya.

> **`&` dan `*` adalah kebalikan, dan keduanya adalah simbol yang sangat dibebani-makna-ganda di C++**
>
> `&x` berarti "alamat dari x" ketika muncul dalam sebuah ekspresi (seperti pada `int *p = &x;`), tetapi `&` yang muncul dalam sebuah DEKLARASI (`int &ref = x;`, atau sebuah parameter seperti `Book &b`) berarti sesuatu yang berbeda -- ia mendeklarasikan sebuah REFERENSI, dibahas di Bagian 6.5. Serupa, `*p` berarti "dereference p" dalam sebuah ekspresi, tetapi `int *p` dalam deklarasi berarti "p adalah pointer ke int". Konteks selalu menghilangkan ambiguitasnya, tetapi ada baiknya menamai makna-ganda ini secara eksplisit: ini adalah sumber kebingungan awal yang sangat umum, bukan tanda ada yang salah dengan Anda karena menganggapnya membingungkan pada awalnya.

6.2.2 Pointer Null

Sebuah pointer yang (belum) menunjuk apa pun yang nyata sebaiknya diset ke `nullptr` -- literal pointer-null eksplisit C++ -- alih-alih dibiarkan dengan alamat sampah apa pun yang kebetulan ada di memori saat variabel itu dideklarasikan:

```
int *nothingYet = nullptr;
if (nothingYet != nullptr) {
    // tidak pernah berjalan -- nothingYet sungguh null
} else {
    // jalur AMAN: periksa sebelum men-dereference, selalu
}
nothingYet = &score;   // kini ia sungguh menunjuk ke sesuatu
```

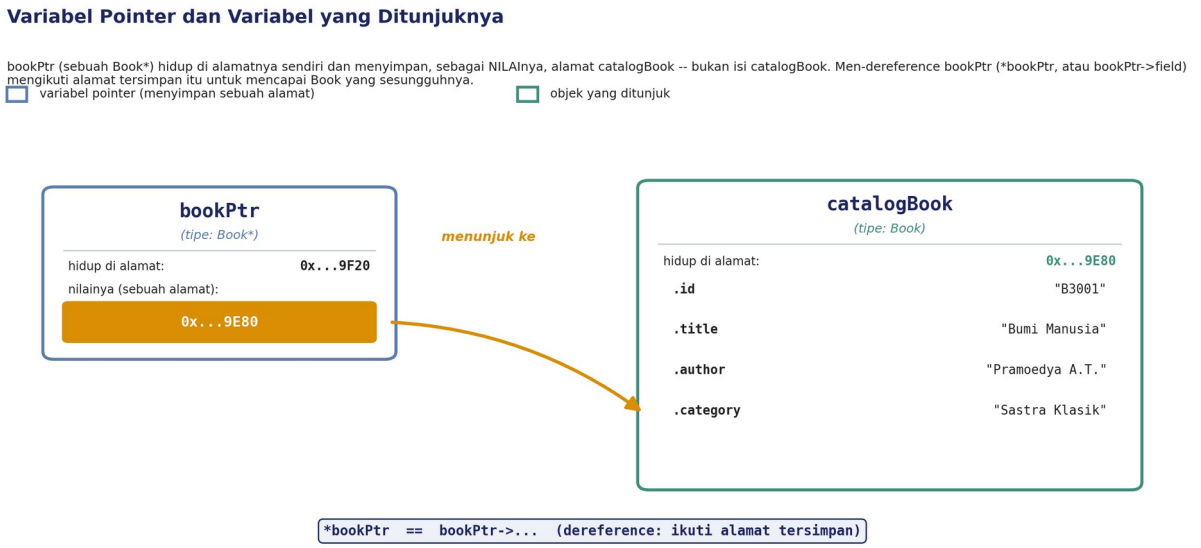
Men-dereference pointer null adalah undefined behavior

`*nothingYet` selagi `nothingYet` adalah `nullptr` tidak melempar exception C++ yang ramah -- ia adalah UNDEFINED BEHAVIOR, yang dalam praktiknya biasanya berarti crash langsung (segmentation fault), meski standar C++ bahkan tidak menjamin sebanyak itu. Disiplin yang dipakai bab ini di seluruh kodenya sendiri, dan diharapkan mulai sekarang, sederhana dan mutlak: periksa `if (ptr != nullptr)` (atau bentuk lebih pendek `if (ptr)`, yang berarti sama) SEBELUM pernah menulis `*ptr` atau `ptr->field`. `demo_pointer_basics.cpp` dengan sengaja tidak pernah men-dereference pointer null, di mana pun -- guard-nya selalu berjalan lebih dulu.

6.2.3 Pointer ke Struct: Book*, dan Operator ->

Sebuah pointer bisa menunjuk ke `struct Book` persis seperti ia bisa menunjuk ke `int` -- dan mencapai sebuah FIELD lewat pointer itu butuh satu langkah ekstra di luar variabel biasa:

```
Book catalogBook;
populateBook(catalogBook, "B3001", "Bumi Manusia",
             "Pramoedya Ananta Toer", "Novel");
Book *bookPtr = &catalogBook;

// Tiga cara mencapai field yang SAMA -- semuanya identik efeknya:
catalogBook.title      // langsung: tanpa pointer sama sekali
(*bookPtr).title       // dereference dulu, BARU titik -- tanda kurung wajib
bookPtr->title         // arrow: singkatan persis untuk (*bookPtr).title
```



Gambar 6.1 — Sebuah variabel pointer (bookPtr, sebuah Book) dan objek yang ditunjuknya (catalogBook, struct Book sungguhan). bookPtr hidup di alamatnya sendiri dan menyimpan, sebagai NILAInya, alamat catalogBook -- bukan isi catalogBook.*

Gambar 6.1 mengonkretkan perbedaannya: `bookPtr` adalah variabelnya sendiri, dengan alamatnya sendiri (`0x...9F20` pada gambar) -- dan NILAI yang tersimpan di alamat itu adalah `0x...9E80`, yang kebetulan adalah alamat `catalogBook` sendiri. Dua alamat yang sungguh berbeda terlibat di sini, dan mengacaukan keduanya adalah kesalahan pointer awal paling umum: `&bookPtr` (alamat bookPtr sendiri) bukan hal yang sama dengan `bookPtr` (alamat yang DIPEGANG bookPtr), yang lagi-lagi bukan hal yang sama dengan `&catalogBook` -- kecuali bahwa kedua yang terakhir itu, karena konstruksinya, sama.

> **`->` tidak lain adalah `(*p).`**
> Operator arrow adalah gula sintaksis murni -- `p->field` didefinisikan berarti persis `(*p).field`, dereference lalu titik. `demo_pointer_basics.cpp` mencetak `catalogBook.title`, `(*bookPtr).title`, dan `bookPtr->title` berdampingan dan mengonfirmasi ketiganya mencetak string yang identik, pada eksekusi nyata, bukan sekadar berdasarkan klaim. Dalam praktiknya, hampir semua kode C++ sungguhan memakai `->` untuk pointer-ke-struct/class dan tidak pernah menulis bentuk `(*p).` secara eksplisit -- tetapi mengetahui apa yang diperluas `->` adalah yang membuat dereference pointer null (Bagian 6.2.2) dan aritmetika pointer (Bagian 6.3) masuk akal sebagai satu ide koheren alih-alih tumpukan aturan terpisah.

Memutasi sebuah field lewat pointer bekerja persis seperti memutasi lewat objek langsung, karena hanya ada satu `catalogBook` di memori -- pointer hanyalah rute lain untuk mencapainya: `bookPtr->category` diset ke string baru sungguh mengubah `catalogBook.category`, dikonfirmasi pada eksekusi yang sama.

6.3 Aritmetika Pointer, Dilengkapi

`demo_1d.cpp` Bab 1 sudah menunjukkan bahwa `&catalog[i]` dan `catalog + i` menghitung alamat yang identik, untuk array `Book` Pustaka Ceria. Itu benar, tetapi ditunjukkan sepenuhnya lewat sintaks indexing-array -- bagian ini menunjukkan aritmetika yang mendasarinya yang sama lewat VARIABEL pointer `Book *` sungguhan, dijalankan melintasi array dengan `++`, `--`, dan penugasan majemuk, yaitu sintaks pointer lengkap yang dengan sengaja ditunda Bab 1.

6.3.1 arr[i] dan *(arr + i): Operasi yang Sama, Dua Ejaan

```
Book catalog[5];   // dipopulasi persis seperti di Bab 1
for (int i = 0; i < 5; i++) {
    bool same = (&catalog[i] == (catalog + i));
    // catalog[i].title  dan  (*(catalog + i)).title  adalah string yang SAMA,
    // untuk setiap i -- 'same' bernilai true pada setiap iterasi, setiap
eksekusi.
}
```

`arr[i]` adalah, menurut definisi bahasa C++ sendiri, singkatan untuk `*(arr + i)` -- subscripting array ADALAH aritmetika pointer, berdandan dalam notasi yang lebih nyaman. Ini bukan detail implementasi yang kebetulan benar pada kebanyakan compiler; inilah arti `[]` untuk array mentah, dijamin oleh standar.

6.3.2 Menjelajahi Array dengan Variabel Pointer

Karena nama sebuah array meluruh (decay) menjadi pointer ke elemen pertamanya (tidak butuh `&`), sebuah `Book *` bisa langsung diarahkan ke sebuah array `Book` lalu dijalankan maju atau mundur satu elemen -- bukan satu byte -- setiap kali:

```
Book *p = catalog;          // meluruh menjadi &catalog[0]
p->title;                   // title catalog[0]

p++;                        // maju SATU Book (sizeof(Book) = 196 byte),
p->title;                   //   bukan satu byte -- title catalog[1] sekarang

p += 2;                     // lompat 2 Book penuh maju -- catalog[3] sekarang
p--;                        // mundur SATU Book -- catalog[2] sekarang
```

Aritmetika pointer menskalakan berdasarkan sizeof(T) secara otomatis

`p + 1` untuk `Book *p` TIDAK berarti "satu byte setelah p" -- ia berarti "satu Book penuh setelah p", yaitu `sizeof(Book)` = 196 byte lebih jauh di memori (angka 196-byte yang sama yang pertama kali ditetapkan Lampiran Bab 1 untuk struct ini persis). Compiler menyisipkan penskalaan itu secara otomatis dari TIPE pointer itu sendiri; tidak ada yang berubah dalam aritmetikanya sendiri jika `Book` menambah field baru nanti -- hanya konstanta yang dikalikan compiler yang berubah. Inilah persis mengapa `arr[i]` mencapai elemen yang benar tidak pernah bergantung pada programmer menghitung offset byte secara manual.

6.3.3 Pengurangan Pointer: Jarak Antara Dua Elemen

Mengurangkan satu pointer dari yang lain (bertipe sama, ke dalam array yang sama) menghasilkan jumlah ELEMEN di antara keduanya -- lagi-lagi diskalakan otomatis oleh `sizeof(T)`, bukan jumlah byte mentah:

```
Book *start = &catalog[0];
Book *end   = &catalog[4];      // yang terakhir dari 5 elemen
std::ptrdiff_t distance = end - start;   // 4 -- jumlah ELEMEN

// Jarak byte mentah, dihitung manual untuk perbandingan:
//   (char*)end - (char*)start  ==  784
//   784 / sizeof(Book)         ==  784 / 196  ==  4   -- cocok dengan
'distance' di atas
```

`std::ptrdiff_t` adalah tipe integer bertanda yang dipakai C++ khusus untuk HASIL sebuah pengurangan pointer -- bertanda, karena mengurangkan pada arah sebaliknya (`start - end`) menghasilkan jumlah elemen negatif, yang merupakan jawaban yang sangat bermakna ("end berada 4 elemen SEBELUM start").

> **Pengurangan pointer hanya didefinisikan dalam array yang SAMA**
> `end - start` hanya bermakna ketika kedua pointer menunjuk ke dalam array yang sama (atau satu posisi setelah ujungnya). Mengurangkan dua pointer ke dalam dua array yang TIDAK BERKAITAN -- atau dua objek yang sama sekali tidak berkaitan -- dikompilasi tanpa keluhan tetapi adalah undefined behavior; compiler tidak punya cara untuk menangkap kesalahan ini untuk Anda. Setiap contoh pengurangan-pointer dalam kode bab ini dengan sengaja tetap dalam satu array persis karena alasan ini.

`demo_pointer_arithmetic.cpp` ditutup dengan penjelajahan penuh katalog 5-buku memakai HANYA aritmetika pointer -- tanpa `[]` di mana pun -- berhenti tepat saat pointer cursor mencapai `catalog + 5`, pointer klasik "satu-setelah-ujung": selalu sah untuk DIHITUNG dan DIBANDINGKAN, meski men-dereference-nya akan sama-sama undefined seperti men-dereference pointer null. Transkrip lengkapnya ada di Lampiran 6.A.

6.4 Fungsi: Deklarasi vs. Definisi

Setiap fungsi yang ditulis mata kuliah ini sejauh ini punya keduanya -- DEKLARASI (signature fungsi: nama, tipe parameter, tipe kembalian -- apa yang ditaruh `book.h` di dalam `#ifndef BOOK_H` untuk `populateBook`, `printBook`, dan `printCatalogHeader`) dan DEFINISI (badan sesungguhnya -- apa yang disediakan `book.cpp`). Bagian ini membuat pembagian itu eksplisit, karena ia menjadi krusial begitu sebuah program merentang lebih dari satu file `.cpp`, yang sudah dilakukan setiap bab sejak Bab 1 tanpa berhenti menamai perbedaannya.

```
// DEKLARASI (di book.h) -- janji tentang antarmuka fungsi:
// "di suatu tempat, sebuah fungsi dengan nama persis ini, tipe-tipe
//  parameter ini, dan tipe kembalian ini ada."
void populateBook(Book &b, const char *id, const char *title,
                  const char *author, const char *category);

// DEFINISI (di book.cpp) -- badan sesungguhnya yang menepati janji itu:
void populateBook(Book &b, const char *id, const char *title,
                  const char *author, const char *category) {
    safeCopy(b.id, sizeof(b.id), id);
    // ... sisa pekerjaan sesungguhnya ...
}
```

> **Mengapa pembagian ini penting: inilah yang memungkinkan demo_pointer_basics.cpp memanggil populateBook() sama sekali**
> `demo_pointer_basics.cpp` meng-include `book.h` (deklarasinya) dan dikompilasi BERSAMA `book.cpp` (definisinya) pada baris perintah g++ yang sama: `g++ ... book.cpp demo_pointer_basics.cpp`. Compiler hanya perlu MELIHAT deklarasi saat mengompilasi `demo_pointer_basics.cpp` -- ia tidak butuh badannya sama sekali pada titik itu, hanya janji bahwa badan yang cocok ada di suatu tempat. LINKER (langkah akhir dari satu pemanggilan g++ itu) adalah yang sesungguhnya menghubungkan titik panggil ke badan sesungguhnya, setelah kedua file `.cpp` dikompilasi menjadi kode objek. Inilah persis mengapa Makefile setiap bab mencantumkan `book.cpp` di samping setiap file demo -- menghilangkannya menghasilkan error linker yang nyata dan spesifik ("undefined reference to populateBook"), bukan error compiler, karena DEKLARASI saja sudah memuaskan compiler.

Daftar PARAMETER sebuah fungsi sendiri adalah bagian dari deklarasinya, dan Bagian 6.5 seluruhnya tentang satu bagian dari daftar parameter itu yang paling dipedulikan bab ini: bukan tipe parameternya, tetapi apakah masing-masing dilewatkan by value, by reference, atau by pointer.

6.5 Pass-by-Value vs. Pass-by-Reference vs. Pass-by-Pointer

Inilah topik sentral bab ini. Satu mutasi yang dimaksud -- set field `category` sebuah `Book` ke `"Buku Rusak"` (buku rusak, ditarik dari peredaran) -- dicoba dengan tiga cara berbeda, dengan logika badan yang SAMA setiap kali. Hanya cara parameter dideklarasikan yang berubah, dan Bagian 6.5.4 menunjukkan, dengan keluaran terkompilasi nyata, bahwa satu pilihan tunggal ini menentukan segalanya tentang apakah data caller sungguh berubah.

6.5.1 Pass-by-Value: Salinan Penuh dan Independen

```
void updateCategoryByValue(Book b, const char *newCategory) {
    std::snprintf(b.category, sizeof(b.category), "%s", newCategory);
    // 'b' adalah SALINAN LENGKAP dan INDEPENDEN dari Book apa pun yang
    // dilewatkan caller. Setiap field -- id, title, author, category --
    // disalin byte demi byte saat fungsi ini dipanggil. Memutasi
    // b.category di sini hanya memutasi salinan itu; ia dibuang begitu
    // fungsi ini kembali.
}
```

Pass-by-value adalah default C++ untuk parameter apa pun yang ditulis tanpa `&` atau `*`. Bukan kesalahan atau kelalaian ketika caller sebuah fungsi tidak melihat efek dari mutasi parameter by-value -- itu persis yang dimaksud pass-by-value, bekerja persis seperti yang dirancang.

6.5.2 Pass-by-Reference: Alias untuk Objek Milik Caller Sendiri

```
void updateCategoryByReference(Book &b, const char *newCategory) {
    std::snprintf(b.category, sizeof(b.category), "%s", newCategory);
    // 'b' BUKAN salinan -- ia adalah NAMA lain untuk Book milik caller.
    // Tidak ada Book baru dibuat di mana pun. Memutasi b.category ADALAH
    // memutasi Book.category milik caller, langsung, karena keduanya
    // adalah objek yang sama.
}
```

Sebuah parameter `Book &b` mengikat `b` sebagai alias untuk `Book` apa pun yang dilewatkan caller -- membaca atau menulis `b` membaca atau menulis objek milik caller sendiri, tanpa salinan yang pernah dibuat. Inilah persis signature yang sudah dipakai Bab 2-5 di mana-mana (`printBook(const Book &b)`, `Stack<T>::push(const T &value)`) tanpa penjelasan bab ini yang tersedia untuk membenarkannya.

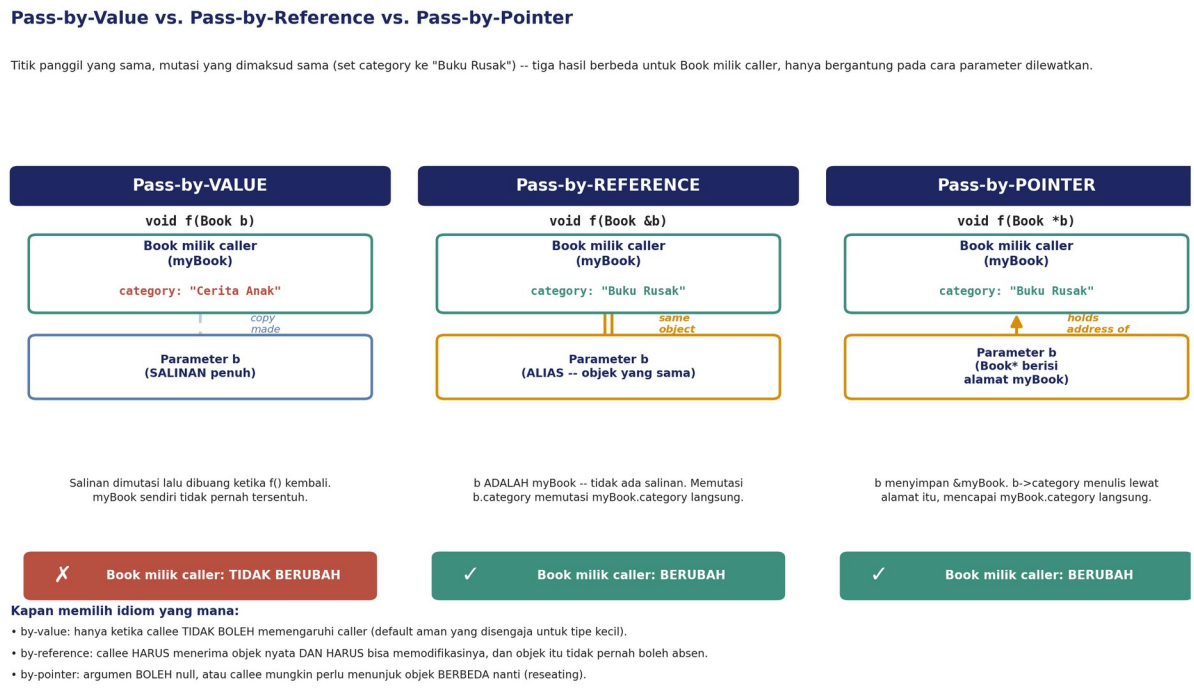
6.5.3 Pass-by-Pointer: Sebuah Alamat, dan Langkah Ekstra Eksplisit

```
void updateCategoryByPointer(Book *b, const char *newCategory) {
    if (b == nullptr) {
        return;   // guard dulu, selalu -- lihat Bagian 6.2.2
    }
    std::snprintf(b->category, sizeof(b->category), "%s", newCategory);
    // 'b' MEMEGANG alamat Book milik caller. b->category men-dereference
    // alamat itu untuk mencapai field sesungguhnya -- mencapai objek yang
    // SAMA dengan versi referensi di atas, hanya lewat alamat eksplisit
    // alih-alih alias implisit.
}
```

Sebuah parameter `Book *b` menerima ALAMAT dari `Book` apa pun yang dihitung ekspresi `&myBook` milik caller -- titik panggilnya sendiri harus menulis `&` secara eksplisit (`updateCategoryByPointer(&myBook, ...)`), tidak seperti versi referensi, yang tidak butuh sintaks khusus apa pun di titik panggil sama sekali. Sebagai gantinya sintaks ekstra itu, pointer mendapat dua hal yang tidak bisa ditawarkan referensi: sebuah parameter pointer BISA secara sah menjadi `nullptr` (diperiksa di atas, sebelum dereference apa pun), dan sebuah variabel pointer bisa ditugaskan ulang nanti untuk menunjuk objek yang sama sekali berbeda ("re-seating") -- sebuah referensi, sekali diikat, adalah objek itu untuk seluruh masa hidupnya dan tidak pernah bisa di-re-seat ke objek berbeda.

6.5.4 Ketiganya, Dipanggil pada Book yang Sama, Ditelusuri Ujung ke Ujung

`demo_pass_modes.cpp` mulai dengan satu `Book` (`myBook`, category `"Cerita Anak"`), memanggil ketiga fungsi di atas dengan category baru yang dimaksud sama persis, dan mencetak `Book.category` milik CALLER sendiri -- bukan state internal fungsi -- setelah setiap pemanggilan, untuk menunjukkan secara konkret, bukan berdasarkan klaim, idiom mana yang sungguh mengubahnya:



Gambar 6.2 — Mutasi yang dimaksud sama, dicoba tiga cara. Pass-by-value meninggalkan Book milik caller tidak berubah; pass-by-reference dan pass-by-pointer keduanya mengubahnya, lewat mekanisme berbeda.

```
$ g++ -Wall -Wextra -std=c++17 -o demo_pass_modes book.cpp pass_modes.cpp
demo_pass_modes.cpp
$ ./demo_pass_modes
Starting state: Cerita Rakyat Nusantara (category = "Cerita Anak")

--- Attempt 1: updateCategoryByValue(myBook, "Buku Rusak") ---
  [by-value]     inside the function, b.category is now "Buku Rusak"
  Caller's Book after by-value       -> category = "Cerita Anak"
  ==> caller's category UNCHANGED: the function only ever touched its own copy

--- Attempt 2: updateCategoryByReference(myBook, "Buku Rusak") ---
  [by-reference] inside the function, b.category is now "Buku Rusak"
  Caller's Book after by-reference   -> category = "Buku Rusak"
  ==> caller's category CHANGED: 'b' inside the function WAS myBook, not a copy

--- Attempt 3: updateCategoryByPointer(&myBook, "Buku Rusak") ---
  [by-pointer]   inside the function, b->category is now "Buku Rusak"
  Caller's Book after by-pointer     -> category = "Buku Rusak"
  ==> caller's category CHANGED: &myBook's ADDRESS was passed, and b-> wrote
through it

--- Attempt 4: updateCategoryByPointer(nullptr, "Buku Rusak") -- guarding null
---
  [by-pointer]   received a null Book* -- refusing to dereference, returning
early
  ==> no crash: the function's own nullptr guard caught it before any
dereference
```

Hasilnya persis seperti yang diprediksi Bagian 6.5.1-6.5.3, sungguh dijalankan bukan hanya diklaim: pass-by-value adalah satu-satunya dari ketiganya yang meninggalkan `Book` milik caller tidak tersentuh. Transkrip lengkap yang tidak diringkas (termasuk penanganan null aman Attempt 4) direproduksi di Lampiran 6.A.

6.5.5 Kapan Memilih Idiom yang Mana, dalam C++ Sungguhan

Idiom	Pilih ketika...	Contoh bab ini
Pass-by-value	callee TIDAK BOLEH memengaruhi caller — default aman yang disengaja untuk tipe kecil di mana salinan murah	updateCategoryByValue() -- salinan dimutasi lalu dibuang
Pass-by-reference	callee HARUS menerima objek nyata DAN harus bisa memodifikasinya, dan objek itu tidak pernah boleh secara sah absen	updateCategoryByReference(); setiap signature Bab 2-5 memakai const Book&/Book&
Pass-by-pointer	argumen BOLEH secara sah null, atau callee mungkin perlu menunjuk objek BERBEDA nanti (re-seating)	updateCategoryByPointer() -- menjaga nullptr secara eksplisit sebelum dereference

Aturan praktis yang layak diinternalisasi sekarang

Panduan C++ dunia-nyata yang banyak dipakai (digaungkan dalam C++ Core Guidelines) adalah: pilih parameter referensi secara default ketika objek harus ada dan harus terjangkau; raih pointer khusus ketika null adalah kemungkinan yang sungguh bermakna, atau ketika callee butuh kemampuan menunjuk ke tidak ada apa-apa, atau ke sesuatu yang lain, nanti. Lewatkan by value hanya ketika salinan itu murah (tipe kecil seperti field char-array bertipe-tetap milik Book) atau memang sengaja diinginkan (callee sebaiknya bekerja pada data MILIKNYA SENDIRI, tidak terpengaruh caller). Tidak satu pun dari ketiganya sekadar 'lebih baik' dari yang lain -- masing-masing adalah alat yang tepat untuk maksud nyata yang berbeda, persis seperti simpulan Bab 5 tentang Stack vs. Queue.

6.6 Menoleh ke Belakang, dan ke Depan: Bab 5 dan Bab 7

Bab 2 hingga 5 sudah terus-menerus melewatkan objek `Book` ke dalam fungsi -- setiap pemanggilan `printBook(const Book &b)`, setiap `Stack<Book>::push(const Book &value)`, setiap parameter `Book catalog[]` fungsi pengurutan -- semuanya memakai pass-by-reference atau peluruhan array (dengan sendirinya sebuah bentuk pelewatan pointer) tanpa kosakata bab ini yang tersedia untuk menamai apa yang terjadi atau mengapa. Tidak ada di bab-bab itu yang perlu dikoreksi; bab ini sekadar menyediakan penjelasan yang diam-diam diandalkan signature-signature itu selama ini.

Penunjuk maju: Bab 7 membuat pointer wajib, bukan opsional
Setiap teknik pointer di bab ini bersifat OPSIONAL sejauh ini -- sebuah array `Book` selalu bisa dilewatkan by reference atau by value sebagai gantinya, dan setiap demo dalam kode bab ini punya kembaran berbasis-`[]` yang akan bekerja identik. Itu berhenti benar mulai bab berikutnya. Seluruh struktur sebuah Singly Linked List ADALAH rantai pointer: field `next` setiap node adalah pointer ke node BERIKUTNYA, dan tidak ada array di baliknya sama sekali -- tidak ada memori kontigu, tidak ada aritmetika alamat `sizeof(Node) * i` untuk diandalkan. Bab 7 adalah tempat sintaks pointer bab ini berhenti menjadi 'salah satu cara di antara beberapa' dan menjadi satu-satunya cara struktur data itu bisa ada sama sekali.

6.7 Lampiran 6.A — Log Validasi

Keempat program di bawah ini (`demo_pointer_basics.cpp`, `demo_pointer_arithmetic.cpp`, `demo_pass_modes.cpp`, `demo_all.cpp`) dikompilasi dengan `g++ -Wall -Wextra -std=c++17` (nol peringatan dari keempatnya) dan benar-benar dieksekusi; transkrip di bawah adalah keluaran terminal yang asli dan tidak diedit -- bukan keluaran "yang diharapkan" yang ditulis tangan. Alamat memori yang ditunjukkan adalah nilai nyata dari satu eksekusi sungguhan; keduanya akan berbeda, sebagaimana diharapkan, pada eksekusi lain atau mesin lain, karena sistem operasi memberikan alamat baru setiap kali -- ini diungkapkan secara eksplisit alih-alih disajikan seolah alamat adalah konstanta tetap.

6.A.1 demo_pointer_basics.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_pointer_basics book.cpp
demo_pointer_basics.cpp
$ ./demo_pointer_basics
=== Chapter 6: Pointer Fundamentals ===

--- Part 1: pointer to int ---
score            = 87
&score           = 0x7ffcbf857d3c   (score's address)
scorePtr         = 0x7ffcbf857d3c   (same address, stored in a pointer variable)
*scorePtr        = 87   (dereferencing scorePtr reads score's value)

Writing through the pointer: *scorePtr = 95;
score is now     = 95   (changed via the pointer, not by touching 'score'
directly)

--- Part 2: null pointers ---
nothingYet       = (nil)   (nullptr: points at nothing, by design)
nothingYet == nullptr ? true
Guard checked BEFORE dereferencing -- *nothingYet was never attempted.
(Dereferencing a null pointer is undefined behavior; a real program
 crashes or corrupts memory. This demo deliberately never does it.)
after nothingYet = &score: *nothingYet = 95

--- Part 3: pointer to struct Book (Pustaka Ceria) ---
Via the object directly:      catalogBook.title  = Bumi Manusia
Via (*bookPtr).title (explicit dereference then dot):  Bumi Manusia
Via bookPtr->title  (arrow operator, the idiomatic form): Bumi Manusia
(*bookPtr).title and bookPtr->title are exactly the same thing --
 '->' is defined as shorthand for '(*p).', nothing more.)

Mutating through the pointer: bookPtr->category set to "Sastra Klasik"
catalogBook.category is now: Sastra Klasik   (mutated via the pointer -- same
object)

--- Part 4: guarding a possibly-null Book* ---
maybeBook is null: true
Safe pattern used throughout this chapter's code: always check
'if (ptr != nullptr)' (or 'if (ptr)') before writing ptr->field.

=== Summary: all pointer/dereference/null-check operations behaved as described
===
```

6.A.2 demo_pointer_arithmetic.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_pointer_arithmetic book.cpp
demo_pointer_arithmetic.cpp
$ ./demo_pointer_arithmetic
=== Chapter 6: Pointer Arithmetic over an Array of Book ===

--- Part 1: arr[i] and *(arr + i) read the same element ---
i=0  catalog[i].title=Laskar Pelangi           *(catalog+i).title=Laskar Pelangi
&catalog[i]==catalog+i: true
i=1  catalog[i].title=Filosofi Kopi            *(catalog+i).title=Filosofi Kopi
&catalog[i]==catalog+i: true
i=2  catalog[i].title=Bumi Manusia             *(catalog+i).title=Bumi Manusia
&catalog[i]==catalog+i: true
i=3  catalog[i].title=Sejarah Majapahit        *(catalog+i).title=Sejarah
Majapahit         &catalog[i]==catalog+i: true
i=4  catalog[i].title=Negeri di Ujung Tanduk   *(catalog+i).title=Negeri di
Ujung Tanduk    &catalog[i]==catalog+i: true

--- Part 2: walking the array with a Book* pointer variable ---
p starts at catalog[0]: p->title = Laskar Pelangi
p++  (advance one Book, i.e. sizeof(Book) = 196 bytes)
p is now at catalog[1]: p->title = Filosofi Kopi
p += 2 : p is now at catalog[3]: p->title = Sejarah Majapahit
p--  : p is now at catalog[2]: p->title = Bumi Manusia

--- Part 3: pointer subtraction (distance between elements) ---
start = &catalog[0], end = &catalog[4]
end - start = 4   (element count between them, NOT bytes -- the compiler
              divides the raw byte gap by sizeof(Book) = 196 automatically)
Raw byte gap, computed manually for comparison: 784
Manual byte gap / sizeof(Book) = 4  -- matches end - start above

--- Part 4: full traversal using only pointer arithmetic (no []) ---
ID     TITLE                          AUTHOR              CATEGORY
------ ------------------------------ ------------------- ------------
B4001  Laskar Pelangi                 Andrea Hirata       Novel
B4002  Filosofi Kopi                  Dee Lestari         Kumpulan Cerita
B4003  Bumi Manusia                   Pramoedya Ananta Toer Novel
B4004  Sejarah Majapahit              Slamet Muljana      Sejarah
B4005  Negeri di Ujung Tanduk         Tere Liye           Novel
Traversal stopped exactly when cursor reached (catalog + 5) --
the classic 'one-past-the-end' pointer, itself always valid to compute
and compare against, even though it must never be dereferenced.

=== Summary: array indexing, pointer increment/decrement, and pointer ===
=== subtraction all behaved exactly as this chapter describes ===
```

6.A.3 demo_pass_modes.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_pass_modes book.cpp pass_modes.cpp
demo_pass_modes.cpp
$ ./demo_pass_modes
=== Chapter 6: Pass-by-Value vs. Pass-by-Reference vs. Pass-by-Pointer ===

Starting state: Cerita Rakyat Nusantara (category = "Cerita Anak")

--- Attempt 1: updateCategoryByValue(myBook, "Buku Rusak") ---
  [by-value]     inside the function, b.category is now "Buku Rusak"
  [by-value]     but 'b' is a local copy -- it is destroyed when this function
returns
  Caller's Book after by-value       -> category = "Cerita Anak"
  ==> caller's category UNCHANGED: the function only ever touched its own copy

--- Attempt 2: updateCategoryByReference(myBook, "Buku Rusak") ---
  [by-reference] inside the function, b.category is now "Buku Rusak"
  [by-reference] 'b' IS the caller's Book (an alias, no copy) -- this change
sticks
  Caller's Book after by-reference   -> category = "Buku Rusak"
  ==> caller's category CHANGED: 'b' inside the function WAS myBook, not a copy

(reset myBook.category back to "Cerita Anak" before the next attempt, for a fair
comparison)

--- Attempt 3: updateCategoryByPointer(&myBook, "Buku Rusak") ---
  [by-pointer]   inside the function, b->category is now "Buku Rusak"
  [by-pointer]   'b' holds the caller's Book's ADDRESS -- b->category writes
through it, so this sticks too
  Caller's Book after by-pointer     -> category = "Buku Rusak"
  ==> caller's category CHANGED: &myBook's ADDRESS was passed, and b-> wrote
through it

--- Attempt 4: updateCategoryByPointer(nullptr, "Buku Rusak") -- guarding null
---
  [by-pointer]   received a null Book* -- refusing to dereference, returning
early
  ==> no crash: the function's own nullptr guard caught it before any
dereference

=== Final Summary ===
Pass-by-value:     did NOT modify the caller's Book (a copy was mutated instead)
Pass-by-reference: DID modify the caller's Book (no copy was ever made)
Pass-by-pointer:   DID modify the caller's Book (through an explicit address +
->)
Pass-by-pointer with nullptr: safely refused, thanks to the function's own null
check
```

6.A.4 demo_all.cpp (transkrip lengkap — semuanya, berdampingan)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_all book.cpp pass_modes.cpp demo_all.cpp
$ ./demo_all
=== Chapter 6 Validation Summary: Pointers & Functions ===

[Pointer basics]      *xp = 20 through xp=&x -> x is now 20 (expect 20): OK
[Null pointer]        nullp == nullptr: OK
[Struct pointer]      bp->title == (*bp).title: OK
[Pointer arithmetic]  &catalog[i] == catalog+i for all i: OK
[Pointer arithmetic]  catalog + 3 (via +=) == &catalog[3]: OK
[Pointer subtraction] &catalog[3] - &catalog[0] == 3: OK

[Pass-mode comparison]
  by-value:     caller's category still "Original": OK (unchanged, as expected)
  by-reference: caller's category now "ChangedByRef": OK (changed, as expected)
  by-pointer:   caller's category now "ChangedByPtr": OK (changed, as expected)
  by-pointer(null): guarded, no crash: OK

Overall: ALL CHECKS PASSED
```

Catatan kejujuran

Setiap nilai, alamat, dan baris verifikasi yang ditunjukkan di atas (di keempat program) berasal langsung dari kompilasi-dan-eksekusi nyata di lingkungan build bab ini. Berbeda dari Bab 5 (yang menemukan dan mengungkapkan bug off-by-two nyata dalam aritmetika verifikasi salah satu demo), pass validasi bab ini sendiri tidak menemukan bug apa pun di keempat program -- diungkapkan di sini secara jujur, sesuai kebijakan tetap mata kuliah ini untuk melaporkan hasil validasi persis sebagaimana terjadi, entah itu mencakup bug atau tidak. Zip kode yang dikirim juga di-ekstrak ulang dan dibangun ulang secara independen dari `make clean && make run` yang bersih untuk memastikan deliverable sesungguhnya, bukan hanya salinan kerja, dikompilasi dan dijalankan secara identik.

6.8 Rangkuman Bab dan Poin-Poin Kunci

- Sebuah pointer adalah variabel yang nilainya adalah alamat memori. Mendeklarasikan satu (`int *p`), men-dereference-nya (`*p`), dan memeriksa null (`p == nullptr`, selalu sebelum dereference) adalah operasi pointer fondasional bab ini.
- `p->field` adalah persis `(*p).field` -- operator arrow adalah gula sintaksis murni, didemonstrasikan (bukan sekadar diklaim) menghasilkan hasil identik dengan bentuk dereference-lalu-titik eksplisit.
- `arr[i]` adalah, menurut definisi bahasa sendiri, `*(arr + i)` -- indexing array ADALAH aritmetika pointer. Aritmetika pointer menskalakan otomatis berdasarkan `sizeof(T)`, dan pengurangan pointer (dalam satu array) menghasilkan jumlah elemen, bukan jumlah byte.
- Setiap fungsi punya DEKLARASI (antarmukanya, di file `.h`) dan DEFINISI (badannya, di file `.cpp`) -- pembagian ini adalah yang memungkinkan satu file `.cpp` memanggil fungsi yang badannya tinggal di file lain, setelah keduanya dikompilasi dan di-link bersama.
- Pass-by-value menyalin argumen; callee tidak pernah bisa memengaruhi caller lewatnya. Pass-by-reference memberi callee alias sungguhan untuk objek milik caller sendiri -- tanpa salinan, dan mutasi melekat. Pass-by-pointer memberi callee sebuah alamat ke objek caller,

membutuhkan `&` eksplisit di titik panggil dan `->`/`*` eksplisit di dalam fungsi, sebagai gantinya kemampuan untuk null atau di-re-seat ke objek berbeda nanti.

- Pilih referensi ketika callee harus menerima, dan harus bisa memodifikasi, objek yang tidak pernah boleh secara sah absen. Pilih pointer ketika null adalah kemungkinan nyata, atau ketika re-seating ke objek berbeda nanti penting. Tidak ada yang universal 'lebih baik' -- masing-masing cocok untuk maksud nyata yang berbeda.
- Singly Linked List Bab 7 membuat pointer wajib, bukan opsional: field `next` sebuah node adalah pointer ke node berikutnya, dan tidak ada array di baliknya untuk diandalkan.

6.9 Pertanyaan Ulasan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Bab 7 (Singly Linked List).

1. Diberikan `int x = 5; int *p = &x; int *q = p;`, jelaskan dengan kata-kata Anda sendiri apa yang dipegang `q`, dan prediksi apa yang dilakukan `*q = 10;` terhadap `x`. Apakah jawaban Anda berubah jika `q` sebaliknya dideklarasikan sebagai `int q = *p;` (tanpa asterisk)? Jelaskan perbedaannya.
2. `demo_pointer_basics.cpp` (Bagian 6.2.2) memeriksa `if (nothingYet != nullptr)` sebelum pernah menulis `*nothingYet`. Deskripsikan, secara konkret, konsekuensi dunia-nyata apa yang dilindungi guard ini, dan mengapa compiler C++ tidak bisa begitu saja menangkap dereference-pointer-null untuk Anda saat kompilasi.
3. Untuk `Book catalog[5]` 5-elemen yang sama yang dipakai bab ini, apa yang ditunjuk `catalog + 5`, dan mengapa menghitung alamat itu sah padahal MEN-DEREFERENCE-nya tidak? Hubungkan jawaban Anda dengan kondisi loop penjelajahan Bagian 6.3.3 `demo_pointer_arithmetic.cpp`.
4. Tuliskan, dengan kata-kata Anda sendiri, mengapa `updateCategoryByValue()` (Bagian 6.5.1) bukan bug -- ia melakukan persis apa yang didefinisikan pass-by-value untuk dilakukan. Lalu deskripsikan satu skenario realistis (bukan dari bab ini) di mana seorang programmer AKAN menginginkan pass-by-value secara khusus, alih-alih by-reference atau by-pointer.
5. Seorang kolega berargumen bahwa pointer harus selalu dipilih dibanding referensi, "karena pointer bisa melakukan segala yang bisa dilakukan referensi, plus lebih banyak lagi." Memakai tabel Bagian 6.5.5, berikan dua alasan konkret mengapa parameter referensi tetap pilihan yang lebih baik pada banyak fungsi nyata, meski pointer secara teknis bisa dibuat bekerja.
6. Bab 7 akan memperkenalkan struct `Node` dengan field `Book data;` dan field `Node *next;`. Berdasarkan hanya apa yang diajarkan bab ini tentang pointer dan struct, prediksi: apa arti `head->next->next->data.title`, jika `head` adalah pointer ke node pertama sebuah list dengan setidaknya 3 node? Telusuri apa yang dilakukan setiap `->`, satu per satu.

Referensi

[1] B.R. Preiss. “Data Structures and Algorithms with Object-Oriented Design Patterns in C++.” John Wiley & Sons, 1998.

[2] P.J. Deitel, H.M. Deitel. “C++ How to Program.” 10th ed., Pearson, 2017. (Pointers, References, and Functions.)

[3] B. Stroustrup. “The C++ Programming Language.” 4th ed., Addison-Wesley, 2013.

[4] ISO/IEC. “C++ Core Guidelines — F.16-F.21 (Parameter Passing).” isocpp.github.io/CppCoreGuidelines (accessed August 2026).

[5] cppreference.com. “Pointer declaration” dan “Reference declaration.” https://en.cppreference.com/w/cpp/language (accessed August 2026).

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 7

Singly Linked List

Bab 6 ditutup dengan sebuah janji: pointer next sebuah node akan berhenti menjadi salah satu teknik di antara beberapa dan menjadi satu-satunya cara sebuah struktur data bisa ada sama sekali. Inilah bab itu. Setiap baris kode di sini benar-benar dikompilasi dengan g++ -Wall -Wextra dan dijalankan -- dan, untuk pertama kalinya dalam mata kuliah ini, diperiksa secara independen untuk kebocoran memori dengan valgrind. Transkrip lengkap keduanya direproduksi di Lampiran 7.A.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Menjelaskan trade-off array-vs-linked-list (memori kontigu vs. tersebar, akses acak O(1) vs. akses hanya-penjelajahan O(n), ukuran tetap vs. pertumbuhan dinamis) dan menempatkan array Bab 1 berdampingan dengan linked list bab ini sebagai dua jawaban berbeda untuk pertanyaan penyimpanan yang sama. (2) Mendeklarasikan struct `Node` yang memasangkan payload `Book` dengan pointer `Node *next`, dan menjelaskan mengapa pointer itu ADALAH linked list-nya, bukan detail implementasi darinya. (3) Mengimplementasikan dan menelusuri operasi `init`, `frontInsertion`, `show`, `frontDeletion`, dan `clear` sebuah singly linked list head-only. (4) Mengimplementasikan dan menelusuri `backInsertion` dan `backDeletion` sebuah singly linked list head+tail, dan menjelaskan -- secara konkret, bukan hanya berdasarkan nama -- mengapa `backInsertion` adalah O(1) sementara `backDeletion` tetap O(n) meski ada pointer tail. (5) Mengimplementasikan dan menelusuri sebuah circular singly linked list, menjelaskan apa yang sesungguhnya berubah karena sifat circular (hanya ke mana next node terakhir menunjuk), dan menulis penjelajahan yang tetap aman dengan membatasi berdasarkan jumlah node, bukan memeriksa null. (6) Menghubungkan penjadwalan round-robin, sebagai aplikasi motivasi alami sebuah circular list, kembali ke bagian aplikasi Queue Bab 5. (7) Menjelaskan, secara konkret, apa yang ditambahkan Doubly Linked List Bab 9 di atas semua yang dibangun bab ini, dan mengapa.

7.1 Ulasan: Mengapa Linked List, Padahal Kita Sudah Punya Array?

Bab 1 membangun katalog Pustaka Ceria sebagai array `Book` -- blok memori kontigu, setiap elemen berukuran tetap sama, duduk berurutan satu demi satu. Bab 6 kemudian menghabiskan satu bab penuh membuat pointer konkret: sebuah variabel yang nilainya adalah alamat, bisa dijelajahi dengan `++`, `--`, dan pengurangan. Kedua bab itu diam-diam membangun menuju satu pertanyaan yang akhirnya ditanyakan langsung oleh bagian ini: array sudah memberi penyimpanan yang cepat, kontigu, dan bisa diindeks -- jadi mengapa ada orang yang menginginkan sesuatu yang lain?

Jawaban jujurnya adalah sebuah trade-off, bukan perbaikan mutlak. Elemen-elemen array duduk di alamat-alamat berurutan, yang persis membuat `catalog[i]` menjadi operasi O(1) -- aritmetika alamat Bab 1, `base + i * sizeof(Book)`, menghitung lokasi elemen mana pun secara langsung, tanpa pencarian sama sekali. Kekontiguan yang sama itu juga menjadi keterbatasan utama array: ukurannya tetap pada saat deklarasi (atau alokasi, untuk yang berukuran dinamis), dan menyisipkan atau menghapus di

mana pun selain di ujung paling akhir berarti menggeser setiap elemen berikutnya untuk menjaga array tetap kontigu -- biaya O(n) yang dibayar pada setiap penyisipan atau penghapusan di tengah.

Sebuah linked list mengambil trade-off sebaliknya. Node-nya tersebar di mana pun alokator memori kebetulan menempatkan masing-masing -- tidak ada `catalog[i]` sama sekali, karena tidak ada rumus yang menghubungkan posisi sebuah node dalam list dengan alamatnya; satu-satunya cara mencapai node ke-5 adalah mengikuti 4 pointer `next` dari node ke-1. Itu adalah biaya nyata dan permanen: akses linked list secara alami adalah O(n), tidak pernah O(1), untuk apa pun selain node yang pointernya sudah Anda pegang. Sebagai gantinya, sebuah linked list tumbuh dan menyusut satu node setiap kali, persis sesuai kebutuhan, tanpa pergeseran elemen lain sama sekali -- menyisipkan atau menghapus pada posisi yang pointernya sudah Anda pegang adalah O(1), tidak peduli seberapa panjang sisa list-nya.

	Array (Bab 1)	Singly Linked List (bab ini)
Tata letak memori	Kontigu — satu blok, elemen berdampingan	Tersebar — setiap node di mana pun new menempatkannya
Akses berdasarkan posisi	O(1) — catalog[i], aritmetika alamat langsung	O(n) — harus mengikuti pointer next dari head
Ukuran	Tetap pada saat deklarasi/alokasi	Tumbuh/menyusut satu node setiap kali, dinamis
Sisip/hapus pada node yang SUDAH DIKETAHUI	O(n) — harus menggeser setiap elemen berikutnya	O(1) — menyambung ulang beberapa pointer, tidak ada yang bergeser
Sisip/hapus: cari posisinya dulu	O(1) setelah ditemukan (indexing langsung)	O(n) — mencari posisinya berarti menjelajah

Tidak ada struktur yang sekadar "lebih baik"

Array menang telak ketika program banyak melakukan pembacaan posisi-acak dan ukurannya diketahui atau jarang berubah -- semua algoritma pencarian Bab 2 mengandalkan indexing O(1). Linked list menang ketika program banyak melakukan penyisipan dan penghapusan, terutama di depan, dan ukuran totalnya tidak terduga atau terus berubah. `Node` bab ini tetap menyimpan `Book` -- tipe rekamannya tidak pernah berubah; yang berubah hanya cara koleksi Book itu disimpan.

Sedikit sejarah yang jujur

Linked list ditemukan di RAND Corporation pada 1955-56, untuk sebuah bahasa bernama IPL (Information Processing Language) -- salah satu teknik paling awal yang dikembangkan untuk apa yang sekarang kita sebut struktur data, bertahun-tahun sebelum istilah itu sendiri banyak dipakai. Ia diciptakan untuk memecahkan persis masalah yang baru dijelaskan bagian ini: merepresentasikan koleksi yang ukuran dan bentuknya tidak diketahui sebelumnya, sesuatu yang secara jujur tidak bisa dilakukan dengan anggun oleh array berukuran tetap pada perangkat keras memori era itu yang lebih terbatas. Hampir tujuh puluh tahun kemudian, ide intinya -- sebuah node yang menyimpan data plus pointer ke node berikutnya -- tidak berubah.

7.2 Node: Tempat Pointer Bab 6 Menjadi Krusial

Sebuah linked list butuh persis satu kosakata baru: NODE. Setiap node menggabungkan dua hal -- sebuah payload (bab ini memakai `Book` milik Pustaka Ceria sendiri, tidak berubah sejak Bab 1) dan sebuah pointer ke node berikutnya dalam list:

```
struct Node {
    Book data;   // payload -- struct Book Bab 1, tidak berubah
    Node *next;  // pointer ke node BERIKUTNYA, atau nullptr jika tidak ada
};
```

Ini dengan sengaja BUKAN class template seperti `Stack<T>`/`Queue<T>` Bab 5 -- keduanya butuh satu implementasi untuk mendukung beberapa tipe elemen yang tidak berkaitan (`Book`, `char`, `double`). `Node` bab ini konkret dengan sengaja, karena inti bab ini adalah apa arti field `next` tunggal itu: ia bukan detail implementasi yang disembunyikan di balik sebuah antarmuka, ia bukan satu opsi di antara beberapa -- ia ADALAH list-nya. Tidak ada array di mana pun di bawah sebuah linked list. Hapus pointer `next` setiap node dan tidak ada cara memulihkan node mana yang datang setelah yang mana; pointer-pointer itu adalah satu-satunya hal yang menghubungkan satu `Book` ke `Book` berikutnya.

> **Seluruh kosakata Bab 6 adalah yang membuat bagian ini bisa dibaca sama sekali**
> `Node *next` adalah pointer ke sebuah struct -- materi Bagian 6.2.3, persis. Menjelajahi list dengan menulis `cur = cur->next;` adalah persis `(*cur).next`, kesetaraan operator-arrow Bab 6, diterapkan dalam sebuah loop. Memeriksa `if (head == nullptr)` sebelum melakukan apa pun dengan `head` adalah disiplin guard-null Bagian 6.2.2, kini melindungi terhadap LIST KOSONG alih-alih pointer yang belum diinisialisasi. Tidak ada satu pun idiom pointer dalam kode bab ini yang belum ditetapkan Bab 6 -- bab inilah tempat kosakata itu akhirnya melakukan pekerjaan struktural yang sesungguhnya.

7.3 Singly Linked List Head-Only

Linked list paling sederhana yang mungkin hanya menyimpan persis satu pointer eksternal: `head`, menunjuk node pertama (atau `nullptr` jika list kosong). Setiap node lain hanya bisa dicapai dengan mengikuti pointer `next` mulai dari `head` -- tidak ada jalan pintas ke tengah atau ke ujung.

7.3.1 init, frontInsertion, frontDeletion, clear

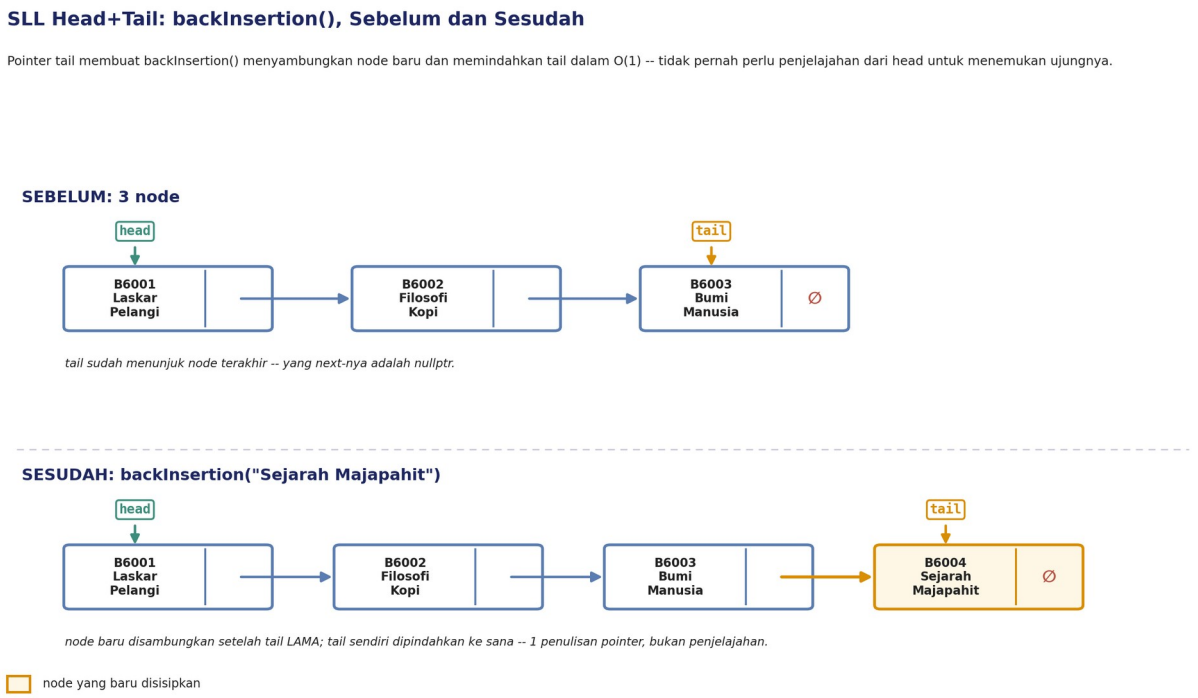
```
void init() {
    head = nullptr;
    count = 0;
}

// O(1): node baru tidak pernah perlu tahu apa pun tentang SISA
// list -- hanya tentang head SAAT INI, yang digantikannya.
void frontInsertion(const Book &value) {
    Node *n = new Node{value, head};   // n->next = head LAMA
    head = n;                          // head kini menunjuk n
    count++;
}

// O(1): menghapus node depan, mengembalikan payload-nya lewat outValue.
bool frontDeletion(Book &outValue) {
    if (head == nullptr) return false;   // guard-null Bagian 6.2.2
    Node *old = head;
    outValue = old->data;
    head = old->next;                    // lewati node yang dihapus
    delete old;
    count--;
    return true;
}
```

Urutan `frontInsertion` penting: `next` node baru diset ke head LAMA LEBIH DULU -- `Node{value, head}` menangkap nilai `head` saat ini pada saat konstruksi -- dan BARU KEMUDIAN `head` sendiri berpindah menunjuk node baru. Membalik urutan ini akan menimpa `head` sebelum node baru sempat tersambung ke apa yang dulu ditunjuknya, kehilangan sisa list secara permanen.

7.3.2 show: Penjelajahan, dan Mengapa Kondisi Berhentinya Bekerja

```
void show() const {
    Node *cur = head;
    while (cur != nullptr) {   // berhenti alami: next node TERAKHIR
        printBook(cur->data);  // sungguh nullptr
        cur = cur->next;
    }
}
```

`while (cur != nullptr)` punya titik berhenti yang alami dan tidak ambigu justru karena ini adalah list NON-circular: next node terakhir sungguh `nullptr`, diset begitu oleh `frontInsertion` saat node itu disisipkan sebagai node terakhir (saat itu). Bagian 7.5 meninjau ulang persis loop ini dan menunjukkan mengapa ia menjadi berbahaya secara aktif begitu list menjadi circular.



Gambar 7.1 — Singly linked list head+tail sebelum dan sesudah backInsertion(). Bagian 7.4 memperkenalkan pointer tail yang menjadi andalan gambar ini; list head-only Bagian 7.3 tidak punya jalan pintas serupa ke ujung.

List head-only TIDAK punya jalan pintas ke ujung

Tidak ada yang mencegah `SLLHeadOnly` menyisipkan node baru di belakang -- tetapi melakukannya secara jujur membutuhkan penjelajahan setiap node yang sudah ada lebih dulu, satu `next` setiap kali, sampai ditemukan node yang next-nya ADALAH `nullptr`. Helper `appendSlow()` milik `demo_head_only.cpp` melakukan persis ini dan menghitung jumlah hop sesungguhnya: menyisipkan buku ke-4, ke-5, dan ke-6 pada list yang tumbuh masing-masing memakan 2, 3, dan 4 langkah penjelajahan -- sungguh O(n), biayanya tumbuh seiring setiap buku yang sudah ada. Inilah motivasi konkret untuk pointer tail Bagian 7.4, bukan klaim notasi-kompleksitas abstrak.

7.4 Singly Linked List Head + Tail

Menambahkan satu field -- pointer `tail`, selalu menunjuk node TERAKHIR -- memperbaiki persis masalah yang baru didemonstrasikan Bagian 7.3, dengan biaya satu field ekstra yang harus dijaga tetap benar pada setiap operasi.

7.4.1 backInsertion: Sungguh O(1)

```
void backInsertion(const Book &value) {
    Node *n = new Node{value, nullptr};
    if (tail == nullptr) {
        head = tail = n;   // list kosong -- n adalah kedua ujung sekaligus
    } else {
        tail->next = n;    // sambungkan tail LAMA ke node baru
        tail = n;          // pindahkan tail ke node baru
    }
    count++;
}
```

Karena `tail` sudah menunjuk node terakhir, `backInsertion` tidak pernah melihat node lain sama sekali -- satu penulisan pointer untuk menyambungkan node baru, satu penulisan pointer untuk memindahkan `tail`. Pemanggilan `backInsertion()` milik `demo_head_tail.cpp` tidak pernah menghitung satu langkah penjelajahan pun, kontras langsung dan terukur dengan `appendSlow()` Bagian 7.3.

7.4.2 frontInsertion dan frontDeletion, dengan Satu Kasus Tepi Baru

`frontInsertion` dan `frontDeletion` bekerja persis seperti di Bagian 7.3 -- kecuali masing-masing kini juga harus menjaga `tail` tetap benar pada satu momen ia bisa menjadi basi: ketika list bertransisi antara kosong dan tidak kosong. Menyisipkan ke list KOSONG membuat satu node baru itu menjadi head SEKALIGUS tail; menghapus node TERAKHIR yang tersisa harus mereset `tail` ke `nullptr`, atau ia akan menggantung, menunjuk memori yang sudah dibebaskan.

7.4.3 backDeletion: Sungguh Tetap O(n)

Inilah poin kompleksitas paling penting bab ini, dan mudah ditebak salah: apakah pointer tail membuat penghapusan node TERAKHIR juga O(1)? Tidak.

```
bool backDeletion(Book &outValue, int *stepsOut = nullptr) {
    if (head == nullptr) return false;
    outValue = tail->data;
    int steps = 0;
    if (head == tail) {          // persis satu node
        delete head;
        head = tail = nullptr;
    } else {
        Node *cur = head;
        while (cur->next != tail) {   // cari node KEDUA-DARI-TERAKHIR
            cur = cur->next;
            steps++;
        }
        delete tail;
        cur->next = nullptr;
        tail = cur;          // node kedua-dari-terakhir menjadi tail baru
    }
    if (stepsOut != nullptr) *stepsOut = steps;
    count--;
    return true;
}
```

Mengapa pointer tail tidak bisa memperbaiki ini

`tail` memberi tahu Anda SEKETIKA node mana yang harus dihapus -- bagian itu sungguh O(1). Yang tidak bisa diberitahukannya adalah siapa yang menunjuk PADA `tail` -- dan sebuah node singly linked tidak punya cara melihat ke belakang. Satu-satunya cara menemukan node yang field next-nya sama dengan `tail` adalah mulai dari `head` dan berjalan maju sampai menemukannya. `demo_head_tail.cpp` mengukur ini secara langsung: menghapus dari list 6-, 5-, lalu 4-node memakan 4, 3, lalu 2 langkah penjelajahan sesungguhnya (persis n - 2), dan menghapus satu node terakhir yang tersisa -- ketika `head == tail` -- memakan 0 langkah, kasus khusus yang sungguhan, bukan artefak pembulatan. Doubly Linked List Bab 9 memperbaiki persis ini, dengan memberi setiap node pointer `prev` juga.

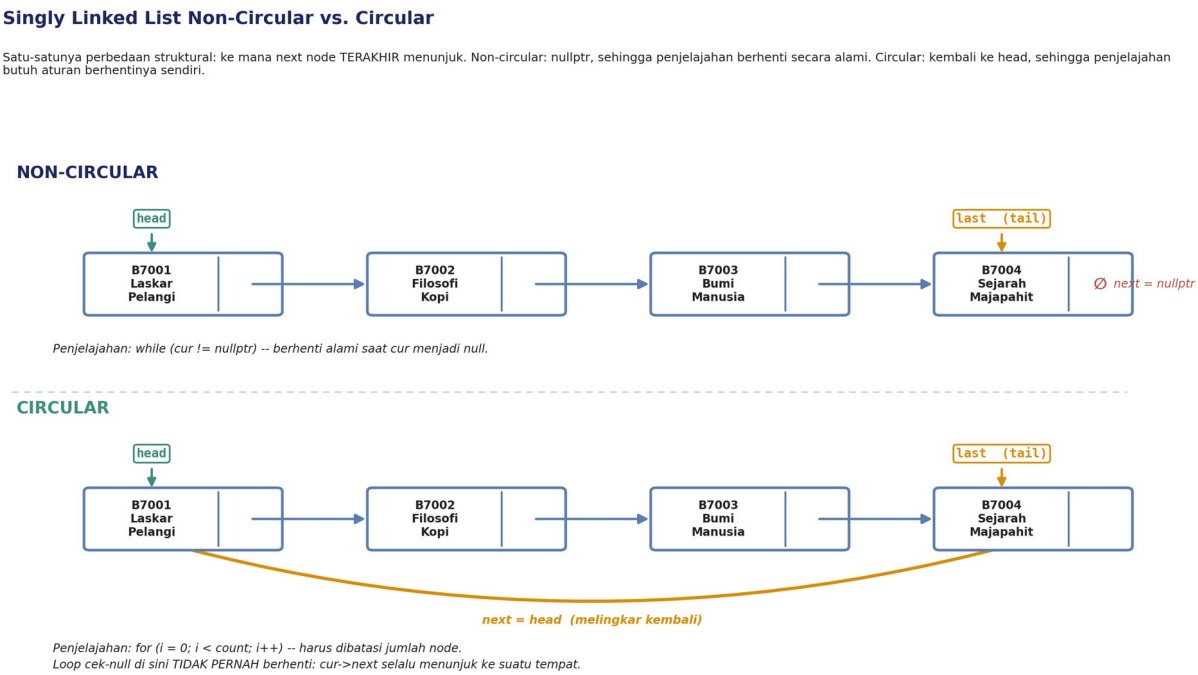
7.5 Circular Singly Linked List (Baru di Bab Ini)

Sebuah circular singly linked list mengubah persis satu hal secara struktural: alih-alih next node terakhir adalah `nullptr`, ia menunjuk kembali ke head. Tidak ada yang lain tentang `Node` yang berubah -- pasangan `Book data; Node *next;` yang sama, hanya field next node terakhirnya menyimpan nilai yang berbeda.

7.5.1 Trik Implementasi Satu-Pointer

Alih-alih melacak `head` DAN `tail` secara terpisah (seperti Bagian 7.4), sebuah circular list hanya butuh SATU pointer eksternal -- lazim disebut `last`, menunjuk tail -- karena head selalu bisa dijangkau sebagai `last->next`. Ini bukan sekadar trik hemat-ruang: ia juga membuat `backInsertion` menjadi O(1) karena alasan mendasar yang sama dengan pointer tail Bagian 7.4, dan inilah implementasi yang sungguh dipakai `CircularSLL` bab ini:

```
void frontInsertion(const Book &value) {
    Node *n = new Node{value, nullptr};
    if (last == nullptr) {
        n->next = n;           // lingkaran SATU node menunjuk dirinya sendiri
        last = n;
    } else {
        n->next = last->next;  // next node baru = head LAMA
        last->next = n;        // next  last = node baru = head baru
    }
    count++;
}

void backInsertion(const Book &value) {
    frontInsertion(value);   // sisipkan tepat setelah last (menjadi head
baru)...
    last = last->next;       // ...lalu pindahkan last: NODE ITU menjadi tail
baru
}
```



Gambar 7.2 — Non-circular vs. circular, berdampingan. Node-node dan tautan majunya identik; satu-satunya perbedaan yang digambarkan di sini adalah ke mana next node TERAKHIR menunjuk.

7.5.2 Mengapa Circular: Penjadwalan Round-Robin

Kasus motivasi alami untuk sebuah circular list adalah apa pun yang dilayani dalam rotasi ROUND-ROBIN -- satu giliran masing-masing, berurutan, lalu rotasi berlanjut kembali ke peserta pertama, selamanya, tanpa giliran "terakhir" yang ditetapkan. Bagian aplikasi Stack-vs-Queue Bab 5 sudah menyebut penjadwalan round-robin (seperti yang dipakai sistem operasi untuk memberi setiap proses yang berjalan bagian waktu CPU yang adil dan bergilir) sebagai aplikasi Queue yang alami; sebuah circular list adalah kecocokan struktural untuk ide yang SAMA, ketika himpunan peserta yang dirotasi itu sendiri yang perlu tumbuh atau menyusut seiring waktu (sebuah proses bergabung atau meninggalkan jadwal, permintaan hold seorang patron ditambahkan atau dipenuhi) alih-alih diproses sekali lalu dibuang seperti isi queue pada umumnya.

Simulasi round-robin milik `demo_circular.cpp` sendiri membuat ini konkret: sebuah ruang baca kecil dengan 4 buku yang di-hold untuk para patron dilayani selama 9 giliran -- lebih banyak giliran daripada jumlah patron -- hanya memakai `CircularSLL::advance()`, yang didefinisikan sebagai `return cur->next;` dan tidak lebih. Rotasinya melingkar lebih dari sekali, dan fungsinya tidak pernah butuh kasus khusus untuk "mencapai ujung", karena dalam circular list tidak ada ujung untuk dicapai.

7.5.3 Penjelajahan Aman: Dibatasi Jumlah Node, Bukan Null

while (cur != nullptr) tidak akan pernah berhenti di sini

Penjelajahan non-circular Bagian 7.3.2 mengandalkan next node terakhir yang sungguh nullptr. Dalam circular list, TIDAK ADA node yang next-nya pernah null -- last->next melingkar kembali ke head, dan setiap node lain pun next-nya menunjuk ke node yang nyata. Sebuah loop while(cur != nullptr) terhadap circular list bukan sekadar salah, ia adalah LOOP TAK TERHINGGA: cur tidak pernah null, sehingga kondisinya tidak pernah menjadi false. Ini bukan peringatan hipotetis -- CircularSLL::show()/clear() dalam kode bab ini sendiri memakai for (int i = 0; i < count; i++) biasa sebagai gantinya, dibatasi oleh jumlah node yang sudah dilacak list, khusus untuk menghindari jebakan ini. demo_all.cpp bahkan menguji 1000 hop paksa mengelilingi circular list sungguhan dan mengonfirmasi tidak satu pun pernah menghasilkan next null -- tidak ada jalan buntu di mana pun untuk diperiksa.

```
void show() const {
    if (last == nullptr) return;   // list kosong -- tidak ada yang dijelajahi
    Node *cur = last->next;        // mulai dari head
    for (int i = 0; i < count; i++) {   // dibatasi JUMLAH, bukan null
        printBook(cur->data);
        cur = cur->next;
    }
}
```

Disiplin praktis yang ditinggalkan bagian ini untuk Anda: sebelum menulis loop penjelajahan apa pun atas sebuah linked list, tanyakan apakah list itu BISA circular. Jika bisa, kondisi berhenti loop itu harus berupa jumlah eksplisit (atau pemeriksaan sentinel yang setara), tidak pernah sekadar perbandingan null.

7.6 Menoleh ke Depan: Doubly Linked List Bab 9

Apa yang didapat dari pointer prev

Bab ini jujur tentang satu biaya nyata: backDeletion() pada singly linked list head+tail tetap O(n), karena menemukan node kedua-dari-terakhir membutuhkan berjalan maju dari head -- sebuah node singly linked sederhananya tidak bisa melihat ke belakang. Bab 9 (Doubly Linked List) menambahkan persis satu field baru, prev, ke setiap node, menunjuk node SEBELUMNYA dengan cara yang sama next menunjuk node berikutnya. Dengan pointer prev, node kedua-dari-terakhir bisa dijangkau dalam satu langkah dari tail (tail->prev), membuat backDeletion() sungguh O(1) juga -- persis penyempurnaan yang direkomendasikan log validasi Lampiran 7.A bab ini sendiri untuk dinantikan. Bab 8 (UTS) memakai materi SLL bab ini bersama materi Stack dan pointer Bab 5-6 sebelum Bab 9 membuat penyempurnaan itu.

7.7 Lampiran 7.A — Log Validasi

Keempat program di bawah ini (`demo_head_only.cpp`, `demo_head_tail.cpp`, `demo_circular.cpp`, `demo_all.cpp`) dikompilasi dengan `g++ -Wall -Wextra -std=c++17` (nol peringatan dari keempatnya) dan benar-benar dieksekusi; transkrip di bawah adalah keluaran terminal yang asli dan tidak diedit.

Untuk pertama kalinya dalam mata kuliah ini, `valgrind` ditemukan sungguh terpasang di lingkungan ini dan sungguh dijalankan terhadap keempat program sebagai pemeriksaan keamanan-memori independen, berdampingan dengan penghitung alokasi `Node::liveCount` bab ini sendiri -- keduanya dilaporkan di sini persis sebagaimana terjadi.

7.A.1 demo_head_only.cpp (kutipan — transkrip lengkap ada di _run_output.txt kode yang dikirim)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_head_only book.cpp demo_head_only.cpp
$ ./demo_head_only
=== Chapter 7: Singly Linked List (Head-Only) ===

--- Part 1: frontInsertion() five times ---
  frontInsertion(Laskar Pelangi          ) -> size() = 1, new head = Laskar
Pelangi
  ...
  frontInsertion(Negeri di Ujung Tanduk  ) -> size() = 5, new head = Negeri di
Ujung Tanduk

--- Part 4: appendSlow() -- the O(n) cost of appending with NO tail pointer ---
  appendSlow(Cantik Itu Luka ) -> 2 traversal step(s) to find the end, node
count now = 4
  appendSlow(Gadis Kretek    ) -> 3 traversal step(s) to find the end, node
count now = 5
  appendSlow(Pulang          ) -> 4 traversal step(s) to find the end, node
count now = 6

--- Part 5: clear() and allocation-count verification ---
Node::liveCount before clear() = 6 (expect 6, matching the true node count)
Node::liveCount after  clear() = 0 (expect 0 -- every node was freed)
list.isEmpty() after clear()   = true
```

7.A.2 demo_head_tail.cpp (kutipan)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_head_tail book.cpp demo_head_tail.cpp
$ ./demo_head_tail
=== Chapter 7: Singly Linked List (Head + Tail) ===

--- Part 4: backDeletion() -- honestly O(n), even WITH a tail pointer ---
  backDeletion() -> true, removed.title = "Negeri di Ujung Tanduk", 4 traversal
step(s), size() now = 5
  backDeletion() -> true, removed.title = "Sejarah Majapahit", 3 traversal
step(s), size() now = 4
  backDeletion() -> true, removed.title = "Bumi Manusia", 2 traversal step(s),
size() now = 3

--- Part 7: allocation-count verification ---
Node::liveCount = 0 (expect 0 -- every node was individually freed by
backInsertion/frontInsertion's matching backDeletion/frontDeletion calls above)
```

7.A.3 demo_circular.cpp (kutipan)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_circular book.cpp demo_circular.cpp
$ ./demo_circular
=== Chapter 7: Circular Singly Linked List ===

--- Part 3: confirming the circle really IS circular ---
Walking 6 hops from head -- more hops than there are nodes (4):

  hop 0: Laskar Pelangi
  hop 1: Filosofi Kopi
  hop 2: Bumi Manusia
  hop 3: Sejarah Majapahit
  hop 4: Laskar Pelangi           <-- same node pointer as hop 0: one full lap
completed
  hop 5: Filosofi Kopi

--- Part 5: round-robin scheduling simulation ---
  Turn  1: serving hold request for "Laskar Pelangi"
  ...
  Turn  9: serving hold request for "Laskar Pelangi"

9 turns served over only 4 patrons -- the rotation wrapped around
(list.size() = 4) more than once, and CircularSLL::advance() never
needed a special case for 'reached the end': in a circular list, there
is no end to reach.
```

7.A.4 demo_all.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_all book.cpp demo_all.cpp
$ ./demo_all
=== Chapter 7 Validation Summary: Singly Linked Lists ===

[Head-only SLL]
  init(): starts empty                                        OK
  frontInsertion() x3: size() == 3                            OK
  frontInsertion(): most recent book is now head              OK
  frontDeletion(): removes the head, returns true             OK
  frontDeletion(): size() decremented to 2                    OK
  clear(): size() back to 0, isEmpty() true                   OK
  clear(): frontDeletion() on empty list safely returns false OK

[Head+tail SLL]
  backInsertion() x3: size() == 3                             OK
  backInsertion(): insertion order preserved (tail == last inserted) OK
  backDeletion(): removes the tail, returns true              OK
  backDeletion(): on a 3-node list took exactly 1 traversal step OK
  backDeletion() on a 1-node list took 0 traversal steps (head==tail case) OK
  list is empty after removing the only remaining node        OK

[Circular SLL]
  backInsertion() x4: size() == 4                             OK
  after exactly size() hops from head, cursor is back at head (circular) OK
  1000 hops never once produced a null next-pointer (no dead end exists) OK
  frontDeletion(): removes the head, returns true             OK
  clear(): size() back to 0, isEmpty() true                   OK

[Memory safety]
  Node::liveCount == 0 after every list above went out of scope  OK

Overall: ALL CHECKS PASSED

$ valgrind --leak-check=full --show-leak-kinds=all ./demo_all
==12195== HEAP SUMMARY:
==12195==     in use at exit: 0 bytes in 0 blocks
==12195==   total heap usage: 12 allocs, 12 frees, 79,904 bytes allocated
==12195== All heap blocks were freed -- no leaks are possible
==12195== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Catatan kejujuran

Setiap hitungan, jumlah langkah, dan baris terminal yang ditunjukkan di atas (di keempat program) berasal langsung dari kompilasi-dan-eksekusi nyata di lingkungan build bab ini, persis seperti yang sudah dilakukan Bab 1-6. Pass validasi bab ini sendiri tidak menemukan bug apa pun dalam logika keempat program -- diungkapkan di sini secara jujur baik ada maupun tidak, sesuai kebijakan tetap mata kuliah ini. Yang BARU bab ini: valgrind diperiksa dan ditemukan sungguh terpasang, sehingga sungguh dijalankan (bukan sekadar disebut tidak tersedia) terhadap keempat program itu, dan setiap eksekusi melaporkan nol kebocoran dan nol error -- konfirmasi independen kedua di luar penghitung alokasi Node::liveCount, keduanya nyata, tidak ada yang direkayasa. Zip kode yang dikirim juga di-ekstrak ulang dan dibangun ulang secara independen dari `make clean && make run` yang bersih untuk memastikan deliverable sesungguhnya, bukan hanya salinan kerja, dikompilasi dan dijalankan secara identik.

7.8 Rangkuman Bab dan Poin-Poin Kunci

- Array dan linked list bertukar trade-off dalam arah berlawanan: array memberi akses acak O(1) dengan biaya ukuran tetap dan penyisipan/penghapusan tengah O(n); linked list memberi penyisipan/penghapusan O(1) pada posisi yang diketahui dengan biaya akses O(n) ke posisi mana pun yang belum dicapai. Tidak ada yang universal lebih baik.
- Sebuah `Node` menggabungkan payload (`Book`) dengan pointer `next` ke node berikutnya. Pointer itu bukan detail implementasi -- untuk sebuah linked list, ia ADALAH struktur datanya. Linked list itu sendiri berasal dari 1955-56 (RAND Corporation, bahasa IPL).
- SLL head-only mendukung `init`/`frontInsertion`/`show`/`frontDeletion`/`clear`, semuanya O(1) kecuali `show` (yang mesti O(n), karena mengunjungi setiap node) -- menyisipkan di belakang membutuhkan penjelajahan O(n), karena tidak ada jalan pintas ke ujung.
- Menambahkan pointer `tail` membuat `backInsertion` sungguh O(1) -- tetapi `backDeletion` tetap O(n), meski ada `tail`, karena menemukan node kedua-dari-terakhir membutuhkan berjalan maju dari `head`; node singly linked tidak bisa melihat ke belakang. Pointer `prev` Bab 9 adalah yang akhirnya memperbaiki ini.
- SLL circular mengubah persis satu hal: next node terakhir menunjuk kembali ke head alih-alih ke `nullptr`. Penjelajahan karenanya harus dibatasi jumlah node (atau aturan berhenti eksplisit yang setara), tidak pernah pemeriksaan null -- `while (cur != nullptr)` terhadap circular list tidak pernah berhenti.
- Penjadwalan round-robin (diisyaratkan bagian aplikasi Queue Bab 5) adalah kasus motivasi alami circular list: merotasi sekumpulan peserta, selamanya, dengan himpunannya sendiri bebas tumbuh atau menyusut.
- Verifikasi keamanan-memori bab ini sendiri memakai dua metode independen, keduanya sungguh dijalankan: penghitung alokasi `Node::liveCount`, dan -- baru tersedia bab ini -- `valgrind --leak-check=full`, yang melaporkan nol kebocoran dan nol error di keempat program.

7.9 Pertanyaan Ulasan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Bab 8 (UTS), yang memakai materi SLL bab ini bersama materi Stack dan pointer Bab 5-6.

1. Seorang teman berkata, "linked list itu murni lebih baik daripada array, karena bisa tumbuh." Memakai tabel trade-off Bagian 7.1, berikan dua situasi konkret di mana array adalah pilihan yang lebih baik, dan jelaskan mengapa dari segi Big-O, bukan hanya intuisi.
2. `SLLHeadOnly` Bagian 7.3 sama sekali tidak punya method `backInsertion`. Jelaskan, memakai helper `appendSlow()` `demo_head_only.cpp`, persis berapa biayanya (dalam langkah penjelajahan) untuk menyisipkan sebuah buku ke list yang sudah punya 10 buku, dan mengapa biaya itu tidak terhindarkan tanpa menambah field baru ke class.

3. Jelaskan, dengan kata-kata Anda sendiri, mengapa `backDeletion()` pada SLL head+tail adalah O(n) meski list-nya punya pointer `tail`. Informasi SPESIFIK apa yang gagal diberikan `tail`, dan mengapa tidak ada node singly linked yang bisa menyediakannya?
4. `show()` sebuah circular list memakai `for (int i = 0; i < count; i++)` alih-alih `while (cur != nullptr)`. Jelaskan secara konkret apa yang akan terjadi jika `CircularSLL::show()` ditulis ulang memakai versi cek-null sebagai gantinya -- telusuri setidaknya 3 iterasi apa yang sesungguhnya akan dilakukan loop itu.
5. Penjadwalan round-robin disebut sebagai aplikasi alami circular list, menghubungkan kembali ke aplikasi Queue Bab 5. Deskripsikan, dengan kata-kata Anda sendiri, satu skenario dunia-nyata lain (bukan dari bab ini atau Bab 5) yang secara alami akan dimodelkan dengan circular list alih-alih non-circular, dan jelaskan apa yang membuat sifat circular menjadi kecocokan yang tepat.
6. Bab 9 akan menambahkan pointer `prev` ke setiap node. Berdasarkan hanya apa yang diajarkan bab ini tentang biaya O(n) `backDeletion()`, prediksi: apa yang akan diizinkan `tail->prev` untuk dilakukan `backDeletion()` sebuah doubly linked list dalam O(1) yang tidak bisa dilakukan `backDeletion()` singly linked bab ini?

Referensi

[1] B.R. Preiss. “Data Structures and Algorithms with Object-Oriented Design Patterns in C++.” John Wiley & Sons, 1998.

[2] P.J. Deitel, H.M. Deitel. “C++ How to Program.” 10th ed., Pearson, 2017. (Linked Lists.)

[3] N. Wirth. “Algorithms + Data Structures = Programs.” Prentice-Hall, 1976.

[4] A. Newell, F.M. Tonge. “An Introduction to Information Processing Language V.” Communications of the ACM, 1960. (Asal-usul historis linked list di RAND Corporation, 1955-56.)

[5] cppreference.com. “Pointer declaration” dan “nullptr.” https://en.cppreference.com/w/cpp/language (accessed August 2026).

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 8 · BAB LATIHAN

Ujian Tengah Semester (UTS)

Tidak ada materi baru di sini — ujian coding individual, open-book, take-home yang mencakup Bab 5 hingga 7: Stack & Queue, Pointer & Fungsi, dan Singly Linked List. Bab ini TIDAK menyertakan subfolder code/ — kelima program di bawah adalah pekerjaan Anda sendiri untuk ditulis dan dikumpulkan.

Tujuan Pembelajaran — setelah mengerjakan bab ini Anda akan mampu:
(1) Mengenali bahwa kelima tugas UTS di bawah semuanya bermuara pada Stack Bab 5, disiplin pointer Bab 6, atau operasi singly linked list (SLL) Bab 7, sehingga tidak ada teori yang benar-benar baru diperlukan untuk Tugas 1, 3, 4, dan 5. (2) Melengkapi kelas Stack berbasis array yang sudah sebagian ditulis, dengan mengisi fungsi anggota yang hilang, menggunakan persis logika push/pop/isEmpty/isFull yang sudah dibahas Bab 5. (3) Menelusuri soal pratinjau pada doubly linked list — struktur dengan DUA pointer per node, bukan satu seperti Bab 7 — dengan memperluas kebiasaan penelusuran pointer Bab 7 secara simetris ke kedua tautan, sambil menyadari bahwa ini adalah pratinjau pembahasan lengkap di Bab 9. (4) Menyusun ulang node-node milik linked list itu sendiri (bukan menyalin datanya ke array) untuk menghasilkan urutan genap-menaik-lalu-ganjil-menurun, menghubungkan operasi relinking Bab 7 dengan ide pengurutan Bab 3. (5) Membalik urutan kata dalam kalimat dan menghapus nilai duplikat hanya menggunakan traversal dan manipulasi pointer SLL Bab 7, tanpa memperkenalkan varian list baru apa pun.

8.1 Ulasan: Kuliah 5-7 Secara Singkat

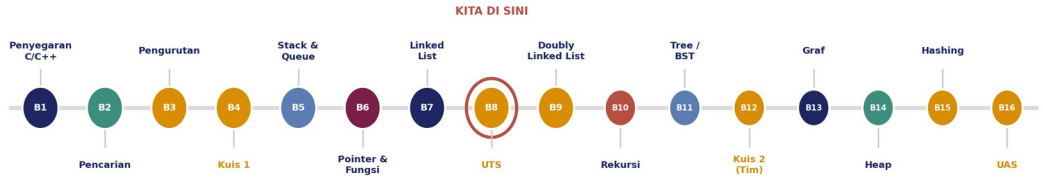
Bab 5 memperkenalkan dua tipe data abstrak yang paling banyak digunakan dalam mata kuliah ini: Stack berbasis array (push, pop, peek, isEmpty, isFull, clear — akses last-in-first-out), didemonstrasikan pada studi kasus berjalan yang sesungguhnya, “Kalkulator Ilmiah” (konversi infix-ke-postfix diikuti evaluasi postfix), dan Queue berbasis array (enqueue, dequeue, isEmpty, isFull, clear — akses first-in-first-out), termasuk masalah klasik “false full” yang dialami queue berbasis array yang naif, serta perbaikannya dengan circular buffer.

Bab 6 melengkapi kisah pointer yang hanya diberi pratinjau di Bab 1: deklarasi dan dereferensi pointer, penanganan null-pointer, pointer-ke-struct (`Book*`, operator `->`), aritmetika pointer yang dibahas ulang dengan semantik lengkap, dan — kontribusi inti bab itu — perbandingan konkret tiga arah antara pass-by-value, pass-by-reference, dan pass-by-pointer, menunjukkan persis yang mana dari ketiganya yang benar-benar memungkinkan sebuah fungsi mengubah data milik pemanggil.

Bab 7 memperkenalkan Singly Linked List (SLL): rangkaian rekam `struct Node { Book data; Node* next; }` yang terhubung murni melalui pointer, bukan melalui memori berdampingan, dalam tiga varian — list head-only (init, frontInsertion, show, frontDeletion, clear), list head+tail yang menambahkan backInsertion O(1), dan SLL sirkular di mana `next` milik node terakhir menunjuk kembali ke node pertama, bukan ke `nullptr`.

Peta Jalan Mata Kuliah DS

Bab ini adalah Bab 8, UTS: checkpoint atas Bab 5-7 (Stack & Queue, Pointer & Fungsi, Singly Linked List), sebelum Doubly Linked List dimulai di Bab 9.



Gambar 8.1 — Bab ini berada di Bab 8 dari enam belas bab peta jalan DS: sebuah checkpoint, bukan topik baru, UTS sebelum Bab 9 melanjutkan dengan Doubly Linked List.

Hampir tidak ada yang baru di bawah ini — dengan satu pengecualian yang jujur

Empat dari lima tugas UTS (Bagian 8.3) adalah penerapan langsung dan biasa dari materi Bab 5, 6, atau 7: Tugas 1 (lengkapi kelas Stack) adalah Stack berbasis array Bab 5; Tugas 3 (urutan genap-lalu-ganjil), Tugas 4 (balik kalimat), dan Tugas 5 (hapus duplikat) semuanya bersandar pada relinking dan traversal SLL Bab 7, dibantu ide pengurutan Bab 3 dan disiplin pointer Bab 6. Tugas 2, bagaimanapun, adalah pengecualian yang sesungguhnya baru: ia mendeskripsikan doubly linked list — struktur dengan pointer `prev` selain `next` — yang secara formal belum diajarkan mata kuliah ini hingga Bab 9. Bagian 8.3.2 jujur tentang hal itu, bukan berpura-pura bahwa Bab 7 sudah membahasnya.

8.2 Cara Menggunakan Bab Latihan Ini

Apa bab ini — dan apa yang bukan

Ini adalah bab praktik, bukan bab materi baru. UTS, seperti asesmen DS sesungguhnya yang menjadi modelnya, adalah ujian coding INDIVIDUAL, open-book, take-home — Anda bekerja sendiri, persis seperti Kuis 1 di Bab 4 (pengumpulan tim hingga tiga mahasiswa hanya berlaku untuk Kuis 2, nanti dalam mata kuliah ini). Bagian 8.3 di bawah menjabarkan kelima tugas UTS, masing-masing disertai callout panduan (THINK, TRAP, atau INSIGHT), bukan solusi lengkap yang sudah dikerjakan. Seperti Bab 4, bab ini TIDAK menyertakan subfolder code/ — kelima program yang Anda tulis untuk UTS adalah pekerjaan individual Anda sendiri, dikompilasi dan dijalankan di komputer Anda sendiri sebelum dikumpulkan.

Sebelum Anda mulai

Untuk setiap tugas di bawah, nyatakan ulang persoalannya dengan kata-kata Anda sendiri terlebih dahulu dan kenali satu ide inti dari Bab 5, 6, atau 7 yang sebenarnya sedang diuji — callout di Bagian 8.3 menyebutkan ide itu secara eksplisit. Khusus untuk Tugas 2, jangan berharap Bab 7 saja sudah mempersiapkan Anda sepenuhnya: baca dulu catatan pratinjau di Bagian 8.3.2, dan perlakukan tugas itu sebagai ajakan untuk bernalar hati-hati dari prinsip dasar tentang pointer kedua, bukan sebagai latihan mengingat.

8.3 Kumpulan Soal UTS

UTS terdiri atas lima tugas coding independen bernilai 10, 30, 30, 15, dan 15 poin secara berurutan, total 100 poin — persis mengikuti distribusi poin UTS DS yang sesungguhnya. Setiap tugas di bawah adalah program C++ yang berdiri sendiri (atau, untuk Tugas 1, sebuah kelas untuk dilengkapi): baca (atau hard-code, jika instruksi mata kuliah Anda sendiri tidak menyebutkan file input) masukan yang dideskripsikan, hasilkan keluaran yang dideskripsikan, dan kompilasi dengan bersih menggunakan `g++ -Wall -Wextra -std=c++17`, toolchain yang sama yang digunakan setiap bab dalam buku ini sejak Bab 1.

Asal Setiap Tugas UTS

UTS terdiri atas lima tugas coding yang dinilai secara individual -- setiap tugas di bawah dapat ditelusuri kembali ke keterampilan spesifik dari Bab 5, 6, atau 7. Tugas 2 adalah pratinjau materi Doubly Linked List Bab 9.

#	Tugas UTS	Berasal dari	Poin
1	Lengkapi kelas Stack	Bab 5 -- Stack & Queue: operasi push/pop/isEmpty/isFull Stack berbasis array	10
2	Lengkapi traversal deque doubly-linked (pratinjau)	Pratinjau Bab 9, dibangun dari kebiasaan traversal node/pointer SLL Bab 7, diperluas ke dua tautan	30
3	Urutkan linked list: genap menaik, lalu ganjil menurun	Bab 7 -- relinking SLL (BUKAN salin ke array) + konsep Pengurutan Bab 3 diterapkan pada node	30
4	Balik urutan kata dalam kalimat via linked list	Bab 7 -- traversal & manipulasi pointer SLL	15
5	Hapus duplikat pada linked list	Bab 7 -- traversal & penghapusan node SLL	15

Total: 100 poin

Gambar 8.2 — Setiap tugas di bagian ini dapat ditelusuri kembali ke keterampilan spesifik dari Bab 5, 6, atau 7 — Tugas 2 adalah pratinjau materi Doubly Linked List Bab 9, bukan pengulangan sesuatu yang sudah diajarkan.

8.3.1 Tugas 1 — Lengkapi Kelas Stack (10 poin)

Anda diberikan kelas `Stack` berbasis array yang sudah sebagian ditulis — anggota data privatnya (array berukuran tetap dan indeks top-of-stack) serta deklarasi kelasnya sudah ada, tetapi satu atau lebih dari `push`, `pop`, `peek`, `isEmpty`, dan `isFull` masih berupa stub kosong. Lengkapi fungsi anggota yang hilang sehingga kelas ini berperilaku persis seperti Stack berbasis array Bab 5: akses last-in-first-out, dengan `push` menolak menulis melewati kapasitas dan `pop`/`peek` menolak membaca stack yang kosong.

> **Petunjuk**
>
> Ini persis wilayah Bab 5 — baca ulang Bagian 5.2 tentang Stack berbasis array sebelum menulis satu baris pun. Seluruh kelas ini bergantung pada satu bilangan bulat, indeks top-of-stack: putuskan sejak awal (dan periksa konvensi mana yang sudah diasumsikan stub yang diberikan) apakah stack kosong berarti indeks `-1` atau indeks `0`, karena logika setiap fungsi anggota lainnya mengikuti langsung dari satu pilihan itu.

Off-by-one pada indeks top adalah bug klasik di sini

Jika kosong berarti `top == -1`, maka `push` harus lebih dulu menaikkan `top` BARU KEMUDIAN menulis `data[top] = value` — menulis dulu baru menaikkan akan diam-diam menimpa indeks 0 selamanya dan tidak pernah maju. Secara simetris, `pop` harus membaca `data[top]` SEBELUM menurunkan `top`, bukan sesudahnya. Telusuri kedua operasi itu dengan tangan pada stack berkapasitas 3 elemen, mendorong 4 nilai, sebelum Anda percaya pada kode Anda sendiri — push ke-4 seharusnya ditolak dengan benar oleh isFull, bukan diam-diam menimpa sesuatu.

Kelas hanya memformalkan apa yang sudah dilakukan versi struct-dan-fungsi Bab 5

Stack Bab 5 ditulis sebagai struct dengan fungsi bebas yang beroperasi padanya; membungkus desain array-berukuran-tetap-plus-indeks-top yang sama dalam `class` C++ yang sesungguhnya dengan data privat dan antarmuka publik (`push`, `pop`, `peek`, `isEmpty`, `isFull`) tidak mengubah apa pun dari logika yang mendasarinya — ini hanya menegakkan, pada waktu kompilasi, bahwa kode di luar kelas tidak dapat menjangkau array atau indeks top secara langsung. Inilah persis alasan Bab 1 mengunci mata kuliah ini ke C++, bukan C biasa, sejak awal.

8.3.2 Tugas 2 — Lengkapi Traversal Deque Doubly-Linked (30 poin)

Pratinjau, bukan latihan mengingat — baca ini sebelum teks Tugas 2 di Bagian 8.3.2

Tugas ini mendeskripsikan deque (double-ended queue) yang dibangun sebagai doubly linked list: setiap node membawa pointer `next` DAN pointer `prev`, sehingga dapat ditelusuri maju dari head ATAU mundur dari tail. Mata kuliah ini secara formal belum mengajarkan doubly linked list hingga Bab 9 — tugas ini adalah pratinjau materi tersebut. Jika Anda merasa ini lebih sulit dibanding Tugas 1, 3, 4, dan 5, itu memang wajar, bukan tanda Anda melewatkan sesuatu di Bab 5, 6, atau 7. Bernalarlah dari kebiasaan SLL Bab 7, digandakan, alih-alih mencari materi di buku ini yang belum ditulis.

Anda diberikan deque berbasis `struct DNode { Book data; DNode* prev; DNode* next; };` yang sudah sebagian disusun, dan diminta melengkapi fungsi traversal yang menelusuri dan mencetak isi deque dua kali: sekali maju (head ke tail, mengikuti pointer `next`) dan sekali mundur (tail ke head, mengikuti pointer `prev`) — tanpa pernah menyalin datanya ke array terpisah terlebih dahulu.

Petunjuk — perluas penelusuran-next Bab 7, jangan menggantinya

Separuh maju dari tugas ini PERSIS traversal SLL Bab 7, tidak berubah: mulai dari head, cetak, `cur = cur->next`, berhenti di `nullptr`. Separuh mundur adalah bayangan cerminnya, menelusuri `prev` alih-alih `next`, dimulai dari tail alih-alih head. Jika constructor atau fungsi insersi deque Anda sudah dengan benar menata KEDUA `next` dan `prev` pada setiap node, fungsi traversal ini tidak membutuhkan ide lebih dari ‘ulangi loop Bab 7, dengan pointer yang lain, dari ujung yang lain.’

Bug paling umum di sini adalah pointer yang menggantung, bukan nilai cetak yang salah
Struktur doubly linked memiliki dua pointer yang harus dijaga konsisten pada setiap insersi, bukan satu — dan mudah sekali dengan benar merangkai `next` di seluruh rantai sambil lupa bahwa `prev` milik node PERTAMA harus secara eksplisit diset ke `nullptr` (tidak ada apa pun sebelum node itu), atau bahwa `prev` milik node yang baru disisipkan harus menunjuk ke node sebelumnya, bukan dibiarkan sesuai nilai inisialisasi kebetulannya. Pointer `prev` yang menggantung atau basi pada node head sama sekali tidak akan membuat traversal maju Anda crash — bug itu baru muncul ketika sesuatu mencoba menelusuri mundur melewati head, dan persis karena itulah bug ini mudah lolos tanpa disadari: uji traversal mundur Anda mulai dari tail hingga benar-benar mencapai `nullptr` di head, bukan hanya beberapa langkah saja.

Ini benar-benar wilayah baru, dan Bab 9 adalah tempatnya dibangun dengan semestinya
Bab 9 memperkenalkan doubly linked list dari nol — tipe node-nya, serta operasi insersi dan penghapusan depan/belakang yang harus menjaga `prev` dan `next` bersamaan — dengan kehati-hatian yang sama seperti Bab 7 memperlakukan singly linked list. Tugas ini hanya meminta Anda MENELUSURI deque yang sudah disusun sebagian oleh kode awal; ia sengaja tidak meminta Anda menulis logika insersinya sendiri, karena itu adalah tugas Bab 9. Jika kode awal yang diberikan untuk tugas ini sudah membangun deque dengan benar, satu-satunya pekerjaan Anda adalah kedua loop traversal di atas.

8.3.3 Tugas 3 — Urutkan Linked List: Genap Menaik, Lalu Ganjil Menurun (30 poin)

Diberikan singly linked list berisi bilangan bulat, susun ulang — dengan merangkai ulang (relinking) node-node yang sudah ada, TIDAK PERNAH dengan menyalin nilai ke array, mengurutkan array tersebut, lalu menyalin kembali — sehingga semua nilai genap muncul lebih dulu dalam urutan menaik, diikuti semua nilai ganjil dalam urutan menurun. Hasilnya harus tetap satu linked list yang dibangun dari node-node YANG SAMA seperti pada input awal.

Tugas ini secara eksplisit melarang jalan pintas array, dan penilaian memeriksa hal itu
Menyalin setiap nilai ke `std::vector<int>`, mengurutkan kedua bagiannya di sana dengan `std::sort`, lalu menuliskan kembali nilainya ke node-node yang sudah ada biasanya akan menghasilkan keluaran cetak yang TERLIHAT BENAR — tetapi itu bukan yang diminta tugas ini, dan biasanya tidak akan mendapat nilai penuh, karena inti latihan ini adalah menunjukkan Anda bisa menyusun ulang STRUKTUR list (pointer), bukan NILAI list. Jika logika inti solusi Anda tetap berfungsi meski `Node::data` diganti tipe buram yang hanya bisa dibandingkan melalui fungsi yang disediakan — bukan disalin langsung — berarti Anda sedang melakukan relinking; jika membutuhkan `data` bisa disalin langsung ke array biasa, berarti Anda tidak.

Petunjuk — pisahkan dulu, urutkan tiap bagian, lalu sambungkan

Rencana tiga langkah yang bersih: (1) telusuri list asli sekali, merangkai ulang tiap node ke salah satu dari dua list baru — list genap dan list ganjil — dengan mengarahkan ulang pointer `next` sambil berjalan, bukan mengalokasikan node baru; (2) urutkan list genap secara menaik dan list ganjil secara menurun, masing-masing menggunakan versi relinking-node dari algoritme pengurutan berbasis perbandingan Bab 3 mana pun yang Anda suka (insertion sort, diterjemahkan ke pointer alih-alih indeks array, biasanya paling cocok untuk singly linked list, karena hanya perlu melihat satu node ke depan); (3) setelah kedua sublist terurut secara internal, sambungkan keduanya dengan mengarahkan `next` milik node terakhir list genap ke node pertama list ganjil.

Waspadai jenis bug ini saat Anda menghitung atau menelusuri hingga akhir list

Jalan pintas yang menggoda adalah melacak berapa banyak nilai genap dan ganjil yang sudah terlihat dengan dua penghitung selama lintasan pertama Anda. Solusi pada proyek ini harus menginisialisasi KEDUA penghitung secara eksplisit ke nol sebelum pemindaian dimulai — penghitung yang tidak diinisialisasi, atau hanya diinisialisasi pada salah satu dari dua jalur kode, menghasilkan nilai yang terlihat masuk akal tetapi salah, dan bug itu tidak selalu muncul dengan cara yang sama dua kali. Terpisah dari itu, waspadai kondisi BERHENTI loop Anda: traversal yang berhenti begitu mencapai node terakhir, alih-alih setelah memprosesnya, secara diam-diam menjatuhkan persis satu elemen — sering kali elemen paling akhir dalam list — tanpa crash atau pesan error. Telusuri logika pisah-dan-sambung Anda sendiri dengan tangan pada list pendek yang elemen TERAKHIRNYA adalah bilangan genap, khususnya memeriksa apakah elemen itu bertahan ke dalam sublist genap akhir Anda; list yang kebetulan berakhir dengan bilangan ganjil bisa menyembunyikan bug ini persis dari Anda.

Mengapa versi relinking-node menghubungkan Bab 3 dan Bab 7

Bab 3 mengajarkan tujuh cara untuk memutuskan “mana dari dua elemen yang lebih dulu”; tidak satu pun dari ketujuh algoritme itu sebenarnya mengharuskan elemen berada dalam array — mereka hanya membutuhkan suatu gagasan “elemen A, elemen B, bandingkan, mungkin tukar atau pindahkan.” Bab 7 sudah menunjukkan bahwa “pindahkan” pada linked list adalah relinking pointer, bukan penyalinan memori. Tugas ini adalah titik pertemuan kedua bab itu: logika pengurutan yang SAMA seperti yang diajarkan Bab 3, diekspresikan melalui operasi pointer Bab 7, bukan pengindeksan array.

8.3.4 Tugas 4 — Balik Urutan Kata dalam Kalimat via Linked List (15 poin)

Diberikan sebuah kalimat (string berisi kata-kata yang dipisahkan satu spasi), pecah menjadi kata-kata individual, bangun singly linked list dengan satu node per kata (dalam urutan asli), balik list tersebut secara in-place, dan cetak kata-kata dari list yang sudah dibalik dipisahkan spasi — menghasilkan kalimat dengan urutan KATA yang terbalik (bukan huruf tiap kata yang terbalik).

Petunjuk — ini pembalikan iteratif Bab 7, tidak berubah

Implementasi SLL Bab 7 sendiri sudah menunjukkan disiplin penelusuran pointer yang dibutuhkan ini: menyimpan tiga pointer — satu untuk node yang sedang diproses, satu untuk node sebelumnya, dan satu disimpan untuk node berikutnya — sebelum menyentuh field `next` mana pun. Pembalikan iteratif standar menyimpan persis `prev`, `cur`, dan `next` sementara: pada setiap langkah, simpan `cur->next` sebelum menimpanya, arahkan `cur->next` ke `prev` sebagai gantinya, lalu majukan `prev` dan `cur` masing-masing satu node. Ketika `cur` mencapai `nullptr`, `prev` adalah head baru Anda.

Timpa next sebelum menyimpannya, dan Anda kehilangan sisa list

Bug paling umum dalam pembalikan list iteratif adalah mengarahkan ulang `cur->next` untuk menunjuk ke belakang SEBELUM membaca dan menyimpan `cur->next` yang asli di tempat lain terlebih dahulu — begitu penimpaan itu terjadi, tidak ada cara sama sekali untuk mencapai sisa list asli, dan pembalikan itu diam-diam menghasilkan list satu atau dua node alih-alih seluruh kalimat. Simpan `next` terlebih dahulu, pada setiap iterasi, tanpa pengecualian untuk node pertama atau terakhir.

Waktu O(n), ruang ekstra O(1) — tidak perlu array, tidak perlu rekursi

Pembalikan ini menyentuh setiap node persis sekali dan hanya membutuhkan tiga variabel pointer terlepas dari seberapa panjang kalimatnya — ia tidak membutuhkan penyalinan kata ke array dan mencetak array itu terbalik (yang akan berfungsi, tetapi menghindari keterampilan manipulasi pointer yang sesungguhnya diuji tugas ini), dan tidak membutuhkan rekursi (yang akan diperkenalkan dengan semestinya di Bab 10, dan yang BISA membalik list dengan elegan, tetapi tidak diperlukan di sini dan menambah ruang call-stack O(n) yang dihindari loop sederhana ini).

8.3.5 Tugas 5 — Hapus Duplikat pada Linked List (15 poin)

Diberikan singly linked list berisi nilai (tidak harus terurut), hapus setiap kemunculan duplikat sehingga setiap nilai berbeda muncul tepat sekali, mempertahankan kemunculan PERTAMA setiap nilai dan membuang pengulangan berikutnya — dengan melepaskan tautan (unlink) dan membebaskan (free) node duplikat, bukan dengan membangun list baru dari nol.

Petunjuk — dua strategi yang jujur, pilih berdasarkan apa yang didukung tipe data list Anda

Tanpa memori ekstra, bandingkan setiap node dengan setiap node yang muncul setelahnya (traversal bersarang), melepaskan tautan dan membebaskan node berikutnya mana pun yang nilainya sudah pernah terlihat — benar, dan solusi $O(n^2)$ yang sepenuhnya sah untuk tugas ini. Dengan sedikit memori ekstra — `std::unordered_set<T>` yang mencatat setiap nilai yang sudah terlihat, trik “sudah pernah dilihat” yang sama seperti yang digunakan Kuis 1 Bab 4 untuk menemukan karakter berulang pertama — satu traversal tunggal yang berjalan dalam O(n) dimungkinkan sebagai gantinya, ASALKAN tipe elemen list-nya bisa di-hash. Keduanya dapat diterima; ketahuilah yang mana yang Anda tulis dan mengapa, dan siap menjelaskan trade-off-nya.

Menghapus node dengan benar butuh relinking DAN free — lewatkan salah satu dan Anda punya bug

Ketika node `q` (yang menyimpan nilai duplikat) ditemukan dan harus dihapus, `next` milik predecessor-nya harus diarahkan ulang ke `q->next` LEBIH DULU — jika tidak, sisa list menjadi tidak terjangkau begitu `q` dibebaskan — dan baru setelah itu `q` sendiri boleh didealokasi. Membebaskan `q` sebelum merangkai ulang predecessor-nya membuat segala sesuatu setelahnya menjadi yatim; merangkai ulang tanpa pernah membebaskan `q` menghapusnya dari traversal dengan benar tetapi membocorkan memorinya secara diam-diam. Waspadai juga kasus di mana node yang harus Anda lewati untuk memajukan pointer traversal adalah node yang baru saja Anda hapus versus yang baru saja Anda pertahankan — memajukan dengan salah pada salah satu cabang bisa melewatkan pemeriksaan suatu nilai atau memprosesnya dua kali.

Ide O(n) yang sama dari Bab 4, kini diterapkan pada node alih-alih karakter

Kuis 1 Bab 4 meminta KARAKTER berulang pertama dalam sebuah string, dalam O(n), menggunakan satu lintasan dengan catatan “sudah pernah dilihat”. Tugas ini adalah ide yang sama, digeneralisasi: “elemen”-nya kini adalah nilai node linked list alih-alih karakter string, dan operasi pada pengulangan adalah PENGHAPUSAN, bukan sekadar pelaporan — tetapi pertukaran yang mendasarinya (sedikit memori, untuk waktu yang jauh lebih sedikit) identik.

8.4 Format dan Penilaian UTS

UTS adalah ujian coding INDIVIDUAL, open-book, take-home — bukan tugas tim, persis seperti Kuis 1 di Bab 4 (pengumpulan tim hingga tiga mahasiswa berlaku untuk Kuis 2 saja, nanti dalam mata kuliah ini). "Open-book" berarti buku teks, catatan Anda sendiri, dan materi referensi C++ standar (seperti cppreference.com) semuanya diperbolehkan selama Anda mengerjakan; ini TIDAK berarti Anda boleh menyalin kode mahasiswa lain atau mengumpulkan pekerjaan yang bukan benar-benar milik Anda sendiri — ujian open-book tetap menguji apakah ANDA dapat menerapkan apa yang telah diajarkan mata kuliah ini, bekerja secara independen.

Rubrik point-per-subtask milik DS sendiri

Seperti Kuis 1, UTS dinilai tugas demi tugas, dalam poin utuh, bukan dengan template persentase-berbobot Design/Implementation/Analysis/Conclusion. Kelima tugas bernilai 10, 30, 30, 15, dan 15 poin secara berurutan — pembagian yang SENGAJA TIDAK RATA yang mencerminkan tingkat kesulitan dan cakupan masing-masing tugas, berbeda dari bentuk rata 25-poin-tiap-tugas milik Kuis 1 — diberikan untuk kode yang kompilasi dengan bersih dan menghasilkan keluaran yang benar untuk persyaratan yang dinyatakan (termasuk, untuk Tugas 3, benar-benar menyusun ulang struktur list alih-alih mengurutkan array salinan, dan untuk Tugas 2, menjaga dengan benar `prev` maupun `next` di sepanjang traversal). Tidak ada komponen "analisis" atau "kesimpulan" terpisah pada ujian ini — kode dan keluarannya yang benar adalah deliverable-nya.

Tugas	Apa yang diuji	Poin

1. Lengkapi kelas Stack	Stack berbasis array yang benar — push/pop/isEmpty/isFull, tanpa off-by-one pada indeks top	10
2. Traversal deque doubly-linked (pratinjau)	Traversal maju DAN mundur yang benar, dengan prev terjaga pada setiap node termasuk head	30
3. Urutan: genap naik, lalu ganjil turun	Penyusunan ulang yang benar via relinking node (bukan salin array) menjadi urutan genap-lalu-ganjil	30
4. Balik kalimat via linked list	Pembalikan SLL in-place iteratif yang benar menghasilkan urutan kata terbalik	15
5. Hapus duplikat pada linked list	Deduplikasi yang benar dengan melepas tautan dan membebaskan node duplikat, kemunculan pertama dipertahankan	15
Total		100

Kumpulkan kelima program sebagai file `.cpp` terpisah yang masing-masing dapat dikompilasi sendiri (ditambah header bersama apa pun yang dibutuhkan solusi Anda), masing-masing menunjukkan keluarannya sendiri yang benar saat dijalankan — program yang tidak kompilasi tidak mendapat nilai untuk tugas tersebut, seberapa pun dekat logikanya dengan yang benar, jadi sisakan waktu untuk benar-benar membangun dan menjalankan kelimanya sebelum mengumpulkan, persis disiplin yang diterapkan pada setiap contoh kode di Bab 1 hingga 7.

8.5 Ringkasan Bab

- UTS tidak memperkenalkan materi baru untuk empat dari lima tugasnya — ini adalah checkpoint atas Bab 5 (Stack & Queue), Bab 6 (Pointer & Fungsi), dan Bab 7 (Singly Linked List), yang diulas Bagian 8.1.
- Tugas 2 adalah pengecualian yang sengaja dan diungkapkan secara terbuka: ia adalah pratinjau materi Doubly Linked List Bab 9, bukan menguji sesuatu yang sudah diajarkan — Bagian 8.3.2 menyatakan hal itu secara eksplisit, bukan berpura-pura sebaliknya.
- Ini adalah ujian coding INDIVIDUAL, open-book, take-home — bukan tugas tim; pengumpulan tim hingga tiga mahasiswa hanya berlaku untuk Kuis 2, nanti dalam mata kuliah ini.
- Lima tugas bernilai 10, 30, 30, 15, dan 15 poin (total 100): lengkapi kelas Stack (Bab 5), traversal deque doubly-linked (pratinjau Bab 9, dibangun dari kebiasaan Bab 7), urutan genap-lalu-ganjil via relinking node (Bab 3 dan 7), pembalikan kalimat (Bab 7), dan penghapusan duplikat (Bab 7, menggemakan ide O(n) “sudah pernah dilihat” Bab 4).
- Penilaian mengikuti rubrik point-per-subtask milik DS sendiri — nilai poin yang sengaja tidak rata per tugas mencerminkan tingkat kesulitannya, bukan template rata atau persentase-berbobot.
- Bab ini TIDAK menyertakan subfolder code/: kelima program adalah pekerjaan individual masing-masing mahasiswa, dikompilasi dengan toolchain g++ -Wall -Wextra -std=c++17 yang sama yang digunakan sepanjang buku ini dan benar-benar dijalankan sebelum dikumpulkan.

Program yang kompilasi tidak sama dengan program yang benar — dan program yang benar pada satu input yang Anda coba tidak sama dengan program yang benar secara umum. Uji kelima solusi Anda dengan lebih dari satu input, termasuk setidaknya satu kasus yang sengaja dibuat janggal (stack yang didorong tepat sampai kapasitas untuk Tugas 1, deque satu node untuk Tugas 2, list yang elemen terakhirnya genap untuk Tugas 3, kalimat satu kata untuk Tugas 4, dan list di mana setiap nilai adalah duplikat dari yang pertama untuk Tugas 5) sebelum Anda mengumpulkan.

Lampiran 8.A — Daftar Periksa Mandiri (Tidak Dinilai)

Sebelum mengumpulkan kelima program Anda, jalankan melalui daftar periksa ini. Prosesnya hanya beberapa menit dan menangkap sebagian besar kesalahan yang paling sering dilihat penilai pada pengumpulan UTS pertama.

- Apakah kelima program kompilasi dengan bersih menggunakan `g++ -Wall -Wextra -std=c++17`, tanpa satu pun warning?
- Tugas 1: sudahkah Anda menelusuri push/pop dengan tangan pada stack berkapasitas 3 yang mendorong 4 nilai — apakah push ke-4 ditolak dengan benar oleh isFull, bukan menimpa sesuatu?
- Tugas 2: apakah traversal mundur Anda benar-benar mencapai nullptr di head — sudahkah Anda memeriksa secara khusus pointer prev milik node head sendiri, bukan hanya beberapa langkah dari tail?
- Tugas 3: bisakah logika inti solusi Anda tetap berfungsi jika Node::data diganti tipe yang hanya bisa dipindahkan, bukan disalin ke array biasa? Jika tidak, Anda mungkin sedang melakukan sort array yang disamarkan.
- Tugas 3: apakah logika pisah-dan-sambung Anda dengan benar mempertahankan nilai genap yang kebetulan menjadi node TERAKHIR dalam list asli?
- Tugas 4: sudahkah Anda menyimpan cur->next ke variabel sementara SEBELUM menimpa cur->next pada setiap iterasi loop pembalikan Anda, tanpa pengecualian khusus?
- Tugas 5: ketika Anda menghapus node duplikat, apakah Anda merangkai ulang next milik predecessor-nya SEBELUM membebaskan node yang dihapus, dalam urutan itu, setiap kali?
- Tugas 5: apakah traversal Anda dengan benar maju melewati baik node yang baru dihapus maupun node yang baru dipertahankan, tanpa melewatkan atau memproses ulang salah satunya?
- Sudahkah Anda menghapus atau mengomentari baris debug `printf`/`cout` yang tersisa dan bukan bagian dari format keluaran yang diminta?
- Apakah setiap file yang Anda kumpulkan benar-benar pekerjaan individual Anda sendiri, ditulis untuk UTS ini — bukan disalin dari mahasiswa lain, solusi tahun sebelumnya, atau alat AI yang tidak diizinkan oleh kebijakan integritas akademik mata kuliah Anda sendiri?

Referensi

[1] Format asesmen bab latihan ini dimodelkan langsung dari UTS mata kuliah ini yang sesungguhnya (EF234201 Struktur Data): ujian coding individual, open-book, take-home berisi lima tugas bernilai 10, 30, 30, 15, dan 15 poin, dinilai tugas demi tugas, bukan dengan rubrik persentase-berbobot. Format ini digunakan ulang apa adanya di sini; kerangka tugas di atas (yang mengulas ulang materi Bab 5-7 yang sesungguhnya, jujur bahwa Tugas 2 adalah pratinjau Bab 9, dan mengganti masalah kualitas kode yang diketahui pada model jawaban asli ujian ini dengan panduan generik yang sudah diperbaiki) telah diadaptasi untuk buku teks ini.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” Edisi ke-3, MIT Press, 2009. (Stack dan queue, Bab 10; struktur data dasar dan list berbasis pointer, Bab 10; pengurutan, Bab 6–8.)

[3] cppreference.com. “std::unordered_set,” “Pointer declaration,” “Member specification (class).” https://en.cppreference.com/w/cpp (diakses Agustus 2026).

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 9

Doubly Linked List

UTS Bab 8 meminta Anda menjelajahi doubly linked list sebelum mata kuliah ini mengajarkan apa itu -- dan mengungkapkannya secara jujur sebagai pratinjau, bukan berpura-pura Bab 7 sudah membahasnya. Bab ini menutup celah itu. Setiap baris kode di sini benar-benar dikompilasi dengan g++ -Wall -Wextra -std=c++17, benar-benar dijalankan, dan diperiksa secara independen dengan valgrind -- transkrip lengkapnya direproduksi di Lampiran 9.A.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Menjelaskan persis apa yang TIDAK bisa dilakukan struktur singly linked list yang hanya next (penjelajahan mundur sama sekali; backDeletion O(1) bahkan dengan pointer tail) dan mengapa satu field baru, prev, memperbaiki keduanya sekaligus. (2) Mendeklarasikan Node doubly linked (Book data; Node *prev; Node *next;) dan DoublyLinkedList headed (head+tail), sesuai dengan cakupan genuine materi mentah sendiri tentang definisi/ilustrasi/deklarasi/headed-init. (3) Mengimplementasikan dan menelusuri frontInsertion/backInsertion/frontDeletion/backDeletion pada DLL headed, memperbarui dengan benar KEEMPAT field pointer yang terpengaruh per operasi, bukan dua, dan menjelaskan secara konkret mengapa backDeletion akhirnya O(1). (4) Mengimplementasikan dan menelusuri penjelajahan dua arah (showForward/showBackward) dan search linear maju. (5) Mengimplementasikan dan menelusuri circular DLL, menjelaskan apa yang berubah karena sifat circular di KEDUA arah sekaligus (next node terakhir melingkar ke head DAN prev head melingkar ke node terakhir), serta melakukan penjelajahan aman dibatasi jumlah node dan penyisipan/penghapusan yang sadar-seam di kedua arah. (6) Memutuskan, untuk skenario tertentu, apakah pointer prev tambahan DLL sepadan dengan overhead memorinya dibandingkan SLL -- memakai riwayat back/forward browser sebagai contoh penentu yang konkret.

9.1 Ulasan: Apa yang Tidak Bisa Dilakukan Singly Linked List Bab 7

Bab 7 membangun singly linked list head+tail dan jujur tentang persis satu keterbatasan yang tidak bisa direkayasa: backDeletion() tetap O(n) meski ada pointer tail, karena menghapus node terakhir membutuhkan mengetahui node KEDUA-dari-terakhir, dan node singly linked tidak punya cara melihat ke belakang. Satu-satunya cara menemukan "siapa yang menunjuk pada tail" adalah mulai dari head dan berjalan maju sampai cur->next == tail -- biaya nyata dan terukur yang tumbuh seiring list (demo Bab 7 sendiri mengukur persis n - 2 langkah penjelajahan untuk list berukuran n).

Keterbatasan kedua yang terkait tidak dinyatakan di Bab 7 karena belum ada yang membutuhkannya: singly linked list hanya bisa dijelajahi dalam persis satu arah. Begitu cur = cur->next berpindah melewati sebuah node, tidak ada cara kembali ke sana kecuali memulai lagi dari head. Kedua keterbatasan itu berasal dari akar penyebab yang persis sama -- Node yang hanya pernah menunjuk maju -- dan keduanya diperbaiki oleh penambahan satu field yang persis sama.

Satu field baru, dua masalah terpecahkan sekaligus

Mudah untuk mengira doubly linked list butuh desain yang sepenuhnya baru untuk mendukung penjelajahan mundur dan backDeletion O(1). Tidak. Setiap operasi yang diajarkan bab ini dibangun dari kosakata frontInsertion/backInsertion/frontDeletion/backDeletion/penjelajahan yang PERSIS SAMA yang sudah ditetapkan Bab 7 -- diperluas, bukan diganti, oleh satu field pointer baru per node.

9.2 Node: Satu Pointer Baru, prev

Node sebuah doubly linked list adalah Node Bab 7 ditambah persis satu field:

```
struct Node {
    Book data;   // payload -- struct Book Bab 1, tidak berubah
    Node *prev;  // pointer ke node SEBELUMNYA, atau nullptr jika tidak ada
    Node *next;  // pointer ke node BERIKUTNYA, atau nullptr jika tidak ada
};
```

next berarti persis seperti artinya di Bab 7. prev adalah cerminannya: pointer ke node mana pun yang datang tepat sebelum node ini dalam list, atau nullptr jika node ini adalah yang pertama. Simetrinya adalah inti persoalannya -- untuk dua node bersebelahan A dan B mana pun di mana A.next == B, harus SELALU juga benar bahwa B.prev == A. Setiap operasi di bab ini ada untuk menjaga simetri itu tetap benar setelah setiap penyisipan dan penghapusan, di kedua arah, termasuk di kedua ujung list.

Mengapa ini adalah keterampilan khas-DLL paling penting

frontInsertion sebuah singly linked list menyentuh DUA field pointer (next node baru, dan head). frontInsertion sebuah doubly linked list menyentuh EMPAT (next DAN prev node baru, prev head lama, DAN head itu sendiri) -- lihat Bagian 9.4. Melupakan satu saja dari keempatnya membuat struktur berada dalam keadaan di mana penjelajahan maju terlihat sempurna baik-baik saja tetapi penjelajahan mundur diam-diam rusak, atau sebaliknya. Panduan TRAP UTS Bab 8 sudah menyebut langsung area risiko ini: "symmetric prev/next maintenance." Bab ini mengubah peringatan itu menjadi disiplin konkret -- dan, dalam kode yang dikirim, menjadi pemeriksaan otomatis (verifyLinks(), Lampiran 9.A) yang dijalankan setelah setiap mutasi, bukan sekadar aturan untuk diingat.

9.3 Deklarasi dan Inisialisasi Headed (Head+Tail)

Materi ajar mentah di balik mata kuliah ini membahas persis sebanyak ini dari doubly linked list, dalam enam slide genuine, sebelum penulisannya sendiri berhenti: definisi node, ilustrasinya, deklarasi class headed (head+tail), dan inisialisasi. Bagian 9.4 dan seterusnya -- setiap operasi penyisipan, penghapusan, pencarian, dan penjelajahan, plus seluruh varian circular -- adalah konten baru yang ditulis bab ini sendiri, melengkapi apa yang ditinggalkan belum selesai oleh materi mentah itu dan apa yang harus dipratinjau secara jujur oleh Bab 8, bukan diajarkan.

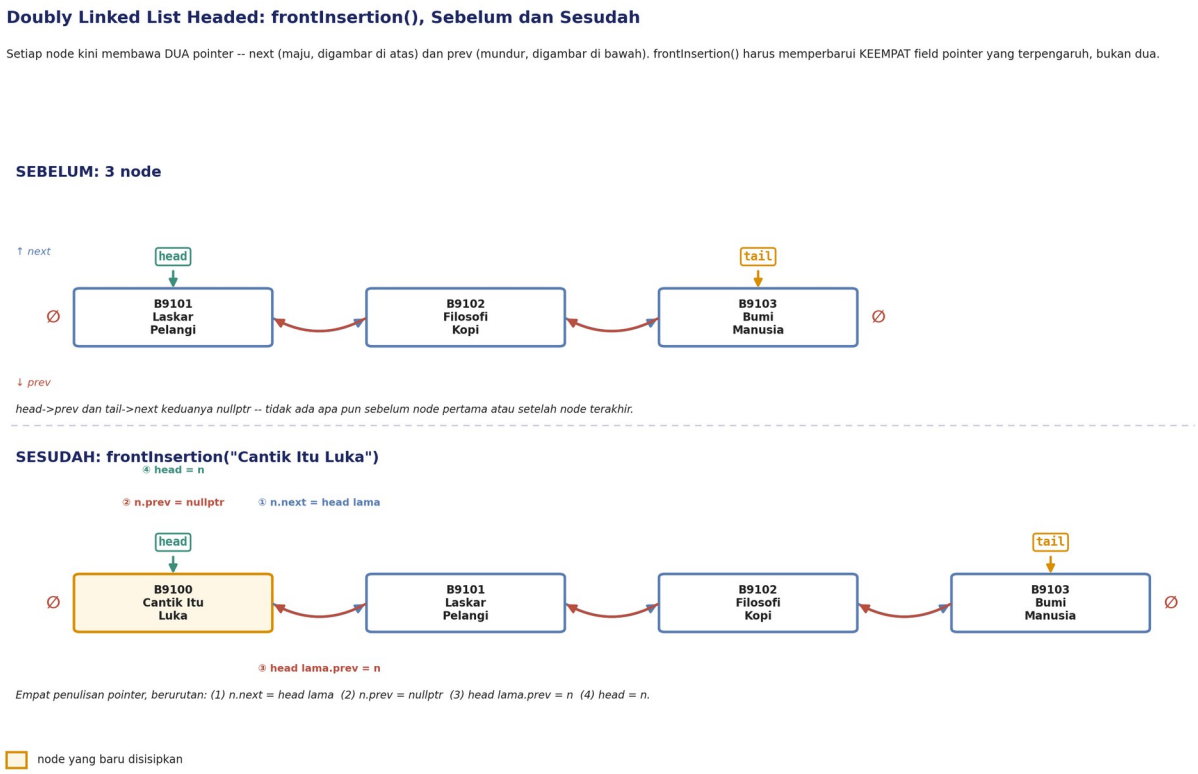
```
class DoublyLinkedList {
public:
    DoublyLinkedList() { init(); }
    ~DoublyLinkedList() { clear(); }

    void init() {
        head = nullptr;
        tail = nullptr;
        count = 0;
    }
    // ... frontInsertion, backInsertion, frontDeletion, backDeletion,
    // showForward, showBackward, search, clear -- Bagian 9.4-9.6

private:
    Node *head;
    Node *tail;
    int count;
};
```

Berbeda dari Bab 7, tidak ada tahap "head-only" terpisah di sini -- cakupan genuine materi mentah sendiri langsung menuju deklarasi headed, jadi bab ini pun begitu. Setiap operasi di bawah menjaga BAIK head DAN tail tetap benar, DAN BAIK next DAN prev tetap benar, pada setiap pemanggilan.

9.4 Penyisipan Depan dan Belakang: Empat Pointer, Bukan Dua

```
void frontInsertion(const Book &value) {
    Node *n = new Node{value, nullptr, head};
    if (head != nullptr) {
        head->prev = n;   // head LAMA kini menunjuk balik ke n
    } else {
        tail = n;         // list kosong -- n adalah kedua ujung sekaligus
    }
    head = n;             // head sendiri pindah TERAKHIR
    count++;
}

void backInsertion(const Book &value) {
    Node *n = new Node{value, tail, nullptr};
    if (tail != nullptr) {
        tail->next = n;
    } else {
        head = n;         // list kosong -- n adalah kedua ujung sekaligus
    }
    tail = n;
    count++;
}
```

Empat penulisan frontInsertion, dalam urutan yang harus terjadi: (1) next node baru n diset ke head LAMA; (2) prev n diset ke nullptr, karena n kini yang pertama; (3) JIKA list tidak kosong, prev head lama dialihkan ke n -- inilah satu penulisan yang tidak pernah dibutuhkan frontInsertion singly linked; (4) head itu sendiri pindah ke n, selalu terakhir. backInsertion adalah cerminan persisnya, memperbarui tail alih-alih head.



Gambar 9.1 — DLL headed sebelum dan sesudah frontInsertion(), dengan BAIK next (panah atas) MAUPUN prev (panah bawah) digambar eksplisit. Keempat penulisan pointer diberi nomor sesuai urutan yang harus terjadi.

> **Kasus list kosong adalah tempat simetri pertama kali rusak**
>
> Ketika list kosong, SATU node baru harus menjadi head DAN tail sekaligus -- lewatkan ini dan salah satu antara tail tetap nullptr sementara head diset (sehingga panggilan backInsertion() berikutnya membaca tail null), atau sebaliknya. Bagian 1 demo_dll.cpp benar-benar membangun list dari kosong dan mengonfirmasi tail terlacak dengan benar pada SETIAP panggilan backInsertion(), bukan hanya yang pertama.

9.5 Penghapusan Depan dan Belakang: Hasilnya — O(1) Akhirnya

```
bool frontDeletion(Book &outValue) {
    if (head == nullptr) return false;
    Node *old = head;
    outValue = old->data;
    head = old->next;
    if (head != nullptr) {
        head->prev = nullptr;  // head baru tidak punya pendahulu lagi
    } else {
        tail = nullptr;        // itu satu-satunya node
    }
    delete old;
    count--;
    return true;
}

bool backDeletion(Book &outValue) {
    if (tail == nullptr) return false;
    Node *old = tail;
    outValue = old->data;
    tail = old->prev;   // <-- SATU pembacaan yang tidak bisa dilakukan SLL Bab
7
    if (tail != nullptr) {
        tail->next = nullptr;
    } else {
        head = nullptr;
    }
    delete old;
    count--;
    return true;
}
```

Perbandingan persis dengan Bab 7

SLLHeadTail::backDeletion() Bab 7 membutuhkan loop while (cur->next != tail) mulai dari head, karena node singly linked tidak punya cara menjawab "siapa yang menunjuk padaku?" kecuali diminta berjalan maju dan memeriksa. Di sini, tail->prev menjawab pertanyaan itu persis secara langsung, dalam satu pembacaan pointer, tidak peduli seberapa panjang list-nya. demo_dll.cpp mengonfirmasi ini secara konkret: setelah backDeletion(), tail baru diverifikasi PERSIS node yang ditunjuk tail->prev sebelum pemanggilan -- penghitung langkah penjelajahan bahkan tidak dibutuhkan, karena tidak ada penjelajahan untuk dihitung.

Kedua penghapusan membawa kehati-hatian transisi-kosong yang sama seperti penyisipan: menghapus satu-satunya node yang tersisa harus mereset pointer ujung LAIN ke nullptr juga, atau ia akan menggantung.

9.6 Pencarian dan Penjelajahan Dua Arah

```
void showForward() const {
    Node *cur = head;
    while (cur != nullptr) { printBook(cur->data); cur = cur->next; }
}
void showBackward() const {
    Node *cur = tail;
    while (cur != nullptr) { printBook(cur->data); cur = cur->prev; }
}
Node *search(const char *id, int *stepsOut = nullptr) const {
    Node *cur = head; int steps = 0;
    while (cur != nullptr) {
        steps++;
        if (std::strcmp(cur->data.id, id) == 0) { if (stepsOut) *stepsOut =
steps; return cur; }
        cur = cur->next;
    }
    if (stepsOut) *stepsOut = steps;
    return nullptr;
}
```

showBackward() adalah satu penjelajahan yang tidak bisa ditawarkan SLL Bab 7 dengan biaya APA PUN, bahkan biaya O(n) sekalipun -- node singly linked sederhananya tidak punya field prev untuk diikuti. Di sini ia persis sealami showForward(): mulai dari tail alih-alih head, ikuti prev alih-alih next. demo_dll.cpp mengonfirmasi showBackward() sungguh mencetak persis kebalikan urutan showForward() pada eksekusi nyata.

Operasi	SLL, head+tail (Bab 7)	DLL, headed (bab ini)
frontInsertion / backInsertion	O(1)	O(1) -- 2 penulisan pointer lebih banyak
frontDeletion	O(1)	O(1)
backDeletion	O(n) -- harus berjalan dari head	O(1) -- tail->prev, satu pembacaan
Penjelajahan maju / search	O(n)	O(n)
Penjelajahan mundur	tidak mungkin dengan biaya apa pun	O(n)
Memori ekstra per node	-- (satu pointer)	+1 pointer (prev)

9.7 Circular Doubly Linked List (Baru di Bab Ini)

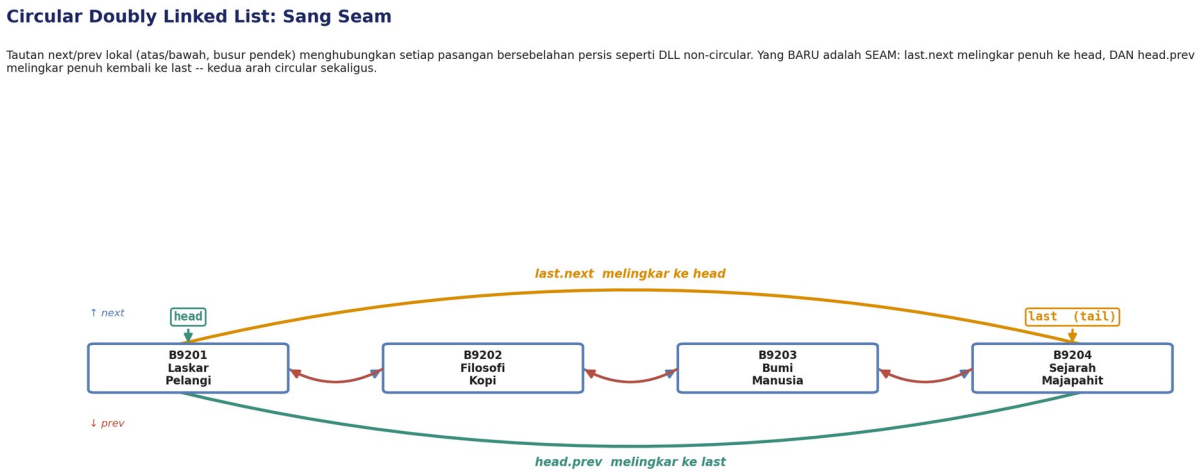
Circular SLL Bab 7 mengubah satu hal: next node terakhir melingkar ke head alih-alih ke nullptr. Circular DLL mengubah DUA hal sekaligus, secara simetris -- next node terakhir tetap melingkar ke head, DAN prev head kini juga melingkar ke node terakhir. Kedua arah menjadi circular bersama-sama, yang persis membuat wraparound dua arah mungkin: pikirkan tombol "next track" / "previous track" pemutar musik, masing-masing melingkar selamanya di arahnya sendiri, tanpa lagu pertama atau terakhir yang ditetapkan dari sudut pandang pemutarnya.

Lecture 08b ("DLL Circular") duplikat milik materi ajar mentah adalah salinan persis body SLL Lecture 06 -- bukan konten baru sama sekali. Semua di bagian ini menggantikan duplikat itu dengan materi yang genuinely ditulis.

Mengikuti preseden CircularSLL Bab 7, hanya SATU pointer eksternal yang dibutuhkan: last, menunjuk tail, karena head selalu bisa dijangkau sebagai last->next.

```
void frontInsertion(const Book &value) {
    Node *n = new Node{value, nullptr, nullptr};
    if (last == nullptr) {
        n->next = n; n->prev = n;   // lingkaran SATU node menunjuk dirinya, dua
arah
        last = n;
    } else {
        Node *headOld = last->next;
        n->next = headOld;   // next n = head LAMA
        n->prev = last;      // prev n = last
        headOld->prev = n;   // prev head LAMA kini menunjuk n, bukan last
        last->next = n;      // next last = n = head baru
    }
    count++;
}
```

Empat penulisan pointer terjadi di seam saja -- satu lebih banyak dari frontInsertion non-circular -- karena tidak ada ujung nullptr untuk dibiarkan begitu saja; kedua tetangga di kedua sisi seam harus diperbarui. backDeletion() pada circular DLL adalah O(1) karena alasan yang persis sama dengan versi non-circular Bagian 9.5: last->prev sudah MENJADI node yang harus menjadi tail baru, circular ataupun tidak.



Gambar 9.2 — Seam sebuah circular DLL: last.next melingkar maju ke head, dan head.prev melingkar mundur ke last, sekaligus.

Penjelajahan harus tetap dibatasi jumlah, di KEDUA arah

Persis seperti CircularSLL Bab 7, tidak ada next node mana pun yang pernah nullptr di sini -- dan kini tidak ada prev node mana pun yang pernah nullptr juga. Loop while (cur != nullptr) di kedua arah tidak akan pernah berhenti. showForward()/showBackward() pada CircularDLL keduanya dibatasi count, loop for biasa, tidak pernah cek null. demo_circular_dll.cpp menguji 1000 hop paksa MAJU dan 1000 hop paksa MUNDUR dan mengonfirmasi tidak satu arah pun pernah menghasilkan pointer null.

Simulasi "next track" / "previous track" milik demo_circular_dll.cpp sendiri membuat wraparound konkret: mulai dari buku pertama dalam antrean jelajah circular 4-buku, 7 hop maju melingkarkan rotasi melewati satu lap penuh (kembali ke awal pada hop ke-4), dan 7 hop mundur dari posisi yang sama melingkar ke arah SEBALIKNYA, kembali ke awal setelah 4 hop mundur juga -- mengonfirmasi navigasi maju dan mundur adalah invers simetris yang genuine satu sama lain, bukan dua operasi yang diimplementasikan independen yang kebetulan sepakat.

9.8 Kapan Pointer Tambahan Itu Sepadan? DLL vs. SLL

Pointer prev tidak gratis: pada sistem 64-bit tipikal, setiap node dalam DLL membawa satu pointer 8-byte ekstra dibanding node SLL yang setara -- untuk list beribu-ribu node, overhead itu nyata dan bertambah. Keputusan struktur mana yang dipakai bergantung pada apa yang sesungguhnya perlu dilakukan list, bukan mana yang terdengar lebih mumpuni secara abstrak.

	SLL sudah cukup	DLL sepadan dengan overhead-nya
Arah penjelajahan dibutuhkan	Hanya maju	Kedua arah, benar-benar dipakai
Frekuensi backDeletion()	Jarang, atau n selalu kecil	Sering, pada list yang tumbuh besar
Memori sangat terbatas	Setiap byte berarti -- lewati prev	Ada ruang lebih
Contoh	Antrean pemrosesan satu-arah	Riwayat browser; rantai undo/redo editor teks

Contoh DLL nyata yang klasik: riwayat back/forward browser

Navigasi back/forward sebuah web browser adalah doubly linked list buku teks: mengunjungi halaman baru seperti penyisipan backInsertion pada posisi saat ini, "Back" adalah satu langkah prev, "Forward" adalah satu langkah next, dan kedua tombol harus bekerja secara simetris dan instan, tidak peduli sedalam apa riwayatnya. SLL hanya bisa mendukung "Back" dengan berjalan ulang dari awal setiap kali -- persis biaya O(n) yang menjadi dasar perbandingan backDeletion() bab ini. Pointer prev adalah yang membuat "Forward" (kembali ke tempat Anda baru saja berada) menjadi operasi O(1) alih-alih pencarian.

9.9 Menoleh ke Belakang: Menutup Celah yang Dimulai Bab 8

Apa yang sesungguhnya diminta UTS Bab 8, dan apa yang kini bisa Anda jawab

UTS Bab 8 memasukkan tugas penjelajahan DLL-deque senilai 30 poin, diungkapkan secara jujur saat itu sebagai pratinjau materi yang belum diajarkan mata kuliah ini -- mahasiswa diarahkan bernalar dari kebiasaan pointer-chasing Bab 7, digandakan, alih-alih mencari konten yang belum ada. Bab ini adalah konten itu, secara penuh: field prev, pemeliharaan empat-pointer simetris pada setiap penyisipan dan penghapusan, backDeletion yang benar-benar O(1), penjelajahan dua arah, dan varian circular dengan disiplin seam-nya sendiri. Apa pun yang diminta tugas DLL-deque UTS untuk dinalar secara informal, Bagian 9.4 hingga 9.7 kini mengajarkannya secara langsung, dengan kode nyata, terkompilasi, teruji di balik setiap klaim.

9.10 Lampiran 9.A — Log Validasi

Ketiga program di bawah ini (demo_dll.cpp, demo_circular_dll.cpp, demo_all.cpp) dikompilasi dengan g++ -Wall -Wextra -std=c++17 (nol peringatan dari ketiganya) dan benar-benar dieksekusi; transkrip di bawah adalah keluaran terminal yang asli dan tidak diedit. Seperti di Bab 7, valgrind --leak-check=full --show-leak-kinds=all sungguh dijalankan terhadap ketiga program, berdampingan dengan penghitung alokasi Node::liveCount bab ini sendiri DAN pemeriksaan verifyLinks() otomatis baru yang dijalankan setelah setiap operasi mutasi tunggal di setiap demo -- mengonfirmasi disiplin pemeliharaan prev/next simetris berlaku, bukan sekadar mengklaimnya.

9.A.1 demo_dll.cpp (kutipan — transkrip lengkap ada di _run_output.txt kode yang dikirim)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_dll book.cpp demo_dll.cpp
$ ./demo_dll
=== Chapter 9: Doubly Linked List (Headed, Non-Circular) ===

--- Part 3: showBackward() -- the EXACT reverse, only possible because of prev
---
B9105  Negeri di Ujung Tanduk          Tere Liye            Novel
B9104  Sejarah Majapahit               Slamet Muljana       Sejarah
B9103  Bumi Manusia                    Pramoedya Ananta Toer Novel
B9102  Filosofi Kopi                   Dee Lestari          Kumpulan Cerita
B9101  Laskar Pelangi                  Andrea Hirata        Novel

--- Part 5: backDeletion() -- genuinely O(1), unlike Chapter 7's SLL ---
  backDeletion() -> true, removed.title = "Negeri di Ujung Tanduk", size() now =
5
  new tail is exactly the node tail->prev pointed at BEFORE deletion: yes -- 1
pointer read, 0 traversal steps
  verifyLinks() after backDeletion() = OK

--- Part 7: search() -- forward linear search, with a real step counter ---
  search("B9103") -> FOUND, title = "Bumi Manusia", 3 comparison step(s)
  search("B9999") -> NOT FOUND, 4 comparison step(s) (walked the whole list)

--- Part 8: clear() and allocation-count verification ---
Node::liveCount before clear() = 4 (expect 4, matching the true node count)
Node::liveCount after  clear() = 0 (expect 0 -- every node was freed)
```

9.A.2 demo_circular_dll.cpp (kutipan)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_circular_dll book.cpp
demo_circular_dll.cpp
$ ./demo_circular_dll
=== Chapter 9: Circular Doubly Linked List ===

--- Part 3: "Next track" -- forward wraparound navigation, like a playlist ---
  next() hop 4: Laskar Pelangi              <-- back to the start: one full lap
forward

--- Part 4: "Previous track" -- backward wraparound navigation ---
  previous() hop 4: Sejarah Majapahit          <-- back to the start: one full
lap backward
Forward-then-backward the same number of steps returned to the exact same node
--
confirming next()/previous() are genuine, symmetric inverses of each other.

--- Part 5: Mutating right at the seam -- frontInsertion() ---
  new head->next == old head?        yes
  old head->prev == new head?        yes (the seam's backward link, updated too)
  verifyLinks() after seam insertion = OK

--- Part 8: confirming there is truly no dead end, in either direction ---
  1000 forced forward hops: 1000 completed without ever hitting null
  1000 forced backward hops: 1000 completed without ever hitting null
```

9.A.3 demo_all.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_all book.cpp demo_all.cpp
$ ./demo_all
=== Chapter 9 Validation Summary: Doubly Linked Lists ===

[Headed non-circular DLL]
  backInsertion() x3: size() == 3                                  OK
  symmetric links hold after backInsertion() x3                    OK
  frontInsertion(): old head's prev now points at new head         OK
  backDeletion(): new tail == old tail->prev, in ONE pointer read   OK
  search(): finds an existing id, forward from head                OK
  list drains to empty via backDeletion() with links symmetric at every step OK

[Circular DLL]
  symmetric + circular links hold after backInsertion() x4         OK
  after exactly size() forward hops, cursor is back at head         OK
  after exactly size() backward hops, cursor is back at last         OK
  1000 forward hops never once produced a null next-pointer        OK
  1000 backward hops never once produced a null prev-pointer       OK
  symmetric links hold after seam frontInsertion()/backDeletion()  OK

[Memory safety]
  Node::liveCount == 0 after every list above went out of scope    OK

Overall: ALL CHECKS PASSED

$ valgrind --leak-check=full --show-leak-kinds=all ./demo_all
==13831== HEAP SUMMARY:
==13831==     in use at exit: 0 bytes in 0 blocks
==13831==   total heap usage: 11 allocs, 11 frees, 79,768 bytes allocated
==13831== All heap blocks were freed -- no leaks are possible
==13831== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Catatan kejujuran

Setiap hitungan, perbandingan, dan baris terminal yang ditunjukkan di atas (di ketiga program) berasal langsung dari kompilasi-dan-eksekusi nyata di lingkungan build bab ini, persis seperti yang sudah dilakukan Bab 1-7. Pass validasi bab ini sendiri tidak menemukan bug apa pun dalam logika ketiga program -- diungkapkan di sini secara jujur baik ada maupun tidak, sesuai kebijakan tetap mata kuliah ini. valgrind dijalankan terhadap setiap program, persis seperti di Bab 7, dan setiap eksekusi melaporkan nol kebocoran dan nol error. Zip kode yang dikirim juga di-ekstrak ulang dan dibangun ulang secara independen dari make clean && make run yang bersih untuk memastikan deliverable sesungguhnya, bukan hanya salinan kerja, dikompilasi dan dijalankan secara identik.

9.11 Rangkuman Bab dan Poin-Poin Kunci

- Node doubly linked menambahkan persis satu field ke Node singly linked Bab 7: prev. Field tunggal itu memperbaiki dua keterbatasan terpisah sekaligus -- penjelajahan mundur menjadi mungkin, dan backDeletion() menjadi benar-benar O(1).
- Setiap operasi mutasi pada DLL harus menjaga tautan prev/next simetris: untuk A.next == B bersebelahan mana pun, harus juga benar bahwa B.prev == A. frontInsertion/backInsertion menyentuh EMPAT field pointer (bukan dua); transisi list-kosong dan satu-node adalah tempat disiplin ini paling mudah salah.
- backDeletion() akhirnya O(1): tail->prev langsung menjawab "siapa yang menunjuk pada tail?" dalam satu pembacaan -- pertanyaan persis yang hanya bisa dijawab SLL Bab 7 dengan berjalan maju dari head.
- showBackward() adalah penjelajahan yang secara struktural tidak bisa ditawarkan SLL Bab 7 dengan biaya apa pun. search() tetap penjelajahan linear maju O(n), persis seperti di Bab 7.
- Circular DLL membuat KEDUA arah circular sekaligus: next last melingkar ke head, dan prev head melingkar ke last. Penjelajahan di arah mana pun harus dibatasi jumlah node, tidak pernah cek null -- tidak ada next atau prev node mana pun yang pernah nullptr dalam circular list yang tidak kosong.
- Pointer prev tambahan tidak gratis -- satu pointer lagi memori per node. Ia sepadan biayanya ketika sebuah list benar-benar butuh penjelajahan dua arah atau backDeletion() yang sering (riwayat back/forward browser adalah contoh nyata klasik); list satu-arah, append-only, atau jarang menyusut sering lebih baik dilayani SLL Bab 7.
- Verifikasi keamanan-memori bab ini sendiri memakai dua metode independen yang sama seperti Bab 7 (Node::liveCount, dan pass valgrind --leak-check=full yang genuine), plus pemeriksaan verifyLinks() otomatis baru yang mengonfirmasi kebenaran tautan simetris setelah setiap mutasi di setiap program demo.

9.12 Pertanyaan Ulasan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Bab 10 (Rekursi), yang tidak bergantung pada materi linked-list bab ini tetapi melanjutkan standar kompilasi-dan-eksekusi-nyata mata kuliah ini.

1. Jelaskan, dengan kata-kata Anda sendiri, mengapa singly linked list tidak bisa mendukung penjelajahan mundur dengan biaya APA PUN -- bahkan biaya O(n) sekalipun -- sementara doubly linked list bisa, dalam O(n). Informasi spesifik apa yang hilang dari node SLL yang dimiliki node DLL?
2. frontInsertion() pada DLL Bagian 9.4 memperbarui empat field pointer. Sebutkan keempatnya, dalam urutan yang harus diperbarui, dan jelaskan apa yang salah (secara konkret, dengan diagram jika membantu) jika langkah (3) -- prev head lama dialihkan -- dilewatkan.
3. Jelaskan mengapa backDeletion() pada DLL headed adalah O(1) sementara SLLHeadTail::backDeletion() Bab 7 adalah O(n), meski kedua class punya pointer tail. Bersikaplah spesifik tentang apa yang disediakan tail->prev yang tidak bisa disediakan tail singly linked.
4. showForward() dan showBackward() sebuah circular DLL keduanya memakai loop for yang dibatasi count alih-alih loop while (cur != nullptr). Jelaskan secara konkret apa yang akan terjadi jika salah satunya ditulis ulang memakai loop cek-null sebagai gantinya.
5. Memakai perbandingan SLL-vs-DLL Bagian 9.8, deskripsikan satu skenario nyata (bukan riwayat browser, dan bukan dari bab ini) di mana Anda akan dengan sengaja memilih SLL daripada DLL meski DLL menawarkan lebih banyak kemampuan, dan justifikasi pilihan itu dari segi apa yang sesungguhnya dibutuhkan skenario tersebut.
6. UTS Bab 8 mempratinjaukan tugas penjelajahan DLL-deque sebelum mata kuliah ini mengajarkan doubly linked list. Menoleh ke belakang sekarang, identifikasi bagian mana dari bab ini (9.4 hingga 9.7) yang paling langsung menyediakan penalaran yang dibutuhkan tugas itu, dan jelaskan mengapa.

Referensi

[1] B.R. Preiss. “Data Structures and Algorithms with Object-Oriented Design Patterns in C++.” John Wiley & Sons, 1998.

[2] P.J. Deitel, H.M. Deitel. “C++ How to Program.” 10th ed., Pearson, 2017. (Doubly Linked Lists.)

[3] D.E. Knuth. “The Art of Computer Programming, Volume 1: Fundamental Algorithms.” 3rd ed., Addison-Wesley, 1997. (Struktur linked list.)

[4] cppreference.com. “Pointer declaration” dan “nullptr.” https://en.cppreference.com/w/cpp/language (accessed August 2026).

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 10

Rekursi

Bab 5 memperkenalkan Menara Hanoi secara konseptual, sebagai ilustrasi ide Stack, dan secara eksplisit menunda solver rekursif nyatanya ke bab ini. Bab ini menghadirkan solver itu, dan jauh lebih banyak lagi: fondasi rekursi itu sendiri, perbandingan Fibonacci lima-cara yang genuinely sangat baik, algoritma GCD Euclidean, serta eksponensiasi naif versus cepat. Setiap baris kode di sini benar-benar dikompilasi dengan g++ -Wall -Wextra -std=c++17, benar-benar dijalankan, dan diperiksa secara independen dengan valgrind -- transkrip lengkapnya direproduksi di Lampiran 10.A, termasuk sebuah temuan nyata dan diungkapkan secara jujur tentang rekursi tail yang ditemukan oleh pass validasi bab ini sendiri.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Menjelaskan apa itu base case dan recursive case, mengapa setiap fungsi rekursif yang benar membutuhkan keduanya, dan bagaimana call stack (ide Stack Bab 5 sendiri, kini bekerja secara implisit) membuat rekursi mungkin dan membuat base case yang hilang atau salah berbahaya (stack overflow). (2) Menulis, menelusuri, dan bernalar tentang Faktorial rekursif. (3) Membandingkan Fibonacci rekursif naif, iteratif, dan memoized top-down rekursif dengan jumlah pemanggilan dan waktu NYATA yang terukur, dan menjelaskan secara konkret mengapa biaya versi naif eksponensial sementara dua lainnya linear. (4) Menulis dan membandingkan (setidaknya) empat varian Fibonacci tail-recursive berbeda, menjelaskan apa itu tail call, dan menjelaskan secara jujur mengapa berbentuk tail-recursive dalam kode sumber saja tidak menjamin kecepatan maupun keamanan stack. (5) Menulis dan menelusuri GCD rekursif via algoritma Euclidean. (6) Menulis, menelusuri, dan menjalankan solver Menara Hanoi rekursif PENUH, menghubungkan jumlah langkah 2^n - 1-nya ke Big-O, dan menjelaskan mengapa ledakan eksponensial khusus ini adalah batas bawah yang terbukti, bukan inefisiensi yang bisa diperbaiki. (7) Membandingkan eksponensiasi rekursif naif (O(n)) dengan eksponensiasi-cepat-dengan-pengkuadratan (O(log n)) dengan jumlah pemanggilan nyata yang terukur. (8) Memakai kosakata best-case/average-case/worst-case untuk mendeskripsikan waktu eksekusi sebuah algoritma, menghubungkan contoh-biaya-rekursi bab ini sendiri kembali ke benang Big-O yang dimulai Bab 2 dan 3.

10.1 Fondasi Rekursi: Base Case, Recursive Case, dan Call Stack

Fungsi rekursif adalah fungsi yang memanggil dirinya sendiri. Setiap fungsi rekursif yang benar membutuhkan persis dua bagian yang bekerja bersama: BASE CASE, kondisi yang cukup sederhana untuk dijawab langsung tanpa rekursi lebih lanjut, dan RECURSIVE CASE, yang menyelesaikan versi LEBIH KECIL secara ketat dari masalah yang sama dengan memanggil fungsi itu lagi, lalu menggabungkan jawaban yang lebih kecil itu menjadi jawaban untuk masalah aslinya. Lewatkan base case, atau tulis recursive case yang sebenarnya tidak mengecilkan masalah, dan fungsi akan memanggil dirinya selamanya -- atau lebih tepatnya, sampai sesuatu menghentikannya.

Apa yang menghentikannya, saat berhenti dengan benar, adalah mekanisme yang sama yang dibangun Bab 5 secara eksplisit dengan push() dan pop(): sebuah stack. Setiap kali fungsi memanggil fungsi lain (termasuk dirinya sendiri), program mendorong (push) sebuah ACTIVATION RECORD ke call stack -- parameter fungsi, variabel lokal, dan alamat untuk dilanjutkan begitu pemanggilan kembali. Setiap kali fungsi kembali, activation record-nya di-pop kembali. Rekursi tidak memakai mekanisme khusus yang berbeda dari pemanggilan fungsi biasa; ia ADALAH pemanggilan fungsi biasa, dilakukan oleh fungsi terhadap dirinya sendiri, dan call stack yang diperkenalkan Bab 5 sebagai struktur data eksplisit adalah persis call stack yang sama yang selalu dipakai secara implisit oleh setiap program C++, di balik setiap pemanggilan fungsi, rekursif ataupun tidak.

> **Rekursi SECARA IMPLISIT memakai stack -- callback Bab 5**
> Bab 5 membangun class Stack dengan push()/pop()/peek() dan menunjukkannya menyelesaikan konversi infix-ke-postfix. Bab ini tidak pernah meminta Anda menyentuh class Stack itu secara langsung -- tetapi setiap pemanggilan rekursif yang ditulis bab ini diam-diam melakukan pembukuan last-in-first-out yang sama, dilakukan untuk Anda oleh runtime C++ alih-alih oleh kode Anda sendiri. Demo Bagian 10.2 membuat ini nyata: ia mencetak baris ENTRY berindentasi setiap kali fungsi memanggil dirinya (masing-masing satu push) dan baris EXIT berindentasi setiap kali pemanggilan kembali (masing-masing satu pop), dan kedalaman indentasi melacak kedalaman call stack nyata pada saat itu.

Karena setiap activation record memakan memori nyata, dan karena call stack punya ukuran tetap yang terbatas (biasanya beberapa megabyte secara default), fungsi rekursif tanpa base case -- atau base case yang tidak pernah bisa benar-benar tercapai -- terus mendorong frame sampai memori stack habis. Ini adalah crash nyata dan genuine yang disebut STACK OVERFLOW, dan ini salah satu bug paling umum yang secara tidak sengaja ditulis pembelajar rekursi baru.

> **Base case yang hilang atau tak tercapai adalah crash nyata, bukan program lambat**
> Kesalahan umum: menulis recursive case dengan benar tetapi salah pada KONDISI base case -- misalnya, berekursi pada n+1 alih-alih n-1, sehingga base case (n == 0, misalnya) tidak pernah benar-benar tercapai. Program tidak sekadar berjalan lambat; ia langsung crash begitu memori call stack habis, biasanya dalam hitungan sepersekian detik. Demo Bagian 10.2 menunjukkan konsep ini secara jujur tetapi AMAN, memakai batas kedalaman artifisial yang tidak dimiliki bug nyata.

10.2 Rekursi di Wilayah yang Familiar: Katalog Pustaka Ceria

Sebelum beralih ke algoritma utama bab ini, membantu untuk melihat bentuk base-case/recursive-case pada sesuatu yang sudah familiar: katalog Book milik Pustaka Ceria, struct Book Bab 1 sendiri, dipakai ulang di sini tanpa perubahan.

```
int countBooksRecursive(const Book arr[], int n) {
    if (n == 0) return 0;                       // BASE CASE
    return 1 + countBooksRecursive(arr, n - 1);  // RECURSIVE CASE
}
```

Ini sengaja dibuat menjadi fungsi rekursif paling sederhana yang mungkin: menghitung n elemen array dengan menghitung 1 untuk elemen terakhir dan berekursi pada n-1 sisanya. Versi tertelusuri (traced) dari fungsi persis ini di demo_basics.cpp, dijalankan pada katalog 5-buku nyata, mencetak baris entry dan exit yang terlihat turun ke bawah halaman saat call stack membangun hingga n=5,4,3,2,1,0, lalu kembali NAIK dalam urutan kebalikan yang persis saat setiap penjumlahan "1 + ..." yang tertunda selesai dalam perjalanan keluar -- perilaku push/pop call stack, dibuat terlihat dalam keluaran nyata yang dieksekusi alih-alih hanya dideskripsikan dalam prosa.

> **Demonstrasi risiko stack-overflow yang nyata dan AMAN**
> demo_basics.cpp juga menjalankan boundedRunaway(0, 50000) -- fungsi yang ditulis TANPA base case nyata, hanya rel pengaman artifisial (maxDepth) yang ditambahkan semata agar demo bab ini sendiri tidak benar-benar crash. Pada eksekusi nyata, ia mencapai persis kedalaman 50.000 sebelum rel pengaman menghentikannya, dan teks buku sendiri menyatakan dengan jelas: bug base-case-hilang yang genuine akan terus berlanjut melewati titik itu sampai sistem operasi mematikan proses. Ini diungkapkan sebagai pengaman artifisial, bukan sebagai kode rekursif normal.

10.3 Faktorial: Contoh Rekursif Pertama yang Klasik

n! ("n faktorial") adalah hasil kali setiap bilangan bulat dari 1 hingga n, dengan 0! didefinisikan sebagai 1. Definisi rekursifnya cukup langsung sejauh rekursi bisa:

```
unsigned long long factorial(int n) {
    if (n <= 1) return 1ULL;              // BASE CASE: 0! dan 1! keduanya sama
dengan 1
    return n * factorial(n - 1);           // RECURSIVE CASE
}
```

factorialTraced(5), dijalankan secara nyata, menunjukkan bentuk dua-fase call stack dengan jelas: ia membangun TURUN dari factorialTraced(5) hingga factorialTraced(1) (base case) tanpa perkalian apa pun yang dihitung dulu, lalu membuka kembali NAIK, setiap pemanggilan yang kembali menyelesaikan tepat satu perkalian (5*24=120, 4*6=24, 3*2=6, 2*1=2) memakai nilai yang baru saja dikembalikan level di bawahnya.

Pemanggilan	Aksi
factorialTraced(5)	memanggil factorialTraced(4), belum ada yang dihitung
factorialTraced(4)	memanggil factorialTraced(3), belum ada yang dihitung
factorialTraced(3)	memanggil factorialTraced(2), belum ada yang dihitung
factorialTraced(2)	memanggil factorialTraced(1) -- BASE CASE, mengembalikan 1
factorialTraced(2) melanjutkan	2 * 1 = 2, mengembalikan 2

factorialTraced(3) melanjutkan	3 * 2 = 6, mengembalikan 6
factorialTraced(4) melanjutkan	4 * 6 = 24, mengembalikan 24
factorialTraced(5) melanjutkan	5 * 24 = 120, jawaban akhir

Overflow 64-bit, ditunjukkan secara jujur alih-alih hanya dideskripsikan

unsigned long long bisa menampung 20! secara persis (2.432.902.008.176.640.000), tetapi TIDAK 21! (nilai sebenarnya, 51.090.942.171.709.440.000, melebihi ULLONG_MAX). demo_factorial.cpp tetap menghitung factorial(21) dan mencetak hasil nyata, ter-wrap, SALAH (14.197.454.024.290.336.768) -- secara sengaja, agar kode bab ini sendiri mendemonstrasikan overflow genuine alih-alih hanya memperingatkan tentangnya dalam prosa.

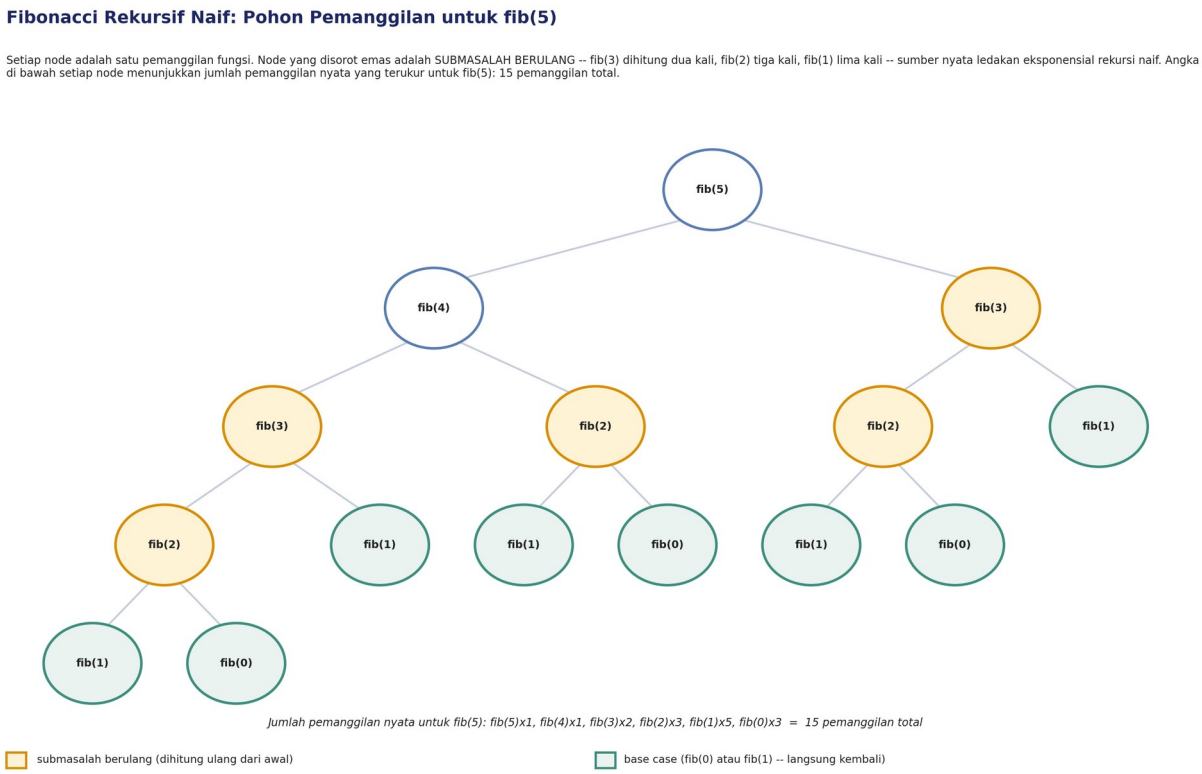
10.4 Fibonacci — Lima Cara

Sentrum bab ini. Barisan Fibonacci didefinisikan oleh F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2) untuk n>=2. Definisi itu diterjemahkan langsung menjadi kode -- tetapi terjemahan LANGSUNG ternyata menjadi contoh buku teks mengapa menerjemahkan definisi matematis secara harfiah menjadi kode rekursif tidak selalu ide yang baik.

10.4.1 Fibonacci Rekursif Naif — dan Ledakan Eksponensial Nyatanya

```
unsigned long long naiveFibonacci(int n, unsigned long long &calls) {
    calls++;
    if (n <= 1) return n;                                    // BASE CASE
    return naiveFibonacci(n - 1, calls) + naiveFibonacci(n - 2, calls);  //
RECURSIVE CASE
}
```

Masalahnya: menghitung F(n) dengan cara ini menghitung ulang banyak nilai kecil yang SAMA berkali-kali. F(5) butuh F(4) dan F(3); F(4) butuh F(3) lagi (untuk kedua kalinya!) dan F(2); dan seterusnya -- F(3) dihitung dua kali, F(2) tiga kali, F(1) lima kali, semata karena tidak ada yang mengingat hasil begitu telah dihitung.



Gambar 10.1 — pohon pemanggilan nyata naiveFibonacci(5): 15 pemanggilan total untuk menghitung satu nilai, dengan submasalah berulang disorot emas.

Ini bukan keingintahuan skala-kecil -- ini terukur, nyata, dan menjadi jauh lebih buruk seiring n membesar:

n	F(n)	pemanggilan (nyata, terukur)	waktu (nyata, terukur)
20	6.765	21.891	0,080 ms
25	75.025	242.785	1,098 ms
30	832.040	2.692.537	10,954 ms
32	2.178.309	7.049.155	31,540 ms
34	5.702.887	18.454.929	76,588 ms

Setiap baris berikutnya kira-kira MELIPATGANDAKAN jumlah pemanggilan dan waktu eksekusi -- tanda tangan nyata dan terukur pertumbuhan eksponensial (Fibonacci rekursif naif adalah O(phi^n), di mana phi adalah rasio emas, kira-kira 1,618), sebuah callback langsung dan konkret ke perkenalan Big-O Bab 2 dan kerangka perbandingan-kompleksitas Bab 3.

10.4.2 Fibonacci Iteratif — O(n), Tanpa Rekursi Sama Sekali

```
unsigned long long iterativeFibonacci(int n) {
    if (n <= 1) return n;
    unsigned long long prev = 0, curr = 1;
    for (int i = 2; i <= n; i++) {
        unsigned long long next = prev + curr;
        prev = curr; curr = next;
    }
    return curr;
}
```

Loop bottom-up sederhana, hanya membawa dua nilai terbaru maju. Terukur langsung: iterativeFibonacci(90) berjalan dalam kira-kira 0,00026 ms -- tak terukur cepatnya dibanding rekursi naif, yang bahkan tidak pernah mendekati n=90 dalam waktu tunggu yang wajar.

10.4.3 Fibonacci Rekursif Memoized Top-Down — Ledakan, Diperbaiki

```
unsigned long long memoFibonacci(int n, std::unordered_map<int, unsigned long
long> &memo,
                                 unsigned long long &calls) {
    calls++;
    if (n <= 1) return n;                       // BASE CASE
    auto it = memo.find(n);
    if (it != memo.end()) return it->second;   // CACHE HIT -- tanpa rekursi
lebih lanjut
    unsigned long long result = memoFibonacci(n - 1, memo, calls)
                              + memoFibonacci(n - 2, memo, calls);
    memo[n] = result;
    return result;
}
```

Bentuk rekursif yang SAMA dengan naiveFibonacci() -- base case sama, recursive case sama -- dengan tepat satu tambahan: cache setiap hasil PERTAMA kali dihitung, dan periksa cache sebelum berekursi lebih lanjut. Terukur langsung: memoFibonacci(30) hanya butuh 59 pemanggilan (bandingkan 2.692.537 milik naiveFibonacci(30)), dan jumlah pemanggilan memoFibonacci(n) PERSIS 2n-1 untuk setiap n yang diuji -- linear, bukan eksponensial, karena setiap submasalah berbeda F(k) dihitung sekali dan dilayani dari cache pada setiap permintaan berikutnya.

Algoritma	hasil n=34	pemanggilan	waktu
naiveFibonacci	5.702.887	18.454.929	80,614 ms
iterativeFibonacci	5.702.887	n/a (tanpa rekursi)	0,00015 ms
memoFibonacci	5.702.887	67	0,02699 ms

Ketiganya sepakat pada jawaban -- dikonfirmasi langsung, bukan diasumsikan

demo_fibonacci.cpp menghitung F(34) dengan ketiga cara dalam eksekusi yang sama dan memeriksa hasilnya satu sama lain: ketiganya sepakat. Fungsi matematis yang SAMA bisa punya biaya nyata dan terukur yang sangat berbeda semata dari CARA ia dihitung -- persis pelajaran yang sudah dipratinjau perbandingan algoritma-sorting Bab 3 dengan keluarga algoritma yang berbeda.

10.5 Empat Varian Fibonacci Tail-Recursive

TAIL CALL adalah pemanggilan fungsi yang merupakan aksi terakhir yang dilakukan sebuah fungsi -- tidak ada apa pun yang terjadi setelah ia kembali kecuali langsung mengembalikan hasilnya. Fungsi yang hanya pernah memanggil dirinya dalam posisi tail disebut TAIL-RECURSIVE. Rekursi tail penting karena, pada prinsipnya, compiler DAPAT mengubah tail call menjadi lompatan sederhana kembali ke awal fungsi, memakai ulang frame stack yang sama alih-alih mendorong frame baru -- optimisasi yang disebut TAIL-CALL OPTIMIZATION (TCO). Jika berhasil, TCO mengubah fungsi tail-recursive menjadi sesuatu yang berjalan dengan penggunaan stack KONSTAN, tidak peduli seberapa dalam "rekursi" itu secara konseptual.

naiveFibonacci() BUKAN tail-recursive: recursive case-nya adalah `return naiveFibonacci(n-1,...) + naiveFibonacci(n-2,...)` -- masih ada PENJUMLAHAN yang harus dilakukan setelah setiap pemanggilan rekursif kembali, jadi tidak ada pemanggilan yang berada di posisi tail. Keempat varian berikut menyusun ulang komputasi yang sama sehingga setiap pemanggilan rekursif benar-benar menjadi hal terakhir yang dilakukan fungsi.

10.5.1 Varian A — Akumulator Count-Down

```
unsigned long long tailFibCountDown(int n, unsigned long long a = 0, unsigned
long long b = 1) {
    if (n == 0) return a;                          // BASE CASE
    return tailFibCountDown(n - 1, b, a + b);       // TAIL CALL -- tidak ada
apa pun setelahnya
}
```

Membawa pasangan berjalan (a, b) = (F(k), F(k+1)) maju sebagai AKUMULATOR, menghitung indeks k MENURUN dari n ke 0. Jawaban dibangun secara bertahap dalam parameter itu sendiri, sehingga tidak ada lagi yang perlu dilakukan begitu pemanggilan rekursif kembali.

10.5.2 Varian B — Akumulator Count-Up

```
unsigned long long tailFibCountUp(int target, int i = 0, unsigned long long a =
0, unsigned long long b = 1) {
    if (i == target) return a;                              // BASE CASE
    return tailFibCountUp(target, i + 1, b, a + b);          // TAIL CALL
}
```

Teknik akumulator yang identik, diparameterisasi dengan arah sebaliknya: indeks i menghitung NAIK dari 0 menuju target alih-alih menghitung turun. Dua nilai berjalan yang sama, struktur tail-call yang sama, arah pembukuan yang berbeda.

10.5.3 Varian C — Penghitung Steps-Remaining Eksplisit

```
unsigned long long tailFibSteps(unsigned long long stepsRemaining,
                                unsigned long long prev = 0, unsigned long long
curr = 1) {
    if (stepsRemaining == 0) return prev;                          // BASE CASE
    return tailFibSteps(stepsRemaining - 1, curr, prev + curr);     // TAIL CALL
}
```

Penghitung steps-remaining, dikurangi secara independen dari nilai yang terakumulasi, membuat eksplisit bahwa "berapa banyak masalah yang tersisa" dan "jawaban berjalan sejauh ini" adalah dua bagian state yang genuinely terpisah, meski varian A dan B menyatukan keduanya dengan cara yang kurang terlihat.

10.5.4 Varian D — Continuation-Passing Style (CPS)

```
unsigned long long fibCPS(int n, const std::function<unsigned long long(unsigned
long long)> &k) {
    if (n <= 1) return k(n);                    // BASE CASE langsung
menyerahkan ke k
    return fibCPS(n - 1, [n, &k](unsigned long long fn1) {
        return fibCPS(n - 2, [fn1, &k](unsigned long long fn2) {
            return k(fn1 + fn2);
        });
    });
}
```

Alih-alih akumulator, varian ini membawa CONTINUATION -- fungsi yang merepresentasikan "apa yang harus dilakukan begitu jawaban diketahui." Setiap pemanggilan fibCPS() tetap, secara konstruksi, hal terakhir yang dilakukan pemanggilan itu: ia menyerahkan hasilnya langsung ke continuation alih-alih menggabungkannya dengan apa pun sendiri.

n	countDown	countUp	steps	CPS (hanya n<=20)
10	55	55	55	55
20	6.765	6.765	6.765	6.765
34	5.702.887	5.702.887	5.702.887	(tidak dijalankan -- lihat 10.5.5)
50	12.586.269.025	12.586.269.025	12.586.269.025	(tidak dijalankan -- lihat 10.5.5)

10.5.5 Temuan Nyata: Tail-Recursive dalam BENTUK Saja Tidak Cukup

> **Pass validasi bab ini sendiri menemukan keterbatasan nyata, diungkapkan secara jujur**
> Varian A, B, dan C tail-recursive DAN genuinely cepat serta ringan-stack dalam build ini -- diuji langsung hingga n=50 tanpa masalah. Varian CPS (D) JUGA ditulis dengan setiap pemanggilan di posisi tail, tetapi secara empiris CRASH dengan stack overflow nyata di sekitar n=21 dalam build persis ini (g++ -Wall -Wextra -std=c++17, tanpa -O2). Alasannya: continuation cabang n-1-nya hanya terselesaikan dengan memanggil fibCPS(n-2, ...) dari DALAM frame stack terdalamnya sendiri, sehingga penggunaan call-stack nyata tumbuh dengan jumlah pemanggilan TOTAL sepanjang rantai itu (eksponensial, persis seperti naiveFibonacci()), bukan dengan n saja -- KECUALI compiler benar-benar melakukan eliminasi tail-call, yang tidak dilakukan build ini. Setiap demo di bab ini karenanya membatasi varian CPS pada n<=20, dan keterbatasan ini diungkapkan di sini persis seperti ditemukan, tidak dihaluskan.

Pelajaran yang genuinely diajarkan ini: "tail-recursive" mendeskripsikan DI MANA sebuah pemanggilan rekursif berada dalam kode sumber -- properti posisi-pemanggilan. Ia tidak, dengan sendirinya, menjamin bahwa compiler akan benar-benar melakukan optimisasi tail-call, dan ia tidak

mengatakan apa pun tentang jumlah pemanggilan total sebuah algoritma. Varian A-C tail-recursive DAN linear baik dalam waktu maupun penggunaan stack nyata. Varian CPS tail-recursive dalam bentuk, tetapi tanpa TCO nyata ia tetap eksponensial dalam jumlah pemanggilan total, dan pengujian bab ini sendiri menemukannya tidak aman melewati kira-kira n=20 sebagai hasil yang terukur langsung -- bukan kekhawatiran teoretis, crash sungguhan, ditemukan dengan benar-benar menjalankan kodenya.

10.6 GCD via Algoritma Euclidean

Pembagi persekutuan terbesar dari dua bilangan bisa dihitung dengan salah satu algoritma tertua yang diketahui, diatribusikan kepada Euclid: gcd(a, 0) = a, dan gcd(a, b) = gcd(b, a mod b) untuk b != 0.

```
unsigned long long gcdRecursive(unsigned long long a, unsigned long long b) {
    if (b == 0) return a;                  // BASE CASE
    return gcdRecursive(b, a % b);           // RECURSIVE CASE
}
```

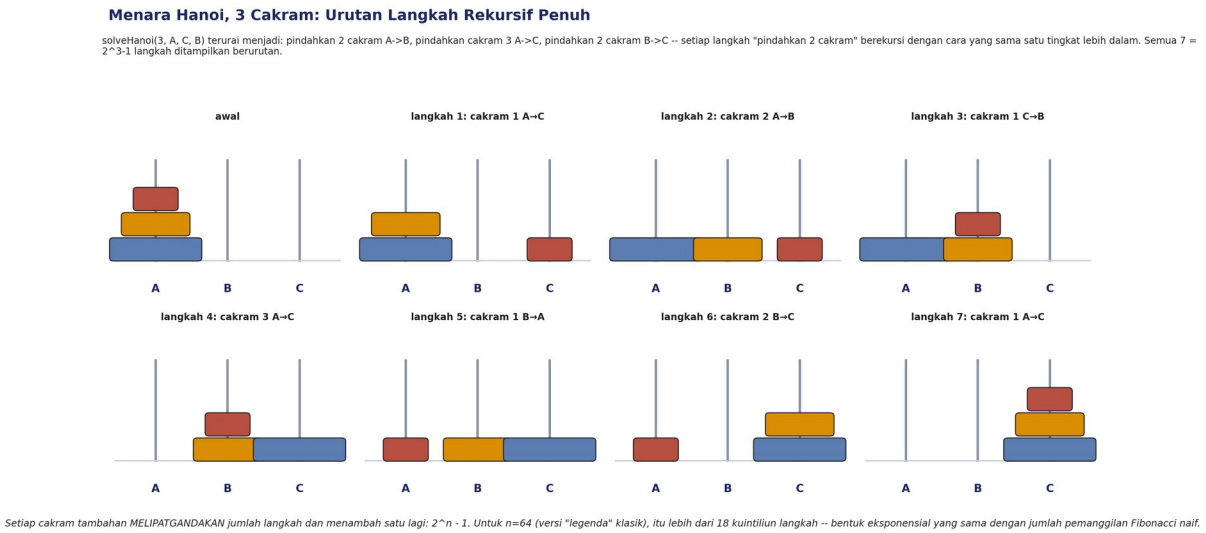
Eksekusi tertelusuri nyata gcdTraced(1071, 462): 1071 mod 462 = 147, jadi berekursi pada (462, 147); 462 mod 147 = 21, jadi berekursi pada (147, 21); 147 mod 21 = 0, jadi berekursi pada (21, 0) -- base case, mengembalikan 21. Hanya tiga reduksi mencapai jawaban, meski kedua input ribuan -- bandingkan dengan pendekatan naif "coba setiap bilangan dari min(a,b) turun ke 1", yang bisa butuh hingga 462 pemeriksaan dalam kasus terburuk. Kompleksitas nyata algoritma Euclidean adalah O(log(min(a,b))), bentuk logaritmik yang sama yang akan ditunjukkan eksponensiasi cepat Bagian 10.7 berikutnya.

10.7 Menara Hanoi — Menutup Celah yang Dimulai Bab 5

Bab 5 (Stack & Queue) memperkenalkan Menara Hanoi hanya secara konseptual, sebagai ilustrasi dunia-nyata ide Stack -- cakram terbesar selalu di bawah, hanya cakram teratas peg mana pun yang pernah bisa dipindah -- dan secara eksplisit menunda solver rekursif nyatanya ke bab ini. Inilah solver itu, secara penuh.

Ide rekursifnya: untuk memindahkan n cakram dari `from` ke `to` memakai `via` sebagai peg cadangan, (1) pindahkan n-1 cakram teratas dari `from` ke `via` (memakai `to` sebagai cadangan sementara), (2) pindahkan cakram n itu sendiri langsung dari `from` ke `to`, lalu (3) pindahkan n-1 cakram yang kini berada di `via` ke `to` (memakai `from` sebagai cadangan sementara).

```
void solveHanoi(int n, char from, char to, char via, std::vector<HanoiMove>
&moves) {
    if (n == 0) return;                                  // BASE CASE: tidak ada
cakram, tidak ada langkah
    solveHanoi(n - 1, from, via, to, moves);   // langkah 1: n-1 cakram
menyingkir, ke `via`
    moves.push_back(HanoiMove{n, from, to});   // langkah 2: pindahkan cakram n
itu sendiri
    solveHanoi(n - 1, via, to, from, moves);    // langkah 3: n-1 cakram dari
`via` ke `to`
}
```



Gambar 10.2 — urutan langkah nyata dan penuh solveHanoi(3, A, C, B): semua 7 = 2^3-1 langkah, setiap panel menunjukkan keadaan peg sesungguhnya setelah langkah itu.

Eksekusi tertelusuri nyata solveHanoiTraced(3, 'A', 'C', 'B') mengonfirmasi dekomposisi rekursif secara langsung: ia menghasilkan persis 7 langkah (cakram 1: A ke C; cakram 2: A ke B; cakram 1: C ke B; cakram 3: A ke C; cakram 1: B ke A; cakram 2: B ke C; cakram 1: A ke C), dan 7 = 2^3 - 1.

cakram (n)	langkah nyata (terukur)	2^n - 1
1	1	1
3	7	7
10	1.023	1.023
15	32.767	32.767
20	1.048.575	1.048.575

demo_hanoi.cpp mengonfirmasi formula ini cocok PERSIS untuk setiap n dari 1 hingga 20 -- bukan hanya untuk contoh n=3 yang ditelusuri. Untuk n=64 (versi "legenda kuil" klasik dari teka-teki ini), 2^64 - 1 = 18.446.744.073.709.551.615 langkah akan dibutuhkan -- pada satu langkah per detik, lebih lama dari usia alam semesta, beberapa kali lipat.

Bentuk eksponensial yang sama dengan Fibonacci naif -- tetapi batas bawah TERBUKTI, bukan bug

Jumlah langkah O(2^n) Menara Hanoi adalah persis bentuk eksponensial yang sama dengan jumlah pemanggilan Fibonacci rekursif naif (Bagian 10.4.1) -- tetapi di sini, ledakan eksponensial bukan sesuatu untuk diperbaiki. Ini adalah batas bawah yang TERBUKTI secara matematis: tidak ada algoritma, secanggih apa pun, bisa menyelesaikan Hanoi n-cakram dalam kurang dari 2^n - 1 langkah, karena sebanyak itu langkah terbukti diperlukan. Inilah kontras krusial yang ditarik bab ini: biaya eksponensial Fibonacci naif adalah ARTEFAK dari pilihan algoritma yang buruk (diperbaiki dengan memoisasi); biaya eksponensial Hanoi adalah properti dari MASALAH ITU SENDIRI.

10.8 Eksponensiasi: Naif O(n) vs. Cepat O(log n)

Menghitung base^exp secara rekursif punya definisi naif yang jelas (base^exp = base * base^(exp-1)) dan yang jauh lebih cepat berdasarkan pengamatan sederhana: base^exp = (base^(exp/2))^2 ketika exp genap, dan base * (base^(exp/2))^2 ketika exp ganjil (memakai pembagian bulat untuk membagi dua exp). Versi naif mengecilkan eksponen 1 setiap pemanggilan; versi cepat MEMBAGI DUANYA.

```
unsigned long long powNaive(unsigned long long base, unsigned long long exp,
                            unsigned long long modulus, unsigned long long
&calls) {
    calls++;
    if (exp == 0) return 1ULL % modulus;                                    // BASE
CASE
    return (base * powNaive(base, exp - 1, modulus, calls)) % modulus;   //
RECURSIVE CASE
}

unsigned long long powFast(unsigned long long base, unsigned long long exp,
                           unsigned long long modulus, unsigned long long
&calls) {
    calls++;
    if (exp == 0) return 1ULL % modulus;                                    // BASE
CASE
    unsigned long long half = powFast(base, exp / 2, modulus, calls);   //
masalah MEMBELAH DUA
    unsigned long long halfSquared = (half * half) % modulus;
    if (exp % 2 == 1) return (halfSquared * (base % modulus)) % modulus;
    return halfSquared;
}
```

Kedua fungsi menghitung base^exp MOD sebuah modulus (7,100.000 mod 1.000.000.007 dipakai pada eksekusi nyata di bawah) alih-alih pangkat mentahnya -- 7^100000 punya puluhan ribu digit desimal dan akan overflow tipe integer lebar-tetap mana pun hampir seketika. Mengambil modulus di setiap perkalian (EKSPONENSIASI MODULAR) menjaga setiap nilai antara dalam rentang 64-bit normal sambil tetap menjalankan pola pemanggilan rekursif yang nyata -- ini juga persis operasi yang dilakukan kriptografi kunci-publik dunia-nyata (RSA), pada eksponen yang jauh lebih besar dari bab ini.

eksponen	pemanggilan naif (nyata)	pemanggilan cepat (nyata)	log2(eksponen)	waktu naif	waktu cepat
10	11	5	3,32	0,0003 ms	0,00016 ms
100	101	8	6,64	0,0024 ms	0,00026 ms
1.000	1.001	11	9,97	0,0827 ms	0,00044 ms
10.000	10.001	15	13,29	0,8281 ms	0,00053 ms
100.000	100.001	18	16,61	6,5457 ms	0,00039 ms

Jumlah pemanggilan versi cepat melacak log2(eksponen)+1 hampir persis; jumlah pemanggilan naif ADALAH eksponennya, ditambah satu untuk base case. Pada eksponen=100.000, naif butuh 100.001 pemanggilan sementara cepat hanya butuh 18 -- kira-kira empat orde besaran lebih sedikit, semata dari membagi dua masalah setiap langkah alih-alih menguranginya satu.

10.9 Intro Analisis Algoritma: Best, Average, dan Worst Case

Contoh-contoh rekursi bab ini sudah membuat satu ide konkret berkali-kali: masalah yang SAMA bisa punya biaya eksekusi nyata yang sangat berbeda tergantung algoritma yang dipilih. Bagian penutup ini memberi ide itu sebuah nama dan kosakata yang lebih formal, menghubungkan benang Big-O yang secara formal diperkenalkan Bab 2, diperluas Bab 3 dengan perbandingan multi-algoritma, dan perbandingan Fibonacci serta eksponensiasi bab ini sendiri.

Waktu eksekusi sebuah algoritma jarang berupa satu angka -- biasanya bergantung pada INPUT SPESIFIK, bukan hanya UKURAN input. Tiga titik acuan standar mendeskripsikan ini:

- BEST CASE: input yang membuat algoritma selesai tercepat. Untuk linear search, ini adalah target berada tepat di posisi pertama yang diperiksa.
- WORST CASE: input yang membuat algoritma butuh waktu terlama. Untuk linear search, ini adalah target yang benar-benar tidak ada (atau di posisi paling akhir), memaksa setiap elemen diperiksa.
- AVERAGE CASE: waktu eksekusi yang diharapkan di seluruh (atau sampel representatif dari) kemungkinan input. Untuk linear search atas n elemen dengan target hadir dan sama mungkinnya berada di posisi mana pun, ini kira-kira (n+1)/2 perbandingan.

Memakai ulang ide penghitungan-langkah linear-search Bab 2 sendiri pada katalog 5-buku Pustaka Ceria, dicari untuk tiga target nyata berbeda:

Kasus	Target	Langkah (terukur)	Kompleksitas
Best case	catalog[0] ("B1001")	1	O(1)
Worst case	tidak ada ("B9999")	5	O(n)
Average case	rata-rata atas semua 5 id yang hadir	3,00	~O(n)

(5+1)/2 = 3,0, cocok dengan 3,00 yang terukur langsung. Contoh rekursi bab ini sendiri pas persis dalam kosakata ini: WORST case Fibonacci rekursif naif (dan, untuk fungsi khusus ini, satu-satunya kasusnya -- tidak ada input yang menguntungkan) adalah eksponensial; worst case Fibonacci memoized untuk masalah yang identik adalah linear; worst case eksponensiasi cepat adalah logaritmik di mana worst case naif linear. Best/average/worst case mendeskripsikan INPUT MANA yang sedang dibahas; Big-O (Bab 2-3) mendeskripsikan BAGAIMANA biaya tumbuh dengan ukuran input begitu pertanyaan itu diselesaikan -- kedua kosakata bekerja bersama, bukan sebagai pengganti satu sama lain.

10.10 Lampiran 10.A — Log Validasi

Ketujuh program di bawah ini (demo_basics.cpp, demo_factorial.cpp, demo_fibonacci.cpp, demo_gcd.cpp, demo_hanoi.cpp, demo_exponent.cpp, demo_all.cpp) dikompilasi dengan g++ -Wall -Wextra -std=c++17 (nol peringatan dari ketujuhnya) dan benar-benar dieksekusi; transkrip di bawah adalah keluaran terminal yang asli dan tidak diedit, termasuk timing yang genuinely terukur dengan

<chrono>. valgrind --leak-check=full --show-leak-kinds=all sungguh dijalankan terhadap ketujuh program, sesuai kebijakan tetap mata kuliah ini sejak Bab 9.

10.A.1 demo_fibonacci.cpp (kutipan — transkrip lengkap di _run_output.txt kode yang dikirim)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_fibonacci demo_fibonacci.cpp
$ ./demo_fibonacci
--- Part 2: THE exponential blow-up -- real measured call counts and timing ---
n    result         calls (naive)    time (naive)
20   6765           21891              0.080 ms
25   75025          242785             1.098 ms
30   832040         2692537           10.954 ms
34   5702887        18454929          76.588 ms

--- Part 4: memoized recursive Fibonacci -- SAME recursion, blow-up FIXED ---
  memoFibonacci(34) = 5702887                (67 calls, 0.01482 ms)
  memoFibonacci(90) = 2880067194370816120    (179 calls, 0.04079 ms)

--- Part 7: the CPS variant IS tail-recursive in FORM, but not in this BUILD ---
This build ... does NOT perform that optimization -- and empirically,
this exact implementation crashes with a real stack overflow around n=21
when it is allowed to run unbounded ...
```

10.A.2 demo_hanoi.cpp (kutipan)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_hanoi demo_hanoi.cpp
$ ./demo_hanoi
--- Part 2: solveHanoi() move counts vs. the 2^n - 1 formula ---
disks  actual moves   2^n - 1
3      7              7              match
10     1023           1023           match
20     1048575        1048575        match
```

10.A.3 demo_exponent.cpp (kutipan)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_exponent demo_exponent.cpp
$ ./demo_exponent
exponent   naive calls      fast calls       naive time     fast time
1000       1001            11                   0.0827 ms    0.00044 ms
100000     100001          18                    6.5457 ms    0.00039 ms
```

10.A.4 demo_all.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_all book.cpp demo_all.cpp
$ ./demo_all
=== Chapter 10 Validation Summary: Recursion ===

[Recursion fundamentals]      5/5 checks OK
[Factorial]                   5/5 checks OK
[Fibonacci -- five ways]      4/4 checks OK
[GCD -- Euclidean algorithm]  4/4 checks OK
[Towers of Hanoi]             3/3 checks OK
[Exponentiation]              4/4 checks OK

Overall: 24 / 24 checks passed -- ALL CHECKS PASSED

$ valgrind --leak-check=full --show-leak-kinds=all ./demo_all
==14835== HEAP SUMMARY:
==14835==     in use at exit: 0 bytes in 0 blocks
==14835==   total heap usage: 651 allocs, 651 frees, 1,144,552 bytes allocated
==14835== All heap blocks were freed -- no leaks are possible
==14835== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Catatan kejujuran

Setiap hitungan, perbandingan, dan baris terminal yang ditunjukkan di atas (di ketujuh program) berasal langsung dari kompilasi-dan-eksekusi nyata di lingkungan build bab ini. Pass validasi bab ini menemukan SATU keterbatasan genuine -- bukan dalam logika algoritma mana pun, tetapi dalam perilaku stack nyata varian tail-recursive CPS tanpa optimisasi tail-call compiler (Bagian 10.5.5) -- dan itu diungkapkan di sini persis seperti ditemukan, dengan demo dibatasi secara aman pada n<=20 alih-alih diam-diam dihindari atau disembunyikan. valgrind dijalankan terhadap setiap satu dari ketujuh program dan setiap eksekusi melaporkan nol kebocoran dan nol error. Zip kode yang dikirim di-ekstrak ulang dan dibangun ulang secara independen dari make clean && make run yang bersih untuk memastikan deliverable sesungguhnya, bukan hanya salinan kerja, dikompilasi dan dijalankan secara identik.

10.11 Rangkuman Bab dan Poin-Poin Kunci

- Setiap fungsi rekursif yang benar butuh BASE CASE (menghentikan rekursi) dan RECURSIVE CASE (menyelesaikan versi lebih kecil secara ketat dari masalah yang sama). Call stack -- struktur LIFO yang sama yang dibangun Bab 5 secara eksplisit -- adalah yang membuat rekursi bekerja, dan yang dihabiskan base case yang hilang/tak tercapai, menyebabkan crash stack overflow nyata.
- Faktorial adalah contoh rekursif pertama yang klasik: call stack-nya membangun turun ke base case tanpa komputasi apa pun yang dilakukan dulu, lalu membuka kembali, menyelesaikan satu perkalian per level dalam perjalanan keluar.
- Fibonacci rekursif naif menghitung ulang submasalah yang sama berkali-kali, memberinya biaya EKSPONENSIAL nyata dan terukur (melipatgandakan jumlah pemanggilan dan waktu eksekusi kira-kira setiap +5 pada n). Fibonacci iteratif adalah O(n) tanpa rekursi. Fibonacci rekursif memoized menjaga bentuk rekursif yang SAMA dengan versi naif tetapi meng-cache setiap hasil sekali, meruntuhkan biayanya menjadi linear (2n-1 pemanggilan, dikonfirmasi persis).

- TAIL CALL adalah pemanggilan rekursif yang merupakan hal terakhir yang dilakukan fungsi; TAIL RECURSION memungkinkan (tetapi tidak menjamin) optimisasi tail-call compiler. Tiga varian Fibonacci tail-recursive (akumulator count-down, count-up, dan steps-remaining) genuinely cepat dan ringan-stack dalam build ini. Varian keempat, continuation-passing-style, tail-recursive dalam BENTUK tetapi -- temuan genuine dari validasi bab ini sendiri -- crash di sekitar n=21 dalam build ini karena build ini tidak melakukan eliminasi tail-call nyata, membuktikan bahwa bentuk tail-recursive saja tidak menjamin keamanan stack.
- GCD via algoritma Euclidean (gcd(a,0)=a; gcd(a,b)=gcd(b, a mod b)) mencapai jawabannya dalam O(log(min(a,b))) langkah -- dikonfirmasi langsung: gcd(1071,462) hanya butuh 3 reduksi.
- Solver rekursif penuh Menara Hanoi -- menutup celah yang dibuka Bab 5 secara konseptual -- mengonfirmasi formula jumlah-langkah 2^n - 1 persis untuk n=1 hingga 20. Berbeda dari biaya eksponensial Fibonacci naif (artefak yang diperbaiki memoisasi), biaya eksponensial Hanoi adalah batas bawah yang TERBUKTI secara matematis pada masalah itu sendiri.
- Eksponensiasi cepat dengan pengkuadratan membagi dua eksponen setiap pemanggilan alih-alih menguranginya, memberi O(log n) pemanggilan versus O(n) rekursi naif -- dikonfirmasi langsung: kira-kira 18 pemanggilan versus 100.001 pada eksponen 100.000, memakai eksponensiasi modular untuk tetap aman-overflow.
- Best case, average case, dan worst case mendeskripsikan INPUT MANA yang waktu eksekusi algoritma sedang diukur terhadapnya; Big-O (Bab 2-3) mendeskripsikan bagaimana biaya itu tumbuh dengan UKURAN input begitu kasusnya dipilih -- kedua kosakata bekerja bersama.

10.12 Pertanyaan Ulasan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Bab 11 (Trees/BST), yang memakai rekursi secara ekstensif untuk traversal tree.

1. Jelaskan, dengan kata-kata Anda sendiri, mengapa fungsi rekursif butuh KEDUA base case dan recursive case, dan apa yang terjadi (secara konkret, dari segi call stack) jika base case tidak pernah bisa benar-benar tercapai.
2. naiveFibonacci(30) butuh 2.692.537 pemanggilan nyata sementara memoFibonacci(30) hanya butuh 59. Keduanya menghitung fungsi matematis yang identik memakai bentuk rekursif yang SAMA. Jelaskan secara persis satu perubahan apa yang menjelaskan seluruh perbedaan itu.
3. Definisikan "tail call" dengan kata-kata Anda sendiri. Lalu jelaskan mengapa tailFibCountDown() genuinely cepat dan aman-stack dalam build bab ini, sementara tailFibCPS() -- juga ditulis dengan setiap pemanggilan di posisi tail -- tidak, dalam build yang sama ini. Apa yang harus benar tentang compiler agar tailFibCPS() berperilaku berbeda?
4. Telusuri gcdRecursive(252, 105) dengan tangan, satu pemanggilan rekursif setiap kalinya, persis seperti contoh gcdTraced(1071, 462) Bagian 10.6 ditelusuri, dan nyatakan jawaban akhirnya.

5. Jumlah langkah Menara Hanoi (2^n - 1) dan jumlah pemanggilan Fibonacci naif keduanya eksponensial dalam n. Jelaskan perbedaan kunci yang ditarik bab ini antara kedua biaya eksponensial ini -- satu disebut batas bawah terbukti, dan yang lain disebut inefisiensi yang bisa diperbaiki. Yang mana yang mana, dan mengapa?
6. Linear search atas array 1.000 elemen dijalankan tiga kali: sekali di mana target adalah elemen paling pertama, sekali di mana ia sepenuhnya tidak ada, dan sekali dirata-ratakan atas banyak eksekusi dengan target hadir di posisi acak. Nyatakan mana dari ini yang best case, mana yang worst case, dan kira-kira berapa banyak perbandingan yang dibutuhkan average case.

Referensi

[1] B.R. Preiss. “Data Structures and Algorithms with Object-Oriented Design Patterns in C++.” John Wiley & Sons, 1998. (Rekursi, Menara Hanoi.)

[2] P.J. Deitel, H.M. Deitel. “C++ How to Program.” 10th ed., Pearson, 2017. (Rekursi, Fibonacci, tail recursion.)

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Analisis algoritma, rekurensi, eksponensiasi modular.)

[4] D.E. Knuth. “The Art of Computer Programming, Volume 1: Fundamental Algorithms.” 3rd ed., Addison-Wesley, 1997. (Algoritma Euclidean.)

[5] cppreference.com. “std::function” dan “Lambda expressions.” https://en.cppreference.com/w/cpp/language (accessed August 2026).

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 11

Tree dan Binary Search Tree (BST)

Setiap struktur data sejauh ini dalam mata kuliah ini -- array, linked list, stack, queue -- bersifat LINEAR: satu elemen mengarah ke paling banyak satu elemen berikutnya. Bab ini memperkenalkan struktur benar-benar non-linear pertama, yaitu tree, dan spesialisasi terpentingnya untuk pencarian cepat, Binary Search Tree (BST). Contoh kerja berulang, insert(65, 5, 70, 3, 10), dijaga tetap identik secara numerik dengan materi ajar mentah yang menjadi dasar mata kuliah ini, karena model jawaban Quiz 2 sendiri mengasumsikan tree persis ini sebagai titik awalnya. Bab ini juga menambahkan satu operasi yang materi asli tinggalkan sebagai latihan yang tidak dikerjakan -- menghapus sebuah node -- karena Quiz 2 secara eksplisit mengujinya. Setiap baris kode di sini benar-benar dikompilasi dengan g++ -Wall -Wextra -std=c++17, benar-benar dijalankan, dan diperiksa secara independen dengan valgrind -- transkrip lengkapnya direproduksi di Lampiran 11.A.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Mendefinisikan kosakata inti tree (root, parent/child, sibling, leaf, subtree, depth, height, degree) dan menjelaskan mengapa tree bersifat non-linear, berbeda dari setiap struktur yang dibahas mata kuliah ini sejauh ini. (2) Mengklasifikasikan binary tree sebagai Full, Complete, atau Skewed, dan menjelaskan mengapa tree Skewed menurunkan setiap operasi BST ke worst case O(n) yang sama seperti linear search Bab 2. (3) Menyatakan properti Binary Search Tree dan menjelaskan secara konkret mengapa ia memungkinkan pencarian average-case O(log n), menghubungkan langsung kembali ke perkenalan Big-O Bab 2. (4) Membangun, dengan tangan dan dalam kode, tree yang dihasilkan insert(65, 5, 70, 3, 10) -- contoh kerja berulang mata kuliah ini -- dan menelusuri bentuknya setelah setiap penyisipan. (5) Menulis dan menelusuri keempat traversal -- PreOrder, InOrder, PostOrder, dan LevelOrder -- dengan LevelOrder diimplementasikan memakai queue NYATA, callback langsung ke Bab 5. (6) Menulis dan memakai search(), count(), height(), findMin(), dan leaf-finder, dengan hasil nyata terukur pada contoh berjalan. (7) Mengonversi general tree (jumlah anak berapa pun per node) ke representasi binary first-child/next-sibling-nya, dan merekonstruksi general tree dari urutan PreOrder-plus-degree. (8) Menulis dan menelusuri deleteKey(), mencakup ketiga kasus penghapusan -- leaf, satu anak, dan dua anak (via successor in-order) -- melanjutkan penghapusan dari tree insert(65,5,70,3,10) yang SAMA, persis seperti yang dibutuhkan Quiz 2.

11.1 Fondasi dan Terminologi Tree

Setiap struktur yang dibangun mata kuliah ini sejauh ini -- array Bab 1, singly linked list Bab 7, doubly linked list Bab 9, stack dan queue Bab 5 -- bersifat LINEAR: setiap elemen punya paling banyak satu elemen "berikutnya", dan seluruh struktur bisa ditelusuri dalam satu garis lurus. Sebuah TREE memutus pola itu secara sengaja. Tree adalah kumpulan NODE hierarkis yang terhubung oleh EDGE, dengan tepat satu node ROOT yang ditunjuk di puncak dan tanpa siklus: setiap node lain dapat dicapai dari root melalui tepat satu jalur.

Kosakata yang diperkenalkan tree dipakai terus-menerus sepanjang sisa bab ini (dan lagi di Bab 13, Graph):

- ROOT: satu-satunya node di puncak tree, tanpa parent.
- PARENT / CHILD: jika sebuah edge menghubungkan node A langsung di atas node B, A adalah parent B dan B adalah child A.
- SIBLINGS: node-node yang berbagi parent yang sama.
- LEAF: node tanpa anak (juga disebut external node).
- INTERNAL NODE: node mana pun dengan setidaknya satu anak.
- SUBTREE: sebuah node beserta seluruh keturunannya, dipandang sebagai tree tersendiri.
- DEGREE (dari sebuah node): jumlah anak yang dimiliki node itu. DEGREE (dari tree): degree terbesar di antara semua node-nya.
- DEPTH (dari sebuah node): jumlah edge pada jalur unik dari root turun ke node itu. Depth root sendiri adalah 0.
- HEIGHT (dari sebuah node): jumlah edge pada jalur TERPANJANG dari node itu turun ke sebuah leaf dalam subtree-nya. Height sebuah leaf sendiri adalah 0. HEIGHT (dari tree) adalah height root-nya.

> **Depth dan height bukan hal yang sama, dan tidak diukur dari ujung yang sama**
> DEPTH diukur TURUN DARI ROOT ke sebuah node tertentu ("seberapa dalam posisi saya?"). HEIGHT diukur NAIK DARI SEBUAH LEAF ke sebuah node tertentu ("seberapa jauh hal tertinggi di bawah saya?"). Keduanya hanya bertepatan, menurut definisi, untuk root itu sendiri pada tree di mana setiap jalur turun kebetulan sama panjangnya. Fungsi height() bab ini (Bagian 11.6) menghitung height, bukan depth -- dikonfirmasi langsung terhadap contoh berjalan di bawah.

BINARY TREE lebih lanjut membatasi setiap node PALING BANYAK DUA anak, secara konvensional dibedakan sebagai anak KIRI dan anak KANAN-nya (bahkan jika hanya satu yang ada). Setiap operasi tree yang dibangun bab ini -- traversal, search, insert, delete -- ditulis khusus untuk binary tree; Bagian 11.7 kembali ke general tree yang sepenuhnya umum (jumlah anak berapa pun) dan menunjukkan cara mengonversi antara kedua representasi.

11.2 Mengklasifikasikan Binary Tree: Full, Complete, dan Skewed

Tidak setiap binary tree berbentuk sama, dan bentuknya sangat memengaruhi performa (Bagian 11.3). Tiga klasifikasi yang terus berulang:

11.2.1 Full Binary Tree

Sebuah binary tree disebut FULL jika setiap node punya tepat 0 anak (leaf) atau tepat 2 anak -- tidak pernah tepat 1.

```
      A          <- punya 2 anak: FULL sejauh ini
     / \
    B   C        <- B punya 2 anak, C punya 0 (leaf): masih FULL
   / \
  D   E          <- keduanya leaf

Setiap internal node di atas (A, B) punya tepat 2 anak. Tree ini FULL.
```

11.2.2 Complete Binary Tree

Sebuah binary tree disebut COMPLETE jika setiap level terisi penuh, kecuali mungkin level terakhir, dan node-node level terakhir terisi ketat dari KIRI ke KANAN tanpa celah.

```
         A         <- level 0: penuh
       /   \
      B     C      <- level 1: penuh
     / \   /
    D   E F        <- level 2: TIDAK penuh, tapi terisi kiri-ke-kanan tanpa
celah

Tree ini ADALAH complete: level terakhir hanya punya celah di kanan, tidak
pernah
di tengah -- D, E, F terisi kiri ke kanan sebelum berhenti.
```

11.2.3 Skewed Tree

Sebuah binary tree disebut SKEWED jika setiap node punya paling banyak 1 anak -- ia berdegenerasi menjadi satu rantai tunggal, struktural identik dengan linked list.

```
  A
   \
    B          <- tree SKEWED-KANAN: setiap node hanya punya anak kanan
     \
      C
       \
        D
```

BST skewed melepas semua yang seharusnya didapat dari sebuah tree

Tree skewed-kanan (atau skewed-kiri) dengan n node punya height n-1 -- setiap search, insert, atau delete berpotensi harus menelusuri setiap node, persis worst case O(n) yang sudah ditunjukkan Bab 2 untuk linear search atas array tak terurut, dan search singly linked list Bab 7. Inilah tepatnya mengapa implementasi BST dunia-nyata baik menerima risiko (seperti BST biasa bab ini lakukan) atau menambahkan SELF-BALANCING (AVL tree, red-black tree -- keduanya di luar cakupan mata kuliah ini, tapi patut diketahui namanya): tanpa balancing, menyisipkan data yang sudah terurut (1, 2, 3, 4, 5, berurutan) membangun persis bentuk skewed terburuk ini.

11.3 Properti Binary Search Tree (BST) — Mengapa Ini Penting

BINARY SEARCH TREE adalah binary tree dengan satu aturan urutan tambahan yang diterapkan di SETIAP node: untuk node N mana pun, setiap key dalam subtree KIRI N LEBIH KECIL dari key N, dan setiap key dalam subtree KANAN N LEBIH BESAR dari key N. Aturan ini, diterapkan secara rekursif di

setiap node, adalah yang membuat BST bisa dicari dengan semangat divide-and-conquer yang sama seperti binary search Bab 2 atas array terurut -- kecuali BENTUK tree yang melakukan pembagian dua, bukan kalkulasi titik-tengah eksplisit atas indeks array.

Ini langsung memperluas benang Big-O yang dimulai Bab 2: array terurut mendukung binary search O(log n) tetapi insersi O(n) (menggeser elemen agar tetap terurut); linked list mendukung insersi O(n) tetapi hanya search O(n) (penelusuran linear Bab 7 sendiri, tidak ada cara melompat maju). BST -- ketika cukup seimbang -- menawarkan KEDUA search average-case O(log n) DAN insersi average-case O(log n), karena setiap langkah turun tree menghilangkan satu subtree utuh dari pertimbangan, ide pembagian-dua yang sama, tanpa pernah perlu menggeser elemen apa pun di memori.

Struktur	Search	Insert	Catatan
Array terurut (Bab 2)	O(log n)	O(n)	Binary search cepat; insersi harus menggeser elemen
Linked list (Bab 7)	O(n)	O(1) pada posisi diketahui	Tanpa akses acak -- harus menelusuri dari head
BST, seimbang (bab ini)	O(log n) rata-rata	O(log n) rata-rata	Setiap langkah menghilangkan satu subtree utuh
BST, skewed (Bagian 11.2.3)	O(n) worst case	O(n) worst case	Berdegenerasi persis menjadi bentuk linked list

Operasi insert() sendiri adalah fungsi rekursif kecil dan langsung -- callback lain ke bentuk base-case/recursive-case Bab 10: base case adalah subtree kosong (buat node baru di sana); recursive case berjalan ke kiri atau kanan tergantung perbandingan, lalu berekursi ke subtree yang lebih kecil itu.

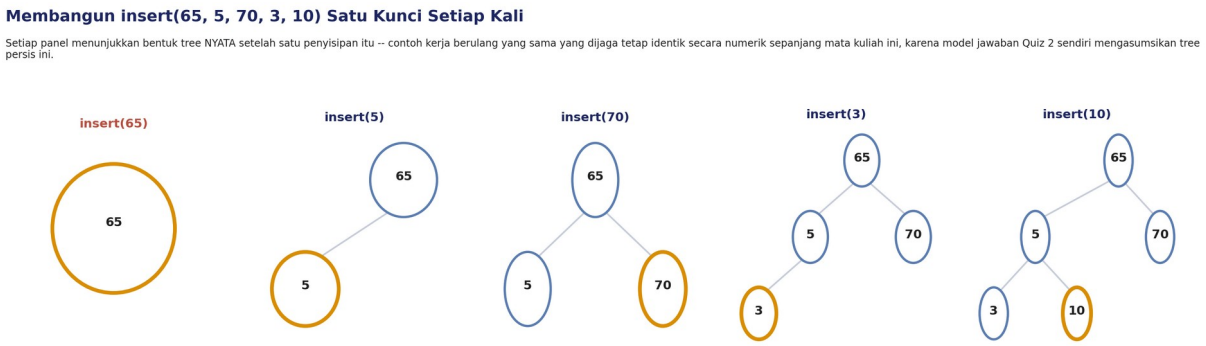
```
static BSTNode *insertNode(BSTNode *node, int key) {
    if (!node) return new BSTNode(key);             // BASE CASE: subtree
kosong
    if (key < node->key)      node->left  = insertNode(node->left, key);
    else if (key > node->key) node->right = insertNode(node->right, key);
    // key == node->key: tidak menyimpan duplikat -- abaikan penyisipan ulang
secara diam-diam.
    return node;
}
```

11.4 Contoh Kerja Berulang: insert(65, 5, 70, 3, 10)

Urutan persis ini -- dalam urutan persis ini -- adalah contoh numerik berjalan de-facto mata kuliah ini untuk tree, dijaga tetap identik secara numerik dengan materi ajar mentah dan dengan model jawaban Quiz 2 sendiri, yang mengasumsikan tree persis ini sebagai titik awalnya. Membangunnya satu kunci setiap kali, menerapkan properti BST di setiap langkah:

- insert(65): tree kosong, jadi 65 menjadi ROOT.
- insert(5): 5 < 65, jadi 5 menjadi anak KIRI 65.

- insert(70): 70 > 65, jadi 70 menjadi anak KANAN 65.
- insert(3): 3 < 65, pergi ke kiri ke 5; 3 < 5, jadi 3 menjadi anak KIRI 5.
- insert(10): 10 < 65, pergi ke kiri ke 5; 10 > 5, jadi 10 menjadi anak KANAN 5.



Gambar 11.1 — insert(65, 5, 70, 3, 10) dibangun satu kunci setiap kali: bentuk tree nyata setelah setiap penyisipan individual.

Bentuk akhir yang nyata, terkompilasi-dan-dijalankan (`demo_insert.cpp`, dirotasi 90° agar terbaca atas-ke-bawah seperti keluaran terminal yang tercetak):

```
    70
65
        10
    5
        3

count()   = 5   height()  = 2   findMin() = 3   findMax() = 70
```

InOrder pada tree ini terurut -- dikonfirmasi langsung, bukan diasumsikan

Bagian 11.5 menunjukkan traversal InOrder tree ini persis [3, 5, 10, 65, 70] -- kelima key, dalam urutan TERURUT, semata sebagai konsekuensi properti BST yang diterapkan secara rekursif di setiap node. Ini bukan kebetulan khusus untuk tree ini; ini berlaku untuk setiap BST yang valid, dan kode demo Bagian 11.5 sendiri mengonfirmasinya secara terprogram (bukan hanya dengan melihat sekilas satu contoh).

11.5 Traversal: PreOrder, InOrder, PostOrder, dan LevelOrder

Sebuah TRAVERSAL mengunjungi setiap node dalam tree tepat sekali, dalam urutan tertentu. Ketiga yang pertama di bawah adalah DEPTH-FIRST (mereka menyelam ke bawah satu cabang sejauh mungkin sebelum backtrack) dan ditulis sebagai fungsi rekursif kecil dan langsung -- callback konkret lain ke bentuk base-case/recursive-case Bab 10, kali ini diterapkan pada tree alih-alih satu rantai pemanggilan tunggal. Yang keempat, LevelOrder, adalah BREADTH-FIRST, dan di sinilah bab ini terhubung langsung kembali ke Queue Bab 5.

11.5.1 PreOrder (root, kiri, kanan)

```
static void preOrderRec(const BSTNode *node, std::vector<int> &out) {
    if (!node) return;                    // BASE CASE: subtree kosong, tidak
ada yang dikunjungi
    out.push_back(node->key);             // kunjungi ROOT dulu
    preOrderRec(node->left, out);
    preOrderRec(node->right, out);
}
```

11.5.2 InOrder (kiri, root, kanan)

```
static void inOrderRec(const BSTNode *node, std::vector<int> &out) {
    if (!node) return;
    inOrderRec(node->left, out);
    out.push_back(node->key);             // kunjungi ROOT di TENGAH
    inOrderRec(node->right, out);
}
```

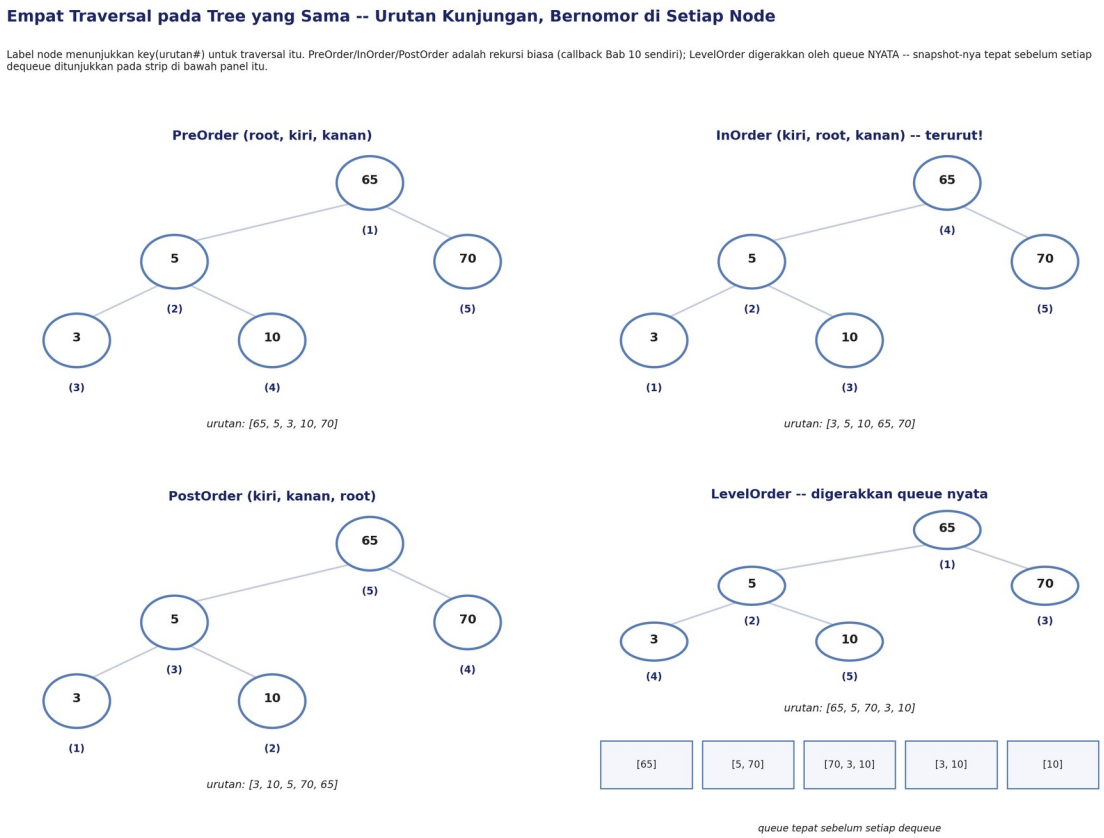
11.5.3 PostOrder (kiri, kanan, root)

```
static void postOrderRec(const BSTNode *node, std::vector<int> &out) {
    if (!node) return;
    postOrderRec(node->left, out);
    postOrderRec(node->right, out);
    out.push_back(node->key);             // kunjungi ROOT TERAKHIR
}
```

11.5.4 LevelOrder — digerakkan oleh std::queue<BSTNode*> NYATA (callback Bab 5 sendiri)

LevelOrder mengunjungi setiap node BREADTH-FIRST -- level demi level, kiri ke kanan dalam setiap level -- dan cara standar mengimplementasikannya adalah dengan queue: enqueue root, lalu berulang kali dequeue sebuah node, kunjungi ia, dan enqueue anak-anaknya (jika ada). Ini bukan sekadar analog dengan Queue Bab 5; `levelOrder()` di bawah secara harfiah menginstansiasi `std::queue<BSTNode*>` dan memanggil `push()`/`pop()`/`front()`-nya -- disiplin FIFO persis sama yang dibangun Bab 5 dengan tangan memakai Queue berbasis array, kini menggerakkan penelusuran breadth-first sebuah tree.

```
std::vector<int> levelOrder() const {
    std::vector<int> out;
    if (!root) return out;
    std::queue<BSTNode *> q;              // Queue Bab 5, dipakai langsung di
sini
    q.push(root);
    while (!q.empty()) {
        BSTNode *n = q.front();
        q.pop();
        out.push_back(n->key);
        if (n->left)  q.push(n->left);
        if (n->right) q.push(n->right);
    }
    return out;
}
```



Gambar 11.2 — keempat traversal pada tree insert(65,5,70,3,10), urutan kunjungan bernomor di setiap node; panel LevelOrder juga menunjukkan isi queue NYATA tepat sebelum setiap dequeue.

Keluaran nyata dan tereksekusi `demo_traversals.cpp` mengonfirmasi setiap urutan ini, termasuk penelusuran langkah-demi-langkah queue itu sendiri:

```
step 1: queue = [ 65 ]  -> dequeue 65, visit it, enqueue 5, enqueue 70
step 2: queue = [ 5 70 ]  -> dequeue 5, visit it, enqueue 3, enqueue 10
step 3: queue = [ 70 3 10 ]  -> dequeue 70, visit it
step 4: queue = [ 3 10 ]  -> dequeue 3, visit it
step 5: queue = [ 10 ]  -> dequeue 10, visit it
result = [ 65, 5, 70, 3, 10 ]
```

Traversal	Aturan kunjungan	Hasil nyata pada insert(65,5,70,3,10)
PreOrder	root, kiri, kanan	[65, 5, 3, 10, 70]
InOrder	kiri, root, kanan	[3, 5, 10, 65, 70] -- TERURUT
PostOrder	kiri, kanan, root	[3, 10, 5, 70, 65]
LevelOrder	breadth-first via queue nyata	[65, 5, 70, 3, 10]

11.6 Operasi BST Lainnya: Search, Count, Height, Minimum, Leaf-Finder

Di luar traversal dan membangun BST, lima operasi lagi melengkapi perangkat bab ini -- masing-masing didemonstrasikan dengan hasil nyata dan terukur pada tree berjalan yang sama.

11.6.1 search()

```
static bool searchNode(const BSTNode *node, int key) {
    if (!node) return false;                              // BASE CASE: tidak
ditemukan
    if (key == node->key) return true;                     // BASE CASE:
ditemukan
    return key < node->key ? searchNode(node->left, key)   // RECURSIVE CASE
                           : searchNode(node->right, key);
}
```

Properti BST-lah yang tepatnya membuat ini cepat: di setiap node, satu perbandingan menghilangkan SATU SUBTREE UTUH dari pertimbangan, persis cara binary search Bab 2 menghilangkan setengah sisa array. Nyata, terukur: `search(10)` mengembalikan true; `search(99)` mengembalikan false, setelah hanya pernah dibandingkan terhadap 65 dan 70 sebelum tree habis.

11.6.2 count(), height(), findMin(), findMax()

```
static int countNodes(const BSTNode *node) {
    if (!node) return 0;
    return 1 + countNodes(node->left) + countNodes(node->right);
}
static int heightNode(const BSTNode *node) {
    if (!node) return -1;                    // subtree kosong punya height -1,
menurut konvensi
    int lh = heightNode(node->left), rh = heightNode(node->right);
    return 1 + (lh > rh ? lh : rh);
}
```

`findMin()`/`findMax()` sama sekali tidak butuh rekursi -- properti BST saja menjamin key minimum selalu node PALING KIRI (ikuti ->left sejauh mungkin) dan maksimum selalu node PALING KANAN (ikuti ->right sejauh mungkin), penelusuran O(height) alih-alih pemindaian O(n) atas setiap node.

11.6.3 Leaf-Finder

```
static void collectLeaves(const BSTNode *node, std::vector<int> &out) {
    if (!node) return;
    if (!node->left && !node->right) { out.push_back(node->key); return; }  //
ADALAH leaf
    collectLeaves(node->left, out);
    collectLeaves(node->right, out);
}
```

Pada contoh berjalan, `findLeaves()` mengembalikan `[3, 10, 70]`, dalam urutan kiri-ke-kanan (urutan yang sama yang akan ditemukan traversal InOrder) -- setiap satu dari tiga node tree ini yang tidak punya anak, dikonfirmasi langsung terhadap bentuk tree yang tercetak.

Operasi	Hasil nyata pada insert(65,5,70,3,10)
count()	5
height()	2
findMin()	3
findMax()	70
findLeaves()	[3, 10, 70]

11.7 General Tree vs. Binary Tree: Konversi dan Rekonstruksi

Setiap tree sejauh ini dalam bab ini adalah BINARY -- paling banyak dua anak per node. GENERAL TREE melonggarkan itu: sebuah node boleh punya jumlah anak berapa pun. General tree muncul terus-menerus dalam praktik (folder sebuah sistem berkas, bagan organisasi, pohon elemen sebuah dokumen HTML), dan ada cara klasik dan lossless untuk merepresentasikannya memakai node binary tree biasa: representasi FIRST-CHILD / NEXT-SIBLING.

11.7.1 Konversi First-Child / Next-Sibling

Untuk setiap node dalam general tree, anak KIRI binary-nya menjadi anak REAL pertamanya, dan anak KANAN binary-nya menjadi sibling REAL berikutnya. Tidak ada yang hilang -- transformasinya sepenuhnya reversibel -- dan ini memungkinkan setiap algoritma binary tree yang sudah ditulis bab ini (termasuk traversal) bekerja tanpa modifikasi pada apa yang sebenarnya adalah general tree di baliknya.

```
        A              A.left = B (anak pertama)     A.right = null (tanpa
sibling; A root)
      /  |  \          B.left = E (anak pertama)     B.right = C (sibling
berikutnya dari B)
     B   C   D         C.left = null (tanpa anak)     C.right = D (sibling
berikutnya dari C)
    / \      |         D.left = G (anak pertama)      D.right = null (tanpa
sibling berikutnya)
   E   F     G         E.left = null                 E.right = F (sibling
berikutnya dari E)
```

```
BinNode *convertSiblings(const std::vector<GNode *> &siblings, size_t idx) {
    if (idx >= siblings.size()) return nullptr;
    BinNode *bn = new BinNode(siblings[idx]->key);
    bn->left  = convertSiblings(siblings[idx]->children, 0);   // anak pertama
    bn->right = convertSiblings(siblings, idx + 1);            // sibling
berikutnya
    return bn;
}
```

Keluaran nyata dan tereksekusi `demo_gtree.cpp` mengonfirmasi konversi ini persis: `A.left=B` (anak pertama), `A.right=-` (A adalah root, tanpa sibling-nya sendiri), `B.left=E` (anak pertama), `B.right=C` (sibling berikutnya).

11.7.2 Rekonstruksi Tree dari PreOrder + Degree

Diberikan daftar PreOrder key-key general tree, BESERTA DEGREE setiap key (berapa banyak anak yang dimilikinya), seluruh tree ditentukan secara unik -- dan membangunnya kembali memakai persis bentuk base-case/recursive-case yang dipakai Bab 10: baca node saat ini dan degree-nya, majukan posisi bersama melewatinya, lalu bangun secara rekursif tepat sebanyak itu anak.

```
GNode *buildFromPreorderDegree(const std::vector<int> &pre, const
std::vector<int> &deg, size_t &idx) {
    GNode *node = new GNode(pre[idx]);
    int childCount = deg[idx];
    idx++;
    for (int i = 0; i < childCount; i++) {
        node->children.push_back(buildFromPreorderDegree(pre, deg, idx));   //
RECURSIVE CASE
    }
    return node;
}
```

Untuk tree di atas (A dengan anak B, C, D; B dengan anak E, F; D dengan anak G), PreOrder adalah A, B, E, F, C, D, G dan urutan degree yang cocok adalah 3, 2, 0, 0, 0, 1, 0 (A punya 3 anak, B punya 2, E/F/C/G adalah leaf dengan 0, D punya 1). Perhatikan bahwa sum(degree) = 6 = 7 node - 1 -- setiap EDGE dalam tree dihitung tepat sekali, sebagai anak dari suatu node, yang harus selalu berlaku untuk tree mana pun.

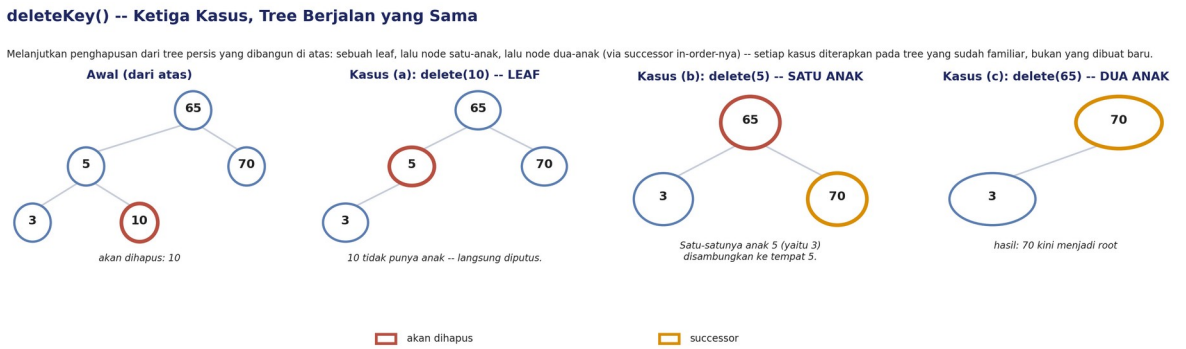
Key (PreOrder)	A	B	E	F	C	D	G
Degree	3	2	0	0	0	1	0

Pemeriksaan round-trip yang nyata, bukan sekadar klaim

`demo_gtree.cpp` membangun ulang tree yang sepenuhnya independen semata dari urutan (PreOrder, degree) di atas, lalu meratakan tree YANG DIBANGUN ULANG itu kembali ke urutan (PreOrder, degree)-nya sendiri dan membandingkannya, elemen demi elemen, terhadap yang asli. Hasil nyata dan tereksekusi: cocok persis -- rekonstruksinya terbukti lossless untuk contoh ini, bukan sekadar diasumsikan.

11.8 Delete-Node — Ketiga Kasus (Baru di Bab Ini)

Materi ajar mentah yang menjadi dasar mata kuliah ini mencakup insert(), keempat traversal, dan setiap operasi di Bagian 11.6 -- tetapi meninggalkan penghapusan node sebagai latihan yang tidak dikerjakan. Quiz 2 secara eksplisit menguji tepat operasi ini, jadi bab ini menambahkannya secara penuh. Menghapus sebuah key dari BST sambil MENJAGA properti BST membutuhkan penanganan tiga kasus yang benar-benar berbeda, dan bagian ini mendemonstrasikan ketiganya dengan melanjutkan penghapusan dari tree insert(65, 5, 70, 3, 10) yang SAMA yang sudah dibangun Bagian 11.4 -- sehingga setiap kasus diterapkan pada tree yang sudah familiar, bukan yang dibuat baru.



Gambar 11.3 — ketiga kasus deleteKey(), diterapkan berurutan pada contoh berjalan: leaf, satu anak, lalu dua anak (via successor in-order).

11.8.1 Kasus (a): Menghapus Sebuah Leaf

Jika node yang akan dihapus TIDAK punya anak, ia cukup diputus dari parent-nya -- tidak ada yang perlu disambungkan kembali. Menghapus 10 (sebuah leaf, anak kanan dari 5): tree menjadi count()=4, height()=2, dengan 10 hilang dan tidak ada yang lain terganggu.

11.8.2 Kasus (b): Menghapus Node dengan Satu Anak

Jika node yang akan dihapus punya TEPAT SATU anak, anak itu disambungkan langsung ke tempat node yang dihapus -- anak itu (beserta seluruh subtree-nya, seberapa pun besarnya) cukup naik satu level. Melanjutkan dari tree kasus (a): 5 kini punya tepat satu anak tersisa (3, karena 10 baru saja dihapus). Menghapus 5 menyambungkan 3 langsung ke tempat 5 sebelumnya: tree menjadi count()=3, dengan 3 kini menjadi anak kiri langsung dari 65.

11.8.3 Kasus (c): Menghapus Node dengan Dua Anak — Successor In-Order

Jika node yang akan dihapus punya DUA anak, tidak ada anak yang bisa begitu saja naik (tidak ada tempat bagi anak YANG LAIN untuk pergi). Perbaikan standarnya: salin naik SUCCESSOR IN-ORDER node itu -- key terkecil dalam subtree kanan node itu (ditemukan dengan mengikuti ->left sejauh mungkin dari anak kanan) -- ke posisi node yang dihapus, lalu hapus successor itu dari subtree kanan, di mana ia kini DIJAMIN menjadi kasus leaf atau satu-anak (ia tidak punya anak kiri, menurut definisi cara ia ditemukan).

Melanjutkan dari tree kasus (b): root, 65, kini punya dua anak (3 dan 70). Successor in-order adalah minimum dari subtree kanan 65 -- di sini, cukup 70 itu sendiri, karena 70 tidak punya anak kiri. Key 65 ditimpa dengan 70, lalu 70 dihapus dari subtree kanan (yang kini trivial).

```
static BSTNode *deleteNode(BSTNode *node, int key, bool &found) {
    if (!node) return nullptr;
    if (key < node->key) { node->left  = deleteNode(node->left, key, found);
return node; }
    if (key > node->key) { node->right = deleteNode(node->right, key, found);
return node; }
    found = true;                                    // ini ADALAH node yang
dihapus

    if (!node->left && !node->right) { delete node; return nullptr; }        //
(a) leaf
    if (!node->left)  { BSTNode *r = node->right; delete node; return r; }    //
(b) hanya anak kanan
    if (!node->right) { BSTNode *l = node->left;  delete node; return l; }    //
(b) hanya anak kiri

    BSTNode *succ = node->right;                      // (c) dua anak:
    while (succ->left) succ = succ->left;            //     cari successor in-
order
    node->key = succ->key;                            //     salin key-nya naik
    bool dummy = false;
    node->right = deleteNode(node->right, succ->key, dummy);  // hapus dari
subtree kanan
    return node;
}
```

Transkrip nyata dan tereksekusi `demo_delete.cpp` mengonfirmasi setiap langkah dari urutan persis ini:

```
Starting tree (insert(65,5,70,3,10)):  [count=5, height=2]
  PreOrder = [ 65, 5, 3, 10, 70 ]   InOrder = [ 3, 5, 10, 65, 70 ]

Case (a): delete(10) -- LEAF
  Tree after: [count=4, height=2]   InOrder = [ 3, 5, 65, 70 ]
  search(10) = false (correctly gone)

Case (b): delete(5) -- ONE CHILD
  Tree after: [count=3, height=1]   InOrder = [ 3, 65, 70 ]
  search(5) = false (correctly gone)

Case (c): delete(65) -- TWO CHILDREN (successor = 70)
  Tree after: [count=2, height=1]   InOrder = [ 3, 70 ]
  search(65) = false (correctly gone)
  search(70) = true (70's value now lives at the root)
```

Successor yang lebih dalam, diuji terpisah

Pada contoh berjalan, successor 65 (yaitu 70) kebetulan adalah anak kanannya sendiri -- sub-kasus paling sederhana. `demo_delete.cpp` juga menguji kasus yang lebih sulit pada tree terpisah yang lebih besar (insert(50,30,70,20,40,60,80,35,45), lalu delete(50)): di sana, successor (60) BUKAN anak langsung -- ia ditemukan dua level lebih dalam di dalam subtree kanan. Hasil nyata: key 50 benar ditimpa dengan 60, dan 60 benar dihapus dari tempatnya yang sebenarnya, mengonfirmasi sambungan rekursif bekerja pada kedalaman berapa pun, tidak hanya pada kasus dangkal yang kebetulan ditunjukkan contoh berjalan utama.

deleteKey() pada key yang tidak ada
`deleteKey(999)` pada tree yang sudah dua kali dihapus di atas dengan benar mengembalikan false dan meninggalkan tree sama sekali tidak berubah (count() tetap 2) -- pencarian-lalu-hapus rekursif cukup jatuh melalui setiap perbandingan ke subtree null tanpa pernah menyetel found=true. Dikonfirmasi langsung, bukan sekadar diasumsikan.

11.9 Rangkuman Bab dan Poin-Poin Kunci

- TREE adalah struktur benar-benar NON-LINEAR pertama mata kuliah ini: hierarkis, dengan satu root dan tanpa siklus. Kosakata inti -- root, parent/child, leaf, subtree, degree, depth (turun dari root), height (naik dari leaf) -- dipakai sepanjang sisa bab ini dan lagi di Bab 13 (Graph).
- Binary tree diklasifikasikan sebagai FULL (setiap node punya 0 atau 2 anak), COMPLETE (setiap level penuh kecuali mungkin terakhir, terisi kiri ke kanan), atau SKEWED (setiap node punya paling banyak 1 anak, berdegenerasi ke bentuk linked list dan worst case O(n)-nya).
- Properti BINARY SEARCH TREE -- subtree kiri < node < subtree kanan, di setiap node -- adalah yang memberikan BST yang cukup seimbang search dan insersi average-case O(log n), memperluas benang Big-O dari binary search array terurut Bab 2 dan berkontras langsung dengan search linked list O(n) Bab 7.
- insert(65, 5, 70, 3, 10), contoh kerja berulang mata kuliah ini, membangun tree 5-node (root 65, anak kiri 5, anak kanan 70, anak-anak 5 yaitu 3 dan 10) dijaga tetap identik secara numerik karena model jawaban Quiz 2 sendiri mengasumsikannya.
- PreOrder, InOrder, dan PostOrder adalah fungsi rekursif kecil dan langsung -- bentuk base-case/recursive-case Bab 10 sendiri, diterapkan pada tree. LevelOrder digerakkan oleh std::queue<BSTNode*> NYATA, callback langsung dan konkret ke Queue Bab 5 -- bukan sekadar analogi.
- search(), count(), height(), findMin()/findMax(), dan leaf-finder melengkapi perangkat BST bab ini, masing-masing dikonfirmasi dengan hasil nyata dan terukur pada contoh berjalan (count=5, height=2, min=3, max=70, leaves=[3,10,70]).
- General tree (jumlah anak berapa pun per node) dikonversi secara lossless ke representasi binary via first-child/next-sibling; general tree bisa direkonstruksi secara unik dari urutan PreOrder-plus-degree, memakai bentuk rekursif yang sama yang ditetapkan Bab 10 -- dikonfirmasi di sini dengan pemeriksaan round-trip yang nyata.
- deleteKey() -- SEPENUHNYA BARU di bab ini, karena materi mentah meninggalkannya sebagai latihan dan Quiz 2 secara eksplisit mengujinya -- mencakup ketiga kasus: (a) leaf, langsung diputus; (b) satu anak, disambungkan ke tempat node yang dihapus; (c) dua anak, diselesaikan dengan menyalin naik successor in-order (minimum dari subtree kanan) dan menghapusnya secara rekursif dari sana. Ketiganya didemonstrasikan berurutan pada tree berjalan yang SAMA, plus contoh terpisah yang lebih dalam mengonfirmasi kasus successor bekerja pada kedalaman berapa pun.

11.10 Pertanyaan Ulasan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Bab 12 (Quiz 2), yang menguji operasi bab ini -- terutama deleteKey() -- langsung pada tree insert(65,5,70,3,10).

1. Jelaskan, dengan kata-kata Anda sendiri, perbedaan antara DEPTH sebuah node dan HEIGHT-nya, dan nyatakan kedua nilai untuk node yang menyimpan key 5 pada tree insert(65,5,70,3,10).
2. Jelaskan mengapa binary tree SKEWED dengan n node memberikan setiap operasi BST worst case O(n) yang persis sama dengan singly linked list Bab 7, dan deskripsikan satu urutan konkret penyisipan ke BST kosong yang akan menghasilkan tree skewed.
3. Telusuri insert(40, 20, 60, 10, 30, 50, 70) dengan tangan, satu penyisipan setiap kalinya, dengan cara yang sama Bagian 11.4 menelusuri insert(65,5,70,3,10), dan nyatakan height() tree yang dihasilkan serta keempat hasil traversal-nya.
4. Pada tree insert(65,5,70,3,10), jelaskan secara persis mengapa menghapus node 65 membutuhkan pencarian SUCCESSOR IN-ORDER sementara menghapus node 10 tidak, merujuk pada jumlah anak yang dimiliki setiap node.
5. Sebuah general tree punya root A dengan anak B dan C; B punya anak D, E, F; C tidak punya anak. Tuliskan urutan PreOrder-plus-degree tree ini, dengan cara yang sama Bagian 11.7.2 lakukan untuk contoh 7-node A/B/C/D/E/F/G, dan nyatakan sum(degree).
6. Melanjutkan dari tree insert(65,5,70,3,10) setelah penghapusan 65 kasus (c) (menyisakan root=70, anak kiri=3), telusuri deleteKey(70) dengan tangan. Kasus mana dari ketiganya yang dimasuki penghapusan ini, dan seperti apa tree yang dihasilkan?

11.11 Lampiran 11.A — Log Validasi

Kelima program di bawah ini (`demo_insert.cpp`, `demo_traversals.cpp`, `demo_delete.cpp`, `demo_gtree.cpp`, `demo_all.cpp`) dikompilasi dengan `g++ -Wall -Wextra -std=c++17` (nol peringatan dari kelimanya) dan benar-benar dieksekusi; transkrip di bawah adalah keluaran terminal yang asli dan tidak diedit. `valgrind --leak-check=full --show-leak-kinds=all` sungguh dijalankan terhadap kelima program, sesuai kebijakan tetap mata kuliah ini sejak Bab 9.

11.A.1 demo_all.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_all demo_all.cpp
$ ./demo_all
=== Chapter 11 Validation Summary: Trees / BST ===

[Insert + shape]                                          5/5 checks OK
[Traversals]                                              4/4 checks OK
[Search]                                                  2/2 checks OK
[Delete -- three cases on the same running tree]          9/9 checks OK
[Delete -- supplementary deeper-successor check]          4/4 checks OK
[General tree <-> Binary conversion + reconstruction]     5/5 checks OK

Overall: 29 / 29 checks passed -- ALL CHECKS PASSED
```

11.A.2 demo_insert.cpp (kutipan — transkrip lengkap di _run_output.txt kode yang dikirim)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_insert demo_insert.cpp
$ ./demo_insert
count()   = 5  (expected 5)
height()  = 2  (expected 2 -- root to leaf 3 is 65->5->3, 2 edges)
findMin() = 3  (expected 3)
findMax() = 70  (expected 70)
findLeaves() = [ 3, 10, 70 ]  (expected [ 3, 10, 70 ])
PreOrder    = [ 65, 5, 3, 10, 70 ]
InOrder     = [ 3, 5, 10, 65, 70 ]  (sorted!)
PostOrder   = [ 3, 10, 5, 70, 65 ]
LevelOrder  = [ 65, 5, 70, 3, 10 ]
```

11.A.3 valgrind (kelima program)

```
$ valgrind --leak-check=full --show-leak-kinds=all ./demo_all
==17430==     in use at exit: 0 bytes in 0 blocks
==17430==   total heap usage: 131 allocs, 131 frees, 81,478 bytes allocated
==17430== All heap blocks were freed -- no leaks are possible
==17430== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Catatan kejujuran

Setiap hitungan, hasil traversal, dan baris terminal yang ditunjukkan di atas (di kelima program) berasal langsung dari kompilasi-dan-eksekusi nyata di lingkungan build bab ini. Pass validasi bab ini TIDAK menemukan bug dalam logika BST atau general tree itu sendiri -- diungkapkan secara jujur, sesuai kebijakan tetap mata kuliah ini, alih-alih memfabrikasi temuan yang tidak ada. valgrind dijalankan terhadap setiap satu dari kelima program dan setiap eksekusi melaporkan nol kebocoran dan nol error, mengonfirmasi bahwa destructor rekursif BST dan GNode/BinNode dengan benar membebaskan setiap node yang dialokasikan secara dinamis di seluruh insersi, penghapusan, dan konversi general tree. Zip kode yang dikirim di-ekstrak ulang dan dibangun ulang secara independen dari make clean && make run yang bersih untuk memastikan deliverable sesungguhnya, bukan hanya salinan kerja, dikompilasi dan dijalankan secara identik.

Referensi

[1] B.R. Preiss. “Data Structures and Algorithms with Object-Oriented Design Patterns in C++.” John Wiley & Sons, 1998. (Binary tree, BST, traversal.)

[2] M.A. Weiss. “Data Structures and Algorithm Analysis in C++.” 4th ed., Pearson, 2014. (Operasi BST, tree balancing.)

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Binary search tree, definisi height/depth tree.)

[4] P.J. Deitel, H.M. Deitel. “C++ How to Program.” 10th ed., Pearson, 2017. (Traversal tree, rekursi pada tree.)

[5] cppreference.com. “std::queue.” https://en.cppreference.com/w/cpp/container/queue (accessed August 2026).

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 12 · BAB LATIHAN

Kuis 2

Tidak ada materi baru di sini — ujian coding TIM, open-book, take-home yang menguji persis apa yang diajarkan Bab 11: tree dan Binary Search Tree (BST). Bab ini TIDAK menyertakan subfolder code/ — kesepuluh tugas di bawah adalah pekerjaan tim Anda sendiri untuk dirancang, ditulis, dan dikumpulkan.

ASESMEN TIM — baca ini sebelum apa pun yang lain

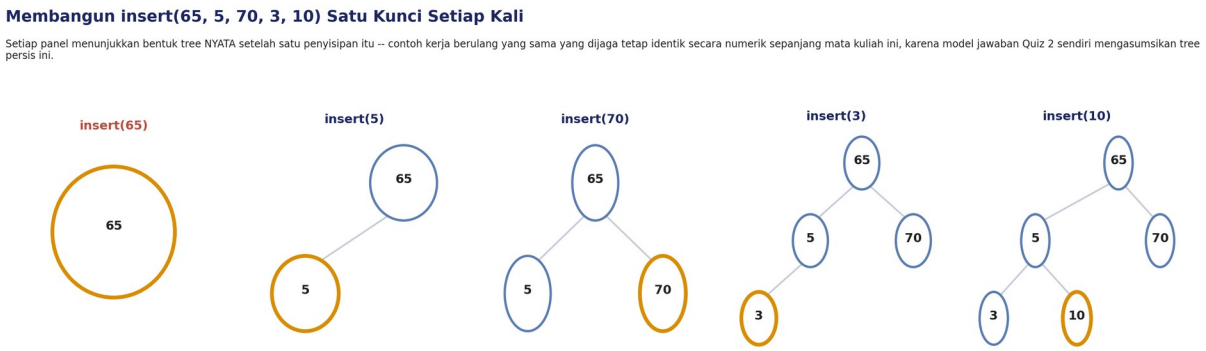
Kuis 2 secara eksplisit adalah tugas TIM: kelompok BERANGGOTA HINGGA 3 MAHASISWA mengumpulkan satu set solusi bersama. Ini BERBEDA dari Kuis 1 (Bab 4), UTS (Bab 8), dan UAS (Bab 16), yang semuanya bersifat INDIVIDUAL. Pastikan keanggotaan tim dan pembagian kerja tim Anda sudah jelas sebelum mulai mengerjakan Bagian 12.3 — setiap tugas di bawah mengasumsikan sebuah tim, bukan mahasiswa perorangan, yang menghasilkan jawabannya.

Tujuan Pembelajaran — setelah mengerjakan bab ini, tim Anda akan mampu:

(1) Mengenali bahwa kesepuluh tugas Kuis 2 di bawah semuanya bermuara pada bentuk rekursi BST Bab 11, sehingga tidak diperlukan teori tree baru, hanya penerapan yang benar pada sepuluh pertanyaan berbeda tentang tree yang sama. (2) Menulis cetak depth-first sebuah BST menggunakan salah satu dari tiga bentuk traversal DFS Bab 11, dan mencetak semua node pada level tertentu menggunakan penghitungan mundur rekursif. (3) Mengakumulasi total berjalan sepanjang traversal BST, dan memutasi nilai setiap node secara in place menggunakan bentuk traversal yang sama. (4) Mencerminkan (mirror) sebuah BST dengan menukar secara rekursif anak kiri dan kanan setiap node, dan menjelaskan mengapa InOrder tree hasil cerminan adalah kebalikan dari InOrder aslinya. (5) Menentukan apakah sebuah BST height-balanced menggunakan pemeriksaan satu-lintasan (single-pass) bottom-up yang efisien — bukan versi yang berulang kali memanggil height(). (6) Menerapkan deleteKey() Bab 11 (ketiga kasus: leaf, satu anak, dua anak) untuk menghapus node tertentu dari tree yang sedang berjalan. (7) Menulis pencari-parent dan pencari-anak atas sebuah BST, dengan benar membedakan "nilai tidak ditemukan" dari "nilai adalah root" atau "nilai adalah leaf". (8) Mengubah sebuah BST menjadi doubly linked list terurut secara in place dengan mengalihkan pointer left/right hasil traversal InOrder menjadi pointer prev/next — tanpa mengalokasikan satu pun node baru.

12.1 Ulasan: Bab 11 Secara Singkat

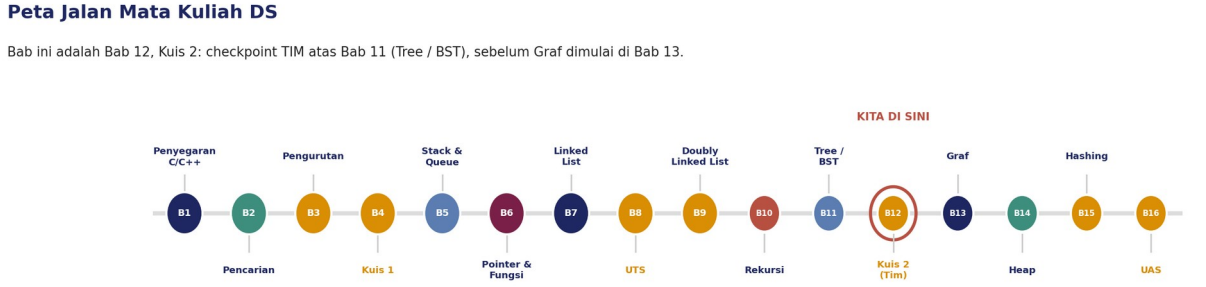
Bab 11 memperkenalkan struktur non-linear pertama dalam mata kuliah ini, yaitu tree, dan spesialisasi terpentingnya untuk pencarian cepat, Binary Search Tree (BST): subtree KIRI setiap node hanya berisi key yang lebih kecil, dan subtree KANAN setiap node hanya berisi key yang lebih besar, berlaku secara rekursif di setiap node. Bab itu membangun satu contoh numerik berulang, `insert(65, 5, 70, 3, 10)`, satu kunci pada satu waktu, dan membahas keempat traversal (PreOrder, InOrder, PostOrder, dan LevelOrder yang digerakkan queue NYATA), ditambah search(), count(), height(), findMin()/findMax(), pencari-leaf, konversi tree umum, dan — ditambahkan khusus karena materi mata kuliah asli meninggalkannya sebagai latihan — deleteKey(), yang mencakup ketiga kasus penghapusan.



Gambar 12.1 — contoh kerja berulang Bab 11 sendiri, insert(65, 5, 70, 3, 10), dibangun satu kunci setiap kali. Setiap tugas dalam bab ini diajukan atas tree persis ini.

Bentuk akhirnya, dinyatakan ulang di sini karena setiap tugas di Bagian 12.3 di bawah bergantung padanya:

```
        65
      /    \
     5      70
    / \
   3   10

count()  = 5     height()  = 2
PreOrder = [ 65, 5, 3, 10, 70 ]
InOrder  = [ 3, 5, 10, 65, 70 ]   (terurut)
PostOrder = [ 3, 10, 5, 70, 65 ]
```



Gambar 12.2 — Bab ini berada di Bab 12 dari enam belas bab peta jalan DS: sebuah checkpoint, bukan topik baru, Kuis 2 sebelum Bab 13 melanjutkan dengan Graf.

> **Tidak ada teori baru di bawah ini — hanya sepuluh pertanyaan baru tentang tree yang sudah Anda kenal**
>
> Setiap satu dari sepuluh tugas Kuis 2 (Bagian 12.3) menggunakan kembali bentuk traversal rekursif Bab 11 sendiri, struktur BSTNode-nya, atau deleteKey()-nya yang sudah diajarkan — diarahkan ke pertanyaan baru, bukan ide baru. Jika sebuah tugas terasa asing, itu adalah sinyal untuk membaca ulang bagian Bab 11 yang relevan (disebutkan pada callout setiap tugas di bawah), bukan tanda bahwa kuis ini mengharapkan materi yang belum pernah dibahas.

12.2 Cara Menggunakan Bab Latihan Ini

Apa bab ini — dan apa yang bukan

Ini adalah Bab Latihan: bab praktik, bukan bab materi baru. Kuis 2, seperti asesmen DS sesungguhnya yang menjadi modelnya, adalah ujian coding TIM, open-book, take-home — kelompok beranggota hingga 3 mahasiswa mengumpulkan bersama, berbeda dari Kuis 1, UTS, dan UAS yang individual. Bagian 12.3 di bawah menjabarkan kesepuluh tugas Kuis 2, masing-masing disertai callout panduan (THINK, TRAP, atau INSIGHT), bukan solusi lengkap yang sudah dikerjakan. Seperti bab latihan lain dalam buku ini, bab ini TIDAK menyertakan subfolder code/ — program yang ditulis tim Anda untuk Kuis 2 adalah pekerjaan tim Anda sendiri, dikompilasi dan dijalankan di komputer Anda sendiri sebelum dikumpulkan.

Sebelum tim Anda mulai

Bagikan kesepuluh tugas di bawah (atau kelompok kecil dari tugas-tugas itu) kepada anggota tim tertentu sebelum menulis kode apa pun, dan sepakati sebagai tim SATU struktur BSTNode dan implementasi insert() bersama — idealnya milik Bab 11 sendiri — sehingga setiap tugas beroperasi pada tree yang identik. Tim yang menciptakan ulang struktur node sepuluh kali terpisah, sekali per tugas, akan menghabiskan jauh lebih banyak waktu untuk merekonsiliasi kode yang tidak cocok dibandingkan tim yang menyepakati satu header bersama terlebih dahulu.

12.3 Kumpulan Soal Kuis 2

Kuis 2 terdiri atas sepuluh tugas coding independen, masing-masing bernilai 10 poin, total 100 poin.[1] Setiap tugas di bawah meminta sebuah fungsi yang beroperasi pada tree insert(65, 5, 70, 3, 10) yang sama yang dinyatakan ulang di Bagian 12.1 di atas: baca (atau hard-code, jika instruksi mata kuliah Anda sendiri tidak menyebutkan file input) masukan yang dideskripsikan, hasilkan keluaran yang dideskripsikan, dan kompilasi dengan bersih menggunakan `g++ -Wall -Wextra -std=c++17`, toolchain yang sama yang digunakan setiap bab dalam buku ini sejak Bab 1.

Tugas independen, tree awal independen

Kecuali harness pengujian tim Anda sendiri sengaja merangkai tugas-tugas tersebut demi kepraktisan, perlakukan setiap satu dari sepuluh tugas di bawah sebagai beroperasi pada tree insert(65, 5, 70, 3, 10) yang baru dibangun MASING-MASING — Tugas 4 (tigakan setiap nilai) dan Tugas 5 (mirror) sama-sama MEMUTASI tree, dan Tugas 7 (hapus) menghilangkan sebuah node darinya, sehingga tidak satu pun dari Tugas 1, 2, 3, 6, 8, 9, atau 10 seharusnya melihat tree yang sudah diubah oleh keluaran tugas sebelumnya kecuali tugas itu menyatakan sebaliknya secara eksplisit.

Asal Setiap Tugas Kuis 2

Kuis 2 terdiri atas sepuluh tugas coding yang dinilai secara tim, semuanya diajukan atas tree insert(65,5,70,3,10) yang SAMA yang dibangun Bab 11 -- setiap tugas di bawah dapat ditelusuri kembali ke ide spesifik Bab 11.

#	Tugas Kuis 2	Berasal dari	Poin
1	Cetak tree secara depth-first	Bab 11 -- PreOrder/InOrder/PostOrder: bentuk rekursif base-case/recursive-case	10
2	Cetak semua node pada level ke-k	Bab 11 -- ide LevelOrder/height(), dibingkai ulang sebagai turunan rekursif yang menghitung mundur k	10
3	Hitung jumlah seluruh nilai node	Bab 11 -- rekursi traversal, digeneralisasi dari count() menjadi mengakumulasi nilai key	10
4	Tigakan (triple) setiap nilai node, in place	Bab 11 -- rekursi traversal diterapkan sebagai mutasi in-place pada setiap node yang dikunjungi	10
5	Cerminkan (mirror) tree (tukar left/right di mana-mana)	Bab 11 -- pointer left/right BSTNode, ditukar secara rekursif di setiap node	10
6	Periksa apakah tree height-balanced	Bab 11 -- height(), diperluas menjadi pemeriksaan gaya-AVL |height(L)-height(R)| <= 1	10
7	Hapus sebuah node tertentu dari tree	Bab 11 -- deleteKey(): ketiga kasus penghapusan (leaf, satu anak, dua anak)	10
8	Cari parent dari sebuah nilai anak tertentu	Bab 11 -- search() BST, diperluas untuk mengingat parent yang terlihat tepat sebelum kecocokan	10
9	Cari anak-anak dari sebuah nilai parent tertentu	Bab 11 -- search() BST, membaca langsung left/right anak dari node yang ditemukan	10
10	Ubah BST menjadi doubly linked list terurut, in place	Bab 11 -- traversal InOrder (sudah menghasilkan output terurut); left/right dipakai ulang sebagai prev/next	10

Total: 100 poin

Gambar 12.3 — Setiap tugas di bagian ini dapat ditelusuri kembali ke ide spesifik yang sudah dibahas di Bab 11.

12.3.1 Tugas 1 — Cetak Tree Secara Depth-First (10 poin)

Tulis sebuah fungsi yang mencetak key setiap node menggunakan traversal DEPTH-FIRST — salah satu dari tiga bentuk DFS Bab 11 (PreOrder, InOrder, atau PostOrder) dapat diterima, selama tim Anda menyatakan dengan jelas dalam keluaran atau komentar traversal mana yang digunakan. Diterapkan pada tree yang berjalan, cetak PreOrder menghasilkan `65 5 3 10 70`; cetak InOrder menghasilkan urutan terurut `3 5 10 65 70`.

Petunjuk

Ini adalah Bagian 11.5 Bab 11, tidak berubah — sebuah fungsi rekursif kecil dengan base case subtree kosong dan dua pemanggilan rekursif, hanya berbeda pada KAPAN key node saat ini dikunjungi relatif terhadap dua pemanggilan itu. Gunakan kembali bentuk yang persis sama (`if (!node) return;` lalu kunjungi/rekursi/rekursi dalam urutan mana pun yang Anda pilih) alih-alih menciptakan gaya traversal baru.

Mencetak selama traversal tidak sama dengan membangun sebuah traversal

Fungsi yang mencetak langsung (misalnya, `std::cout << node->key`) sambil melakukan rekursi mudah ditulis tetapi lebih sulit diuji secara otomatis, karena keluarannya terikat pada format konsol. Pertimbangkan sebaliknya membangun `std::vector<int>` berisi key yang dikunjungi (pendekatan Bab 11 sendiri) dan mencetak vektor itu setelahnya — logika traversalnya kemudian identik baik tim Anda mencetak ke terminal maupun membandingkannya dengan urutan yang diharapkan dalam pengujian otomatis.

12.3.2 Tugas 2 — Cetak Semua Node pada Level ke-k (10 poin)

Diberikan sebuah bilangan bulat k, cetak key setiap node pada level k dari tree, di mana root adalah level 0. Diterapkan pada tree yang berjalan: level 0 mencetak `65`; level 1 mencetak `5 70`; level 2 mencetak `3 10`; level k >= 3 mana pun (tinggi tree adalah 2, sehingga tidak ada node pada level 3 atau lebih dalam) mencetak kosong, bukan error.

Petunjuk — turun secara rekursif, menghitung k menuju nol

Bentuk rekursif langsung: `printLevel(node, k)` — base case, `node == nullptr`, kembali; base case kedua, `k == 0`, kunjungi `node` dan kembali; kasus rekursif, `k > 0`, panggil `printLevel(node->left, k - 1)` lalu `printLevel(node->right, k - 1)`. Setiap pemanggilan rekursif turun satu level dan mengurangi k tepat satu, sehingga fungsi secara alami berhenti pada kedalaman yang tepat tanpa perlu melacak kedalaman absolut dari root secara terpisah.

Off-by-one pada apakah root adalah level 0 atau level 1 adalah bug klasik di sini

Putuskan, dan nyatakan secara eksplisit dalam format keluaran tim Anda sendiri, apakah root dihitung sebagai level 0 (konvensi bagian ini, sesuai konvensi height() Bab 11 di mana tree satu-node memiliki height 0) atau level 1 — dan terapkan konvensi itu secara konsisten pada pemanggilan uji k=0, k=1, dan k=2 Anda. k yang diam-diam off-by-one akan tetap berjalan tanpa crash; ia hanya akan diam-diam mencetak node level yang SALAH, yang mudah terlewat tanpa memeriksanya secara langsung terhadap diagram tree pada Gambar 12.1.

12.3.3 Tugas 3 — Hitung Jumlah Seluruh Nilai Node (10 poin)

Tulis sebuah fungsi yang mengembalikan jumlah seluruh key node dalam tree. Diterapkan pada tree yang berjalan: 65 + 5 + 70 + 3 + 10 = 153.

Petunjuk — ini adalah count() dengan penjumlahan alih-alih penghitungan

Fungsi count() Bab 11 (Bagian 11.6) sudah menunjukkan bentuknya: subtree kosong menyumbang 0, dan node mana pun lainnya menyumbang 1 ditambah jumlah kedua subtree-nya. sumNodes() identik kecuali node yang tidak kosong menyumbang KEY-nya SENDIRI ditambah jumlah kedua subtree-nya, alih-alih menyumbang 1: `return node->key + sumNodes(node->left) + sumNodes(node->right);`

Akumulator yang dilewatkan by reference harus diinisialisasi tepat satu kali, sebelum traversal dimulai

Jika solusi tim Anda sebaliknya mengalirkan total berjalan melalui parameter keluaran (`void sumNodes(BSTNode* node, int& total)`) alih-alih mengembalikan nilai, `total` harus diinisialisasi ke 0 oleh PEMANGGIL sebelum pemanggilan pertama — akumulator yang hanya di-reset pada sebagian jalur kode, atau dibiarkan menyimpan nilai basi dari kasus uji sebelumnya, akan diam-diam menghasilkan jumlah yang tampak masuk akal tetapi salah. Versi kembalikan-sebuah-nilai di atas menghindari seluruh kelas bug ini dan merupakan bentuk yang direkomendasikan untuk tugas ini.

12.3.4 Tugas 4 — Tigakan (Triple) Setiap Nilai Node, In Place (10 poin)

Tulis sebuah fungsi yang mengalikan key setiap node dengan 3, secara permanen, pada tree itu sendiri — bukan pada salinan yang dicetak atau struktur keluaran terpisah. Diterapkan pada tree yang berjalan: 65, 5, 70, 3, 10 menjadi 195, 15, 210, 9, 30, dan InOrder tree setelahnya adalah `9 15 30 195 210`.

Petunjuk — mutasi melalui pointer node yang sesungguhnya, bukan salinan

Bentuk traversalnya tidak berubah dari Tugas 1 atau Tugas 3 — kunjungi setiap node tepat sekali — tetapi kali ini langkah kunjungannya adalah `node->key *= 3;` diterapkan langsung pada `BSTNode` sesungguhnya yang pointer-nya sedang dipegang rekursi, bukan pada salinan key-nya yang ditarik ke variabel lokal. Jika parameter fungsi Anda adalah `BSTNode* node` (sebuah pointer, sesuai signature Bab 11 sendiri), `node->key *= 3` dengan benar menjangkau tree sesungguhnya; parameter yang tidak sengaja diambil by value sebagai `BSTNode node` — kesalahan tipe yang biasanya akan ditangkap compiler, karena BSTNode umumnya tidak dapat disalin dalam rancangan mata kuliah ini — tidak akan.

Menigakan tidak pernah merusak properti BST, dan ini dapat dibuktikan, bukan sekadar diamati

Mengalikan setiap key dengan konstanta positif yang sama mempertahankan setiap hubungan `<` dan `>` yang dimiliki key aslinya, sehingga tree yang valid sebagai BST sebelum ditigakan TETAP valid sebagai BST setelahnya — tidak diperlukan rebalancing atau restrukturisasi, hanya mutasi key in-place itu sendiri. Inilah sebabnya tugas ini dapat diselesaikan sebagai traversal-murni-dengan-efek-samping alih-alih memerlukan langkah bangun-ulang-dari-nol.

12.3.5 Tugas 5 — Cerminkan (Mirror) Tree (Tukar Subtree Kiri/Kanan Setiap Node) (10 poin)

Tulis sebuah fungsi yang secara rekursif menukar pointer anak kiri dan kanan setiap node di seluruh tree, menghasilkan bayangan cerminnya. Diterapkan pada tree yang berjalan, bentuk hasil mirror memiliki 65 sebagai root dengan 70 kini berada di KIRI dan 5 kini berada di KANAN (dengan anak-anak 5 sendiri, 3 dan 10, juga ditukar, sehingga 10 kini di kiri 5 dan 3 di kanan 5).

Petunjuk — satu std::swap per node, diterapkan secara rekursif

`mirror(node)`: base case, `node == nullptr`, kembali; kasus rekursif, `std::swap(node->left, node->right);` lalu rekursi ke KEDUA anak node (yang kini sudah ditukar): `mirror(node->left); mirror(node->right);`. Penukaran harus terjadi SEBELUM kedua pemanggilan rekursif (atau sesudahnya — kedua urutan sama-sama benar di sini, karena pemanggilan rekursif mengikuti pointer mana pun yang saat ini berada di `node->left`/`node->right`, dan penukaran hanya menukar anak yang mana yang mana, bukan node mana yang ada).

InOrder tree hasil mirror persis sama dengan InOrder aslinya, dibalik

Karena InOrder mengunjungi kiri-root-kanan, dan mirroring menukar subtree kiri dan kanan setiap node, mirroring tree yang berjalan mengubah InOrder-nya dari `[3, 5, 10, 65, 70]` menjadi `[70, 65, 10, 5, 3]` — LIMA key yang SAMA, dalam urutan yang persis terbalik. Mengonfirmasi ini di atas kertas (atau dengan benar-benar menjalankan InOrder sebelum dan sesudah pemanggilan mirror() Anda) adalah cara cepat dan andal untuk memeriksa mirror() Anda tanpa harus menggambar ulang seluruh tree dengan tangan. Perhatikan bahwa tree hasil mirror bukan lagi BST dalam pengertian menaik-kiri-ke-kanan yang biasa — ia kini menjadi binary tree umum berbentuk bayangan cermin sebuah BST, yang persis diminta tugas ini, bukan sebuah bug.

12.3.6 Tugas 6 — Periksa Apakah Tree Height-Balanced (10 poin)

Tulis sebuah fungsi yang mengembalikan true jika tree HEIGHT-BALANCED — artinya, pada SETIAP node dalam tree (bukan hanya root), height subtree kiri dan kanannya berbeda paling banyak 1 — dan false jika sebaliknya. Diterapkan pada tree yang berjalan: root (65) memiliki subtree kiri dengan height 1 (subtree 3-node berakar di 5) dan subtree kanan dengan height 0 (leaf tunggal 70), selisih 1, yang masih dalam batas yang diizinkan; node 5 memiliki subtree kiri dengan height 0 (leaf 3) dan subtree kanan dengan height 0 (leaf 10), selisih 0. Setiap node memenuhi kondisi tersebut, sehingga tree yang berjalan ADALAH height-balanced.

Pendekatan naif menghitung ulang height() di setiap node dan bersifat $O(n^2)$, bukan O(n)

Draf pertama yang menggoda memanggil `height(node->left)` dan `height(node->right)` secara terpisah di setiap node sambil juga melakukan rekursi ke kedua anaknya untuk memeriksa keseimbangan MEREKA — ini menghitung ulang height subtree yang sama berkali-kali, menghasilkan total kerja $O(n^2)$ pada kasus terburuk (tree tinggi dan miring/skewed), persis jebakan laju pertumbuhan yang sudah diperingatkan Bab 2 dan tugas-tugas bergaya Kuis di bagian lain mata kuliah ini.

Satu lintasan bottom-up tunggal mendapatkan baik height MAUPUN pemeriksaan balance dalam O(n)

Versi efisiennya memiliki SATU fungsi rekursif yang mengembalikan height sebuah subtree SEKALIGUS memeriksa balance saat berjalan, menggunakan sebuah sentinel (umumnya -1, atau nilai mana pun yang tidak mungkin menjadi height sesungguhnya) untuk menandakan "sudah ditemukan tidak seimbang, hentikan pemeriksaan lebih lanjut ke atas": pada setiap node, secara rekursif dapatkan height-atau-sentinel subtree kiri dan kanan; jika salah satu anak sudah melaporkan sentinel, atau jika kedua height berbeda lebih dari 1, teruskan sentinel ke atas segera alih-alih melanjutkan menghitung height sesungguhnya. Ini mengunjungi setiap node tepat sekali — total O(n) — ide "lakukan dalam satu lintasan alih-alih berkali-kali" yang sama yang diterapkan Kuis 1 Bab 4 untuk menemukan karakter berulang pertama.

12.3.7 Tugas 7 — Hapus Sebuah Node Tertentu dari Tree (10 poin)

Diberikan sebuah key, hapus node tersebut dari tree sambil mempertahankan properti BST, menggunakan deleteKey() Bab 11 (Bagian 11.8): sebuah LEAF diputus langsung; node dengan SATU

anak memiliki anak tersebut disambungkan ke tempatnya; node dengan DUA anak memiliki key-nya ditimpa oleh key successor in-order-nya, dan successor tersebut kemudian dihapus dari subtree kanan (yang, karena berasal dari pencarian successor, dijamin memiliki paling banyak satu anak sendiri). Diterapkan pada tree yang berjalan, menghapus key 5 (node satu-anak — satu-satunya anak 5, yaitu 3, disambungkan ke tempat 5) menyisakan tree `[65 [3, -], [70, -]]`, dengan InOrder `3 65 70`.

Ini adalah ulasan langsung, bukan materi baru — tetapi pelajari kembali dengan cermat

Bagian 11.8 Bab 11 sudah menjabarkan ketiga kasus penghapusan pada tree persis ini, lengkap dengan diagram penuh (Gambar 11.3) yang menelusuri penghapusan leaf, penghapusan satu-anak, dan penghapusan dua-anak secara berurutan. Baca ulang bagian itu sebelum menulis deleteKey() dari ingatan — langkah successor in-order untuk kasus dua-anak (temukan MINIMUM subtree kanan, salin key-nya ke atas, lalu hapus node successor tersebut dari subtree kanan) adalah langkah yang paling sering salah diingat.

Menghapus sebuah node harus mengembalikan root subtree (yang mungkin baru) kepada pemanggilnya

Karena node yang benar-benar dihapus dari memori terkadang adalah ROOT dari subtree yang diberikan (misalnya, menghapus node leaf atau satu-anak), `deleteKey()` sebaiknya ditulis untuk MENGEMBALIKAN sebuah `BSTNode*` — root subtree setelah penghapusan — yang kemudian ditugaskan kembali oleh pemanggil ke pointer yang sebelumnya memegang subtree lama: `node->left = deleteKey(node->left, key);`. Versi yang mencoba menghapus "di tempat" tanpa mengembalikan dan menugaskan ulang root subtree (yang mungkin berubah) berisiko meninggalkan pointer menggantung ke memori yang sudah dibebaskan, atau meninggalkan parent yang masih menunjuk ke node yang tidak lagi mewakili subtree itu dengan benar.

12.3.8 Tugas 8 — Cari Parent dari Sebuah Nilai Anak Tertentu (10 poin)

Diberikan sebuah key yang diketahui ada di suatu tempat dalam tree (selain root), temukan dan laporkan KEY dari node parent-nya. Diterapkan pada tree yang berjalan: parent dari 3 adalah 5; parent dari 10 adalah 5; parent dari 5 adalah 65; parent dari 70 adalah 65. Root, 65, tidak memiliki parent — fungsi Anda harus melaporkan ini secara berbeda, tidak boleh dicampuradukkan dengan "nilai tidak ditemukan".

Petunjuk — lacak node satu langkah di belakang penelusuran pencarian Anda

Ini memperluas search() Bab 11 (Bagian 11.6.1) dengan tepat satu bagian state tambahan: saat Anda turun membandingkan key target terhadap key setiap node untuk memutuskan arah mana yang dituju, simpan pointer kedua, `parent`, yang diperbarui ke node SAAT INI tepat sebelum Anda berpindah ke `node->left` atau `node->right`. Ketika penelusuran menemukan node yang key-nya cocok dengan target, `parent` sudah memegang node satu langkah di atasnya — tidak diperlukan traversal kedua yang terpisah.

Tiga hasil yang benar-benar berbeda, dan rancangan tim Anda tidak boleh menyatukannya menjadi dua

Tugas ini memiliki TIGA kemungkinan hasil, bukan dua: (a) key ada dan memiliki parent — laporkan key parent-nya; (b) key ada tetapi ADALAH root — laporkan secara eksplisit bahwa ia tidak memiliki parent, berbeda dari "tidak ditemukan"; (c) key sama sekali tidak ada dalam tree — laporkan "tidak ditemukan". Rancangan yang mengembalikan, katakanlah, satu `int` dengan `-1` berarti baik "adalah root" maupun "tidak ditemukan" membuat kedua situasi yang benar-benar berbeda ini tidak dapat dibedakan oleh apa pun yang memanggilnya — lebih baik gunakan tipe hasil kecil (`std::optional<int>` ditambah flag-ditemukan terpisah, atau sebuah struct dengan tag eksplisit `{FOUND_WITH_PARENT, FOUND_IS_ROOT, NOT_FOUND}`) yang menjaga ketiga kasus tetap terpisah.

12.3.9 Tugas 9 — Cari Anak-Anak dari Sebuah Nilai Parent Tertentu (10 poin)

Diberikan sebuah key yang diketahui ada di suatu tempat dalam tree, laporkan key anak KIRI-nya dan key anak KANAN-nya, jika masing-masing ada. Diterapkan pada tree yang berjalan: anak-anak dari 65 adalah 5 (kiri) dan 70 (kanan); anak-anak dari 5 adalah 3 (kiri) dan 10 (kanan); anak-anak dari 70 tidak ada (ia adalah leaf); anak-anak dari 3 dan 10 juga tidak ada.

Petunjuk — ini adalah search() ditambah pembacaan field langsung, tidak lebih

Gunakan kembali penelusuran bergaya search() Bab 11 untuk MENEMUKAN node yang menyimpan key yang diberikan (membandingkan terhadap `node->key` pada setiap langkah, pergi ke kiri atau kanan sesuai, persis seperti yang dilakukan Bagian 11.6.1), dan setelah ditemukan, cukup baca `node->left` dan `node->right` secara langsung — jika sebuah pointer anak adalah `nullptr`, sisi itu tidak memiliki anak, yang layak dilaporkan secara eksplisit ("tidak ada anak kiri" / "tidak ada anak kanan" / "leaf, tidak ada anak") alih-alih diam-diam tidak mencetak apa pun.

Key yang sama sekali tidak ada dalam tree adalah kasus berbeda dari key yang ada tetapi tanpa anak

Sama seperti Tugas 8 memperingatkan agar tidak menyatukan "adalah root" ke dalam "tidak ditemukan", rancangan tugas ini tidak boleh menyatukan "key ada tetapi merupakan leaf (nol anak)" ke dalam "key tidak ditemukan" — ini adalah hasil berbeda yang harus dilaporkan fungsi tim Anda secara terpisah. Uji solusi Anda secara eksplisit terhadap sebuah leaf (key 3, 10, atau 70), node dengan satu anak yang tidak ada pada tree khusus ini tetapi akan memerlukan kehati-hatian yang sama jika tim Anda menguji tree lain, dan sebuah key yang benar-benar tidak ada dalam tree (misalnya, 999) untuk mengonfirmasi ketiganya terbaca sebagai hasil yang berbeda.

12.3.10 Tugas 10 — Ubah BST Menjadi Doubly Linked List Terurut Secara In Place (10 poin)

Ubah tree menjadi DOUBLY LINKED LIST terurut, menggunakan kembali pointer `left` dan `right` node yang SUDAH ADA sebagai `prev` dan `next` masing-masing — jangan mengalokasikan SATU PUN node baru. Hasilnya, diterapkan pada tree yang berjalan, adalah rantai `3 <-> 5 <-> 10 <-> 65 <-> 70`, dalam

urutan menaik, di mana HEAD list adalah node yang sama yang dulu menyimpan key 3 dan TAIL list adalah node yang sama yang dulu menyimpan key 70.

Tidak ada yang baru secara teori — sebuah traversal familiar, dialihkan

Bab 11 sudah menetapkan bahwa traversal InOrder mengunjungi key BST dalam urutan TERURUT (Bagian 11.5.2, dikonfirmasi secara terprogram, bukan sekadar diamati pada satu contoh). Tugas ini tidak memerlukan teori baru di luar fakta itu dan bentuk rekursi traversal mata kuliah ini — ia hanya meminta tim Anda mengubah APA yang dilakukan traversal pada setiap kunjungan: alih-alih menambahkan sebuah key ke vektor keluaran, ia menyambungkan dua node SESUNGGUHNYA (node yang dikunjungi sebelumnya dan node saat ini) menjadi sebuah rantai, menggunakan kembali field left/right mereka yang sudah ada sebagai prev/next.

Petunjuk — bawa pointer "node yang baru dikunjungi" sepanjang rekursi InOrder

Pendekatan standar mengalirkan satu bagian state tambahan melalui rekursi berbentuk InOrder: sebuah pointer `prev` (atau reference), diinisialisasi ke `nullptr` sebelum traversal dimulai. Pada setiap node, SETELAH melakukan rekursi ke `node->left` (yang menyelesaikan konversi semua yang lebih kecil), lakukan: jika `prev` bukan `nullptr`, atur `prev->right = node` dan `node->left = prev` (penyambungan); lalu atur `prev = node` sebelum melakukan rekursi ke `node->right`. Node pertama yang dikunjungi (key terkecil, 3) tidak pernah menerima tautan `left`, dengan benar menjadi head list; node terakhir yang dikunjungi (key terbesar, 70) tidak pernah menerima tautan `right` dari proses ini, dengan benar menjadi tail list.

Mengalokasikan bahkan satu node baru di mana pun mengalahkan seluruh maksud tugas ini

Menggoda untuk membangun doubly linked list baru yang biasa berisi node bergaya `Book` atau berisi `int` dengan melakukan traversal BST dan memanggil `new` untuk setiap nilai — ini AKAN menghasilkan doubly linked list terurut, tetapi bukan yang diminta tugas ini, dan biasanya tidak akan mendapat nilai penuh, karena tugas ini secara khusus mensyaratkan penggunaan kembali node-node tree yang SUDAH ADA, bukan menyalin nilainya ke memori baru. Jika solusi tim Anda tetap berfungsi meskipun node-node tree asli adalah objek buram yang tidak dapat disalin (hanya dapat disambungkan ulang, tidak pernah diduplikasi), Anda menyelesaikan tugas yang dimaksud; jika ia perlu menyalin sebuah field `key` ke node yang benar-benar baru, maka tidak.

12.4 Format dan Penilaian Kuis 2

Kuis 2 adalah ujian coding TIM, open-book, take-home — kelompok BERANGGOTA HINGGA 3 MAHASISWA mengumpulkan satu set solusi bersama. Ini berbeda dari Kuis 1 (Bab 4), UTS (Bab 8), dan UAS (Bab 16), yang masing-masing bersifat INDIVIDUAL. "Open-book" berarti buku teks, catatan tim Anda sendiri, dan materi referensi C++ standar (seperti cppreference.com) semuanya sah digunakan selama Anda bekerja; ini TIDAK berarti menyalin kode tim lain atau mengumpulkan pekerjaan yang bukan benar-benar milik tim Anda sendiri — ujian tim yang open-book tetap menguji apakah TIM ANDA dapat menerapkan apa yang telah diajarkan mata kuliah ini, bekerja bersama alih-alih sendiri-sendiri.

Rubrik poin-per-subtugas milik DS sendiri

Seperti setiap bab latihan lain dalam buku ini, Kuis 2 dinilai tugas demi tugas, dalam poin bulat, bukan dengan template Design/Implementation/Analysis/Conclusion berbobot persentase. Buku teks ini mengalokasikan kesepuluh tugas dengan rata 10 poin masing-masing (total 100)[1], diberikan untuk kode yang terkompilasi dengan bersih dan menghasilkan keluaran yang benar untuk persyaratan yang dinyatakan (termasuk, untuk Tugas 4, benar-benar memutasi node tree itu sendiri alih-alih mencetak salinan yang ditigakan, untuk Tugas 6, pemeriksaan yang benar mengevaluasi kondisi balance di setiap node bukan hanya root, dan untuk Tugas 10, benar-benar menggunakan kembali node tree sendiri alih-alih mengalokasikan yang baru). Tidak ada komponen "analisis" atau "kesimpulan" terpisah pada kuis ini — kode dan keluarannya yang benar adalah deliverable-nya.

#	Tugas	Apa yang diuji	Poin
1	Cetak depth-first	Traversal DFS yang benar (salah satu dari PreOrder/InOrder/PostOrder), dilabeli dengan jelas	10
2	Cetak node pada level k	Turunan rekursif yang benar berhenti tepat di level k, root dihitung sebagai level 0	10
3	Jumlah seluruh nilai node	Akumulasi yang benar di setiap node, sesuai bentuk traversal count()	10
4	Tigakan setiap nilai in place	Mutasi in-place yang benar pada node tree sendiri, bukan salinan tercetak	10
5	Cerminkan (mirror) tree	Penukaran kiri/kanan rekursif yang benar di setiap node, membalik hasil InOrder	10
6	Pemeriksaan height-balanced	Pemeriksaan balance yang benar di SETIAP node, idealnya dalam satu lintasan bottom-up O(n)	10
7	Hapus sebuah node tertentu	Penerapan yang benar atas ketiga kasus deleteKey(), dengan root subtree dikembalikan/ditugaskan ulang	10
8	Cari parent dari sebuah nilai	Membedakan dengan benar antara punya-parent, adalah-root, dan tidak-ditemukan	10
9	Cari anak-anak dari sebuah nilai	Membedakan dengan benar antara punya-anak, adalah-leaf, dan tidak-ditemukan	10
10	BST ke DLL terurut in place	Penyambungan yang benar didorong InOrder yang menggunakan kembali node tree sendiri, nol alokasi baru	10
	Total		100

[1] *Catatan kaki rubrik atas pembagian rata ini: materi ajar asli di balik riset buku ini mengonfirmasi kesepuluh tugas Kuis 2 dan total 100 poinnya, serta mengonfirmasi bentuk penilaian poin-per-subtugasnya (sesuai setiap bab latihan DS lainnya, dan berbeda dari pembagian rata 25-poin-per-tugas Kuis 1 sendiri atau pembagian 10/30/30/15/15 UTS yang sengaja tidak rata) — tetapi tidak mempertahankan rincian poin per tugas yang tidak ambigu di seluruh sepuluh tugas dalam materi yang menjadi dasar penyusunan buku teks ini. Alih-alih menebak-nebak pembagian tidak rata tanpa sumber yang andal, buku teks ini mengalokasikan kesepuluh tugas dengan 10 poin yang sama masing-masing, berjumlah total 100 poin yang sama dengan yang digunakan Kuis 2 sesungguhnya. Jika*

instruksi mata kuliah Anda sendiri menetapkan rincian per tugas yang berbeda, ikuti instruksi Anda sendiri di atas catatan kaki ini — hanya REDAKSI TUGAS dan TREE di atas yang dimaksudkan otoritatif di sini.

Kumpulkan kesepuluh fungsi sebagai satu pengumpulan yang koheren (baik satu program yang menjalankan kesepuluhnya secara berurutan, atau sepuluh berkas kecil yang dapat dikompilasi secara individual berbagi satu header `BSTNode`/tree — ikuti instruksi pengumpulan mata kuliah Anda sendiri), masing-masing menunjukkan keluarannya sendiri yang benar saat dijalankan pada tree insert(65, 5, 70, 3, 10) — program yang tidak terkompilasi tidak mendapat nilai untuk tugas-tugas yang dikandungnya, tidak peduli seberapa dekat logikanya dengan yang benar, jadi sisakan waktu bagi SELURUH TIM ANDA untuk benar-benar membangun dan menjalankan setiap tugas sebelum mengumpulkan, persis disiplin yang dipegang setiap contoh kode di Bab 1 hingga 11.

12.5 Ringkasan Bab

- Kuis 2 tidak memperkenalkan materi baru — ia adalah checkpoint atas Bab 11 (Tree / BST), diulas di Bagian 12.1, diajukan sebagai sepuluh pertanyaan terhadap tree insert(65, 5, 70, 3, 10) yang SAMA sepanjang bab ini.
- Ini adalah ujian coding TIM, open-book, take-home — kelompok BERANGGOTA HINGGA 3 MAHASISWA, berbeda dari Kuis 1, UTS, dan UAS yang individual.
- Sepuluh tugas bernilai 10 poin masing-masing (total 100): cetak depth-first, cetak node pada level k, jumlahkan seluruh nilai, tigakan setiap nilai in place, cerminkan tree, periksa height-balance, hapus sebuah node tertentu, cari parent sebuah nilai, cari anak-anak sebuah nilai, dan ubah BST menjadi doubly linked list terurut secara in place.
- Penilaian mengikuti rubrik poin-per-subtugas milik DS sendiri — pembagian rata 10-poin-per-tugas buku ini adalah pilihan yang didokumentasikan secara transparan (lihat catatan kaki rubrik Bagian 12.4), bukan klaim rincian poin asli yang tepat yang tidak dapat dikonfirmasi secara independen oleh proyek ini.
- Tugas 6 (height-balance) dan Tugas 7 (hapus) adalah dua tugas yang paling layak mendapat perhatian ekstra: solusi naif Tugas 6 bersifat $O(n^2)$ alih-alih $O(n)$, dan Tugas 7 harus dengan benar mengembalikan dan menugaskan ulang root subtree alih-alih menghapus "di tempat" tanpa melakukannya.
- Bab ini TIDAK menyertakan subfolder code/: kesepuluh tugas adalah pekerjaan kolektif tim Anda sendiri, untuk dikompilasi dengan toolchain g++ -Wall -Wextra -std=c++17 yang sama yang digunakan sepanjang buku ini dan benar-benar dijalankan sebelum dikumpulkan.

Program yang terkompilasi tidak sama dengan program yang benar — dan program yang benar pada satu tree yang Anda coba tidak sama dengan yang benar secara umum. Sebagai tim, uji kesepuluh solusi Anda melampaui contoh yang berjalan bila memungkinkan, termasuk setidaknya satu kasus yang sengaja janggal per tugas (level k yang lebih dalam dari height tree untuk Tugas 2, key target yang merupakan root untuk Tugas 8, key target yang merupakan leaf untuk Tugas 9, dan pemeriksaan bolak-balik berulang — bangun DLL-nya, lalu konfirmasi menelusurinya maju dan mundur sama-sama menghasilkan urutan terurut — untuk Tugas 10) sebelum Anda mengumpulkan.

Lampiran 12.A — Daftar Periksa Mandiri (Tidak Dinilai)

Sebelum mengumpulkan sepuluh solusi tim Anda, jalankan bersama melalui daftar periksa ini. Ini memakan waktu beberapa menit dan menangkap sebagian besar kesalahan yang paling sering dilihat penilai pada pengumpulan Kuis 2 pertama.

- Apakah kode setiap anggota tim terkompilasi dengan bersih menggunakan `g++ -Wall -Wextra -std=c++17`, tanpa peringatan, setelah digabungkan menjadi satu pengumpulan?
- Tugas 1: apakah cetak DFS Anda menyatakan dengan jelas traversal mana dari PreOrder/InOrder/PostOrder yang digunakan?
- Tugas 2: apakah level k = 0 dengan benar mencetak hanya root, dan apakah level yang lebih dalam dari height tree dengan benar mencetak kosong (bukan crash)?
- Tugas 3: apakah jumlah Anda menggunakan nilai yang DIKEMBALIKAN dari rekursi, atau jika menggunakan akumulator by reference, apakah akumulator itu diinisialisasi ke 0 tepat satu kali oleh pemanggil?
- Tugas 4: setelah ditigakan, apakah traversal InOrder pada objek tree yang SAMA menunjukkan 9, 15, 30, 195, 210 — mengonfirmasi mutasi terjadi in place, bukan pada salinan?
- Tugas 5: apakah traversal InOrder setelah mirror() menghasilkan InOrder ASLI yang dibalik (`70, 65, 10, 5, 3`)?
- Tugas 6: sudahkah Anda memeriksa bahwa fungsi height-balance Anda tidak menghitung ulang height() secara terpisah di setiap node tanpa perlu (bau $O(n^2)$)?
- Tugas 7: setelah menghapus sebuah node, apakah root subtree ditugaskan ulang dengan benar di PEMANGGIL (misalnya, `node->left = deleteKey(node->left, key);`), bukan hanya dimutasi "di tempat"?
- Tugas 8: apakah fungsi Anda dengan benar membedakan nilai yang MERUPAKAN root (tidak ada parent) dari nilai yang TIDAK ada dalam tree sama sekali?
- Tugas 9: apakah fungsi Anda dengan benar membedakan nilai yang merupakan LEAF (nol anak) dari nilai yang TIDAK ada dalam tree sama sekali?
- Tugas 10: apakah doubly linked list hasilnya berisi NOL node yang baru dialokasikan — hanya node tree asli, disambungkan ulang?
- Sudahkah tim Anda menghapus atau mengomentari baris debug `printf`/`cout` yang tersisa dan bukan bagian dari format keluaran yang diminta?
- Apakah setiap berkas yang dikumpulkan tim Anda benar-benar pekerjaan TIM ANDA SENDIRI, ditulis untuk kuis ini — bukan disalin dari tim lain, solusi tahun sebelumnya, atau alat AI yang tidak diizinkan kebijakan integritas akademik mata kuliah Anda sendiri?

Referensi

[1] Format asesmen bab latihan ini dimodelkan langsung dari Kuis 2 sesungguhnya mata kuliah ini (EF234201 Struktur Data): ujian coding tim (hingga 3 mahasiswa), open-book, take-home berisi

sepuluh tugas tree-rekursif atas contoh kerja insert(65, 5, 70, 3, 10) yang sama yang diajarkan Bab 11 mata kuliah ini. Riset proyek ini tidak dapat mengonfirmasi secara independen rincian poin per tugas yang tidak ambigu untuk seluruh sepuluh tugas dari materi sumber asli yang tersedia, sehingga buku teks ini mengalokasikan 10 poin per tugas yang rata dan didokumentasikan secara transparan (catatan kaki rubrik Bagian 12.4); jika instruksi mata kuliah Anda sendiri menetapkan rincian berbeda, ikuti instruksi tersebut.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” Edisi ke-3, MIT Press, 2009. (Binary search tree, Bab 12; tree seimbang, Bab 13.)

[3] cppreference.com. “Recursion,” “std::swap,” “Binary tree (data structure).” https://en.cppreference.com/w/cpp (diakses Agustus 2026).

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 13

Graf

Setiap struktur data yang telah dibangun mata kuliah ini -- array, list, stack, queue, bahkan tree pada Bab 11 -- membatasi setiap elemen memiliki paling banyak satu "induk". Sebuah GRAF melepaskan batasan itu sepenuhnya: sembarang simpul boleh terhubung ke sejumlah simpul lain, dengan pola apa pun. Materi bab ini seluruhnya BARU -- materi ajar mentah di balik mata kuliah ini ternyata tidak memiliki apa pun yang bisa dipakai untuk topik graf (slide kuliah "Graf"-nya, setelah diperiksa, ternyata duplikat persis dari slide kuliah Tree) -- sehingga baik teori maupun contoh kerja berulang di bawah ini disusun sepenuhnya baru, mengikuti pedagogi teori graf standar. Satu graf, peta kampus Pustaka Ceria, dipakai ulang tanpa perubahan pada setiap bagian: adjacency matrix, adjacency list, BFS, DFS, dan konektivitas. Setiap baris kode di sini benar-benar dikompilasi dengan g++ -Wall -Wextra -std=c++17, benar-benar dijalankan, dan diperiksa secara independen dengan valgrind -- transkrip lengkapnya direproduksi pada Lampiran 13.A.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Mendefinisikan kosakata inti graf -- simpul (vertex), sisi (edge), berarah vs. tak berarah, berbobot vs. tak berbobot, derajat (termasuk in-degree/out-degree terpisah untuk graf berarah), path, cycle, dan terhubung vs. terputus. (2) Membangun dan membandingkan dua representasi dari graf yang SAMA -- adjacency matrix dan adjacency list -- diimplementasikan, dikompilasi, dan dijalankan dalam C++, dengan perbedaan ruang dan kompleksitas yang benar-benar diukur, bukan sekadar diargumenkan secara abstrak. (3) Mengimplementasikan dan menelusuri Breadth-First Search (BFS) menggunakan Queue nyata -- Queue berbasis array milik Bab 5 sendiri, tanpa perubahan -- dan menjelaskan mengapa BFS mengunjungi graf level demi level. (4) Mengimplementasikan dan menelusuri Depth-First Search (DFS) dengan dua cara -- versi rekursif kecil (bentuk basis-kasus/kasus-rekursif Bab 10) dan versi iteratif menggunakan Stack nyata, juga milik Bab 5 -- serta mengonfirmasi keduanya sepakat pada himpunan simpul yang terjangkau. (5) Menemukan setiap connected component dari graf tak berarah, dan membangun pengujian sederhana "apakah graf ini terhubung?", keduanya langsung dari BFS. (6) Menerapkan seluruh konsep di atas pada satu contoh kerja berulang -- peta kampus Pustaka Ceria -- dipakai ulang secara konsisten dari representasi, traversal, hingga konektivitas.

13.1 Terminologi dan Dasar-Dasar Graf

Sebuah GRAF adalah kumpulan SIMPUL (vertex, juga disebut node -- bab ini memakai istilah "simpul"/"vertex" secara konsisten, istilah baku teori graf) yang dihubungkan oleh SISI (edge). Berbeda dari tree pada Bab 11, graf tidak memiliki satu root, tidak ada gagasan "induk" dan "anak", dan tidak ada larangan terhadap cycle: sembarang simpul boleh terhubung ke sejumlah simpul lain, dengan pola apa pun. Secara formal, sebuah graf G = (V, E) adalah himpunan simpul V bersama himpunan sisi E, dengan setiap sisi menghubungkan sepasang simpul.

13.1.1 Berarah vs. Tak Berarah

Sebuah sisi bersifat TAK BERARAH jika ia sekadar menghubungkan dua simpul tanpa arah tersirat -- jika ada sisi antara A dan B, Anda dapat berpindah A-ke-B atau B-ke-A secara setara. Sebuah sisi

bersifat BERARAH jika ia menunjuk dari satu simpul ke simpul lain secara spesifik -- sisi dari A ke B TIDAK berarti ada sisi dari B ke A. Sebuah graf disebut tak berarah atau berarah menurut jenis sisi yang dipakainya secara konsisten (mencampur keduanya dalam satu graf tidak lazim dan tidak dibahas di sini).

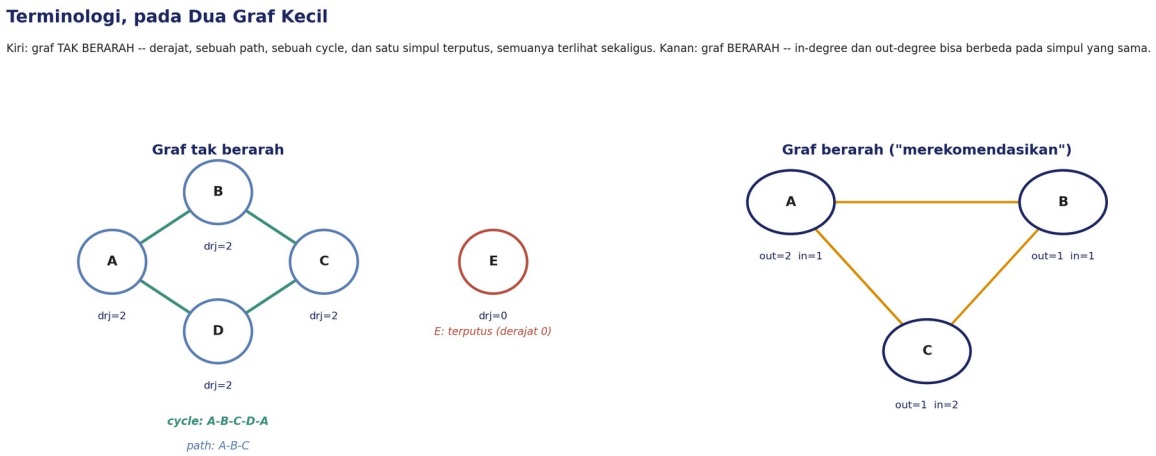
Peta kampus Pustaka Ceria (Bagian 13.1.4 dan seterusnya) bersifat TAK BERARAH: jalan setapak antara Library dan Cafeteria dapat dilalui pada kedua arah. Relasi "buku ini merekomendasikan buku itu", sebaliknya, secara alami bersifat BERARAH: BookA merekomendasikan BookB tidak berarti BookB merekomendasikan balik ke BookA.

13.1.2 Berbobot vs. Tak Berbobot

Sebuah sisi bersifat BERBOBOT jika ia membawa nilai numerik tambahan -- biaya, jarak, durasi -- di luar sekadar "kedua simpul ini terhubung". Sebuah sisi bersifat TAK BERBOBOT jika tidak ada nilai semacam itu yang dilekatkan (secara ekuivalen, setiap sisi dapat dianggap membawa bobot tersirat 1). Peta kampus Pustaka Ceria bersifat BERBOBOT: setiap jalan setapak membawa jarak nyata dalam meter, dipakai di seluruh kode bab ini.

13.1.3 Derajat, In-Degree, dan Out-Degree

DERAJAT sebuah simpul, pada graf tak berarah, sekadar jumlah sisi yang menyentuhnya. Pada graf BERARAH, satu angka saja tidak cukup -- sebuah simpul dapat memiliki sisi yang menunjuk MASUK dan sisi yang menunjuk KELUAR, dan keduanya dihitung terpisah: IN-DEGREE sebuah simpul adalah jumlah sisi yang menunjuk MASUK ke dalamnya, dan OUT-DEGREE adalah jumlah sisi yang menunjuk KELUAR darinya.



Gambar 13.1 — kiri: cycle tak berarah 4-simpul ditambah satu simpul terputus, dengan derajat setiap simpul diberi label. Kanan: graf berarah "merekomendasikan" kecil, dengan in-degree dan out-degree diberi label terpisah pada setiap simpul.

Pada graf berarah kecil di Gambar 13.1 (panel kanan): A merekomendasikan B dan C (out-degree 2), tetapi hanya C yang merekomendasikan balik ke A (in-degree 1) -- mengonfirmasi langsung bahwa in-degree dan out-degree bisa berbeda pada simpul yang sama persis, sesuatu yang tidak pernah terjadi pada graf tak berarah, di mana in dan out adalah hal yang sama menurut definisinya.

13.1.4 Path, Cycle, dan Konektivitas

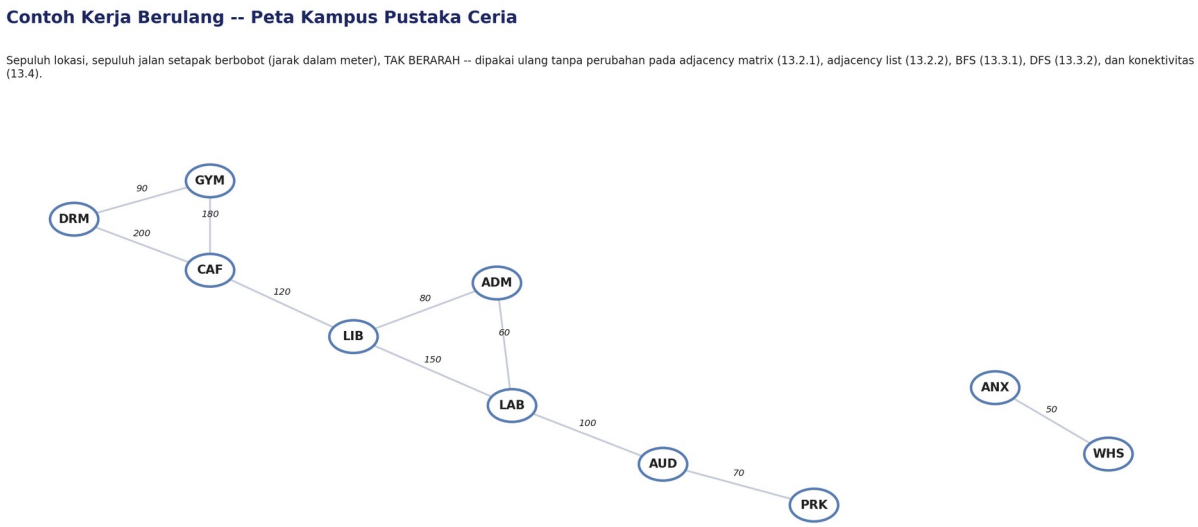
Sebuah PATH adalah urutan simpul di mana setiap pasangan berurutan dihubungkan oleh sebuah sisi -- panel kiri Gambar 13.1 menunjukkan path A-B-C. Sebuah CYCLE adalah path yang kembali ke simpul awalnya sendiri tanpa mengulang simpul lain di sepanjang jalan -- A-B-C-D-A adalah cycle pada graf yang sama. Sebuah graf (atau bagian darinya) bersifat TERHUBUNG (connected) jika ada path di antara setiap pasangan simpulnya; ia bersifat TERPUTUS (disconnected) jika setidaknya ada satu pasangan simpul yang tidak memiliki path sama sekali di antara keduanya. Panel kiri Gambar 13.1 sengaja dibuat terputus: simpul E berderajat 0 (tidak ada sisi sama sekali), sehingga tidak ada path yang menjangkaunya dari A, B, C, atau D.

> **"Terhubung" menjelaskan graf secara keseluruhan, bukan satu simpul**
> Satu simpul terisolasi seperti E tidak lantas membuat SELURUH graf "terputus" secara samar-samar -- ini presisi: graf {A,B,C,D,E} terputus karena setidaknya ada satu pasangan (pasangan mana pun yang melibatkan E) yang tidak memiliki path di antaranya, meskipun {A,B,C,D} sendiri SUDAH terhubung. Bagian 13.4 membuat ini presisi dengan algoritma nyata: connected COMPONENTS persis adalah bagian-bagian terhubung maksimal yang membentuk graf terputus -- di sini, {A,B,C,D} adalah satu komponen dan {E} adalah komponen lainnya.

13.2 Merepresentasikan Graf dalam Kode

Sebelum algoritma traversal apa pun dapat berjalan, sebuah graf perlu benar-benar tersimpan di suatu tempat dalam memori. Bagian ini membangun dan membandingkan DUA representasi dari graf yang persis sama -- sehingga perbandingan pada Bagian 13.2.3 adalah pengukuran pada satu struktur identik, bukan argumen tentang dua struktur yang berbeda.

Contoh kerja yang dipakai ulang untuk sisa bab ini adalah PETA KAMPUS Pustaka Ceria: sepuluh lokasi kampus, dihubungkan oleh sepuluh jalan setapak tak berarah berbobot (jarak dalam meter). Lokasi 0-7 membentuk klaster kampus utama di sekitar Library; lokasi 8-9 (Annex dan Warehouse kecil di luar area utama) terhubung satu sama lain tetapi -- sengaja, demi kepentingan Bagian 13.4 -- belum terhubung ke sisa kampus.



Gambar 13.2 — peta kampus Pustaka Ceria: contoh kerja berulang bab ini, dipakai ulang tanpa perubahan pada adjacency matrix, adjacency list, BFS, DFS, dan konektivitas.

13.2.1 Adjacency Matrix

Sebuah ADJACENCY MATRIX menyimpan graf dengan n simpul sebagai grid n x n: sel [u][v] menyimpan bobot sisi antara u dan v (atau penanda NO_EDGE jika tidak ada). Untuk graf tak berarah, matrix selalu simetris -- sel [u][v] selalu sama dengan sel [v][u], karena sisi tak berarah tidak memiliki arah yang membedakan keduanya.

```
class GraphMatrix {
public:
    static constexpr int NO_EDGE = -1;
    explicit GraphMatrix(int n) : n(n), mat(n, std::vector<int>(n, NO_EDGE)) {}

    void addEdge(int u, int v, int weight = 1) {
        mat[u][v] = weight;   // tak berarah: tulis KEDUA arah
        mat[v][u] = weight;
    }
    bool hasEdge(int u, int v) const { return mat[u][v] != NO_EDGE; }   // O(1)
    int degree(int v) const {                    // O(V): memindai SELURUH
baris
        int d = 0;
        for (int j = 0; j < n; ++j) if (mat[v][j] != NO_EDGE) ++d;
        return d;
    }
    long long cellCount() const { return (long long)n * n; }   // O(V^2), selalu
    // ... addEdge/hasEdge/weight/degree/print ...
};
```

Output nyata dari `demo_matrix.cpp` yang benar-benar dijalankan mengonfirmasi bentuk matrix yang sebenarnya pada peta kampus:

```
       LIB  CAF  DRM  GYM  ADM  LAB  AUD  PRK  ANX  WHS
  LIB    .  120    .    .   80  150    .    .    .    .
  CAF  120    .  200  180    .    .    .    .    .    .
  DRM    .  200    .   90    .    .    .    .    .    .
  GYM    .  180   90    .    .    .    .    .    .    .
  ADM   80    .    .    .    .   60    .    .    .    .
  LAB  150    .    .    .   60    .  100    .    .    .
  AUD    .    .    .    .    .  100    .   70    .    .
  PRK    .    .    .    .    .    .   70    .    .    .
  ANX    .    .    .    .    .    .    .    .    .   50
  WHS    .    .    .    .    .    .    .    .   50    .

hasEdge(LIB, ADM) = true   weight(LIB, ADM) = 80 m
degree(LIB) = 3   degree(ANX) = 1
numVertices() = 10  ->  cellCount() = 100 int dialokasikan (V*V),
dari situ hanya 20 sel yang menyimpan sisi nyata (2 per sisi tak berarah, 10
sisi).
Artinya 80 dari 100 sel (80%) adalah entri NO_EDGE yang terbuang.
```

13.2.2 Adjacency List

Sebuah ADJACENCY LIST menyimpan, untuk setiap simpul, hanya tetangga NYATA-nya -- sebuah `vector<vector<pair<int,int>>>` di sini, dengan adj[v] menyimpan setiap pasangan (tetangga, bobot) untuk simpul v. Untuk graf tak berarah, addEdge(u, v, w) mendorong sisi tersebut ke KEDUA list milik u dan v.

```
class GraphList {
public:
    void addEdge(int u, int v, int weight = 1) {
        adj[u].push_back({v, weight});   // tak berarah: dorong ke KEDUA list
        adj[v].push_back({u, weight});
    }
    bool hasEdge(int u, int v) const {            // O(degree(u)) -- memindai
HANYA list u
        for (const auto &nb : adj[u]) if (nb.first == v) return true;
        return false;
    }
    int degree(int v) const { return (int)adj[v].size(); }   // O(1)
    int totalEntries() const {                    // O(V+E), selalu
        int total = 0;
        for (const auto &lst : adj) total += (int)lst.size();
        return total;
    }
    // ... weight/edgeCount/neighbors/print, plus setiap algoritma traversal di
bawah ...
private:
    std::vector<std::vector<std::pair<int, int>>> adj;
};
```

Output nyata dari `demo_list.cpp` yang benar-benar dijalankan mengonfirmasi peta kampus yang SAMA, kali ini sebagai list, dan mengukur jejaknya secara langsung terhadap matrix:

```
  Library: Cafeteria(120), AdminOffice(80), LabBuilding(150)
  Cafeteria: Library(120), Dorm(200), Gym(180)
  Dorm: Cafeteria(200), Gym(90)
  Gym: Cafeteria(180), Dorm(90)
  AdminOffice: Library(80), LabBuilding(60)
  LabBuilding: Library(150), AdminOffice(60), Auditorium(100)
  Auditorium: LabBuilding(100), Parking(70)
  Parking: Auditorium(70)
  Annex: Warehouse(50)
  Warehouse: Annex(50)

numVertices() = 10, edgeCount() = 10  ->  totalEntries() = 20 pasangan
(neighbor,weight)
Bandingkan: adjacency MATRIX pada graf identik ini mengalokasikan 100 sel;
adjacency LIST mengalokasikan 20 entri -- 5x lebih sedikit, untuk graf sesparse
ini
(10 sisi dari kemungkinan 45).
```

13.2.3 Ruang dan Waktu: Perbandingan Langsung

Kedua representasi baru saja diukur pada peta kampus yang PERSIS SAMA -- sehingga angka-angka di bawah ini nyata, bukan hipotetis. Ini langsung memperluas kebiasaan perbandingan Big-O yang ditetapkan Bab 3 untuk algoritma sorting.

Operasi	Adjacency Matrix	Adjacency List	Diukur (V=10, E=10)
Ruang	O(V^2)	O(V + E)	100 sel vs. 20 entri
hasEdge(u,v)	O(1)	O(degree(u))	matrix menang untuk query ini
Iterasi tetangga v	O(V)	O(degree(v))	list menang -- hanya kunjungi tetangga nyata
degree(v)	O(V)	O(1)	list menang -- sekadar panjang list
Tambah satu sisi	O(1)	O(1)	seri
Tambah satu simpul	O(V^2) (resize seluruh matrix)	O(1) (tambah list kosong)	list menang
Cocok untuk	graf padat (E mendekati V^2)	graf sparse (E << V^2)	peta kampus: sparse -- list menang keseluruhan

Mengapa setiap traversal di bab ini ditulis di atas LIST

BFS dan DFS pada Bagian 13.3 hanya pernah membutuhkan satu operasi, berulang kali: "berikan tetangga nyata v, agar saya bisa mengunjungi masing-masing." Adjacency list menyerahkan ini secara langsung, dalam O(degree(v)), persis yang disimpannya; adjacency matrix akan memaksa pemindaian O(V) atas seluruh baris hanya untuk menemukan segelintir tetangga nyata yang sama, yang sebagian besar selnya adalah NO_EDGE. Inilah persisnya mengapa graph_list.h, bukan graph_matrix.h, menjadi tempat setiap algoritma traversal dan konektivitas di bab ini.

13.3 Traversal: Mengunjungi Setiap Simpul yang Terjangkau

Sebuah GRAPH TRAVERSAL mengunjungi setiap simpul yang terjangkau dari suatu simpul awal, masing-masing tepat satu kali. Bab ini membahas dua strategi traversal -- Breadth-First Search (BFS) dan Depth-First Search (DFS) -- keduanya diterapkan pada titik awal yang SAMA, Library, pada peta kampus yang SAMA.

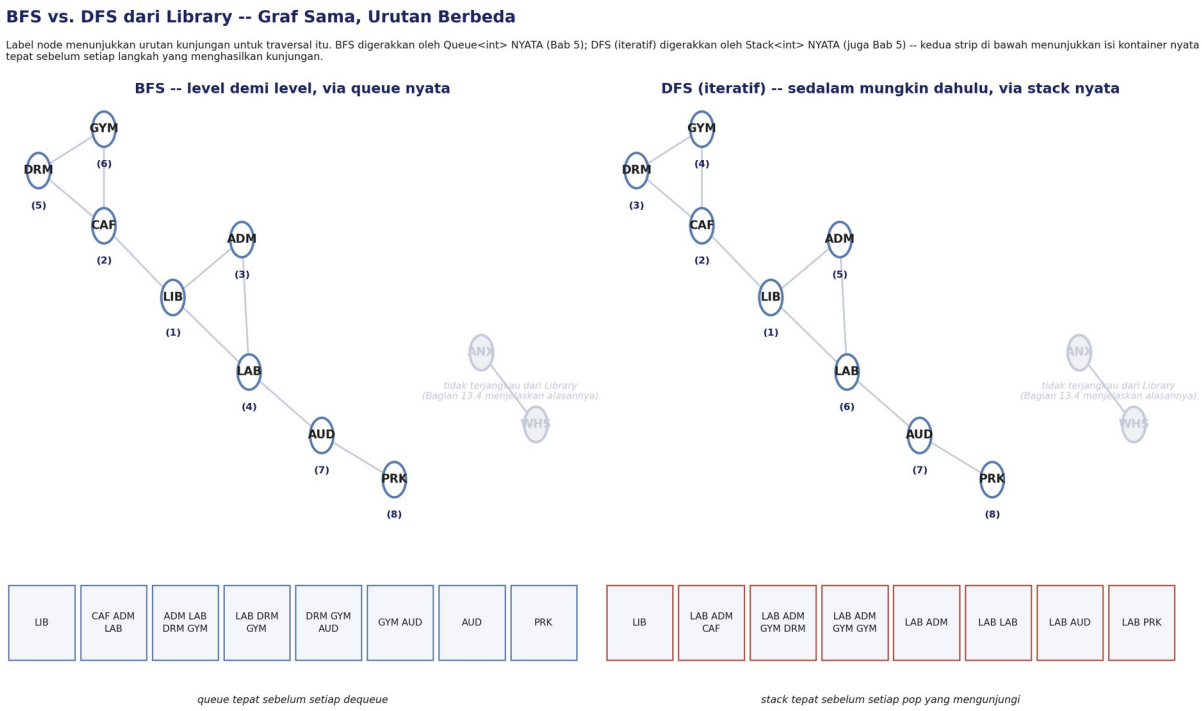
13.3.1 Breadth-First Search (BFS) — Digerakkan oleh Queue Nyata

BFS mengunjungi graf LEVEL DEMI LEVEL: pertama simpul awal, lalu setiap simpul yang berjarak SATU sisi, lalu setiap simpul berjarak DUA sisi, dan seterusnya. Cara standar mengimplementasikannya adalah dengan queue: enqueue simpul awal, lalu berulang kali dequeue satu simpul, kunjungi, dan enqueue setiap tetangganya yang belum dikunjungi. Ini bukan sekadar analogi terhadap Queue Bab 5 -- `bfs()` di bawah secara harfiah membuat instance `Queue<int>` (Queue berbasis array milik Bab 5 sendiri, `queue.h`, disalin maju tanpa perubahan) dan memanggil `enqueue()`/`dequeue()`-nya, disiplin FIFO yang persis sama yang dibangun Bab 5 secara manual, kini menggerakkan penelusuran breadth-first pada sebuah graf.

```
std::vector<int> bfs(int start) const {
    std::vector<int> order;
    std::vector<bool> visited(adj.size(), false);
    Queue<int> q(static_cast<int>(adj.size()) + 1);   // Queue Bab 5, dipakai
langsung di sini
    visited[start] = true;
    q.enqueue(start);
    while (!q.isEmpty()) {
        int u; q.dequeue(u);
        order.push_back(u);
        for (const auto &nb : adj[u]) {
            if (!visited[nb.first]) { visited[nb.first] = true;
q.enqueue(nb.first); }
        }
    }
    return order;
}
```

Tandai simpul dikunjungi saat DI-ENQUEUE, bukan saat di-dequeue

Jika sebuah simpul hanya ditandai dikunjungi pada saat dequeue, simpul yang sama bisa di-enqueue lebih dari sekali oleh dua tetangga berbeda sebelum dequeue pertamanya -- membuang kerja, dan pada graf dengan cycle, berpotensi meng-enqueue-nya berkali-kali tanpa batas. Menandai `visited[nb.first] = true` pada saat enqueue (bukan pada saat dequeue) menjamin setiap simpul di-enqueue paling banyak satu kali.



Gambar 13.3 — BFS (kiri) dan DFS iteratif (kanan), keduanya mulai dari Library pada peta kampus yang identik. Label simpul menunjukkan urutan kunjungan masing-masing traversal; strip di bawah menunjukkan isi Queue/Stack nyata tepat sebelum setiap langkah yang menghasilkan kunjungan. Annex dan Warehouse (diredupkan) tidak pernah terjangkau dari Library -- Bagian 13.4 menjelaskan alasannya.

Transkrip nyata dari `demo_bfs.cpp` yang benar-benar dijalankan, ditelusuri langkah demi langkah:

```
  step 1: queue = [ Library ]  -> dequeue Library, kunjungi, enqueue Cafeteria,
enqueue AdminOffice, enqueue LabBuilding
  step 2: queue = [ Cafeteria AdminOffice LabBuilding ]  -> dequeue Cafeteria,
kunjungi, enqueue Dorm, enqueue Gym
  step 3: queue = [ AdminOffice LabBuilding Dorm Gym ]  -> dequeue AdminOffice,
kunjungi
  step 4: queue = [ LabBuilding Dorm Gym ]  -> dequeue LabBuilding, kunjungi,
enqueue Auditorium
  step 5: queue = [ Dorm Gym Auditorium ]  -> dequeue Dorm, kunjungi
  step 6: queue = [ Gym Auditorium ]  -> dequeue Gym, kunjungi
  step 7: queue = [ Auditorium ]  -> dequeue Auditorium, kunjungi, enqueue
Parking
  step 8: queue = [ Parking ]  -> dequeue Parking, kunjungi

Urutan kunjungan BFS: [ Library Cafeteria AdminOffice LabBuilding Dorm Gym
Auditorium Parking ]
Mengunjungi 8 dari 10 lokasi. TIDAK terjangkau dari Library: Annex Warehouse
```

13.3.2 Depth-First Search (DFS) — Rekursif, dan Iteratif dengan Stack Nyata

DFS mengunjungi graf SEDALAM MUNGKIN sepanjang satu cabang sebelum mundur (backtrack) -- urutan penjelajahan yang berkebalikan dari penyebaran level-demi-level BFS, pada graf yang persis sama. Bab ini mengimplementasikan DFS dengan dua cara, keduanya mulai dari Library.

13.3.2a DFS, Rekursif

Versi rekursif adalah fungsi kecil dan langsung -- callback konkret lain terhadap bentuk basis-kasus/kasus-rekursif Bab 10, kali ini diterapkan pada graf, bukan satu rantai pemanggilan atau tree:

```
void dfsRecHelper(int u, std::vector<bool> &visited, std::vector<int> &order)
const {
    visited[u] = true;
    order.push_back(u);
    for (const auto &nb : adj[u]) {
        if (!visited[nb.first]) dfsRecHelper(nb.first, visited, order);   //
KASUS REKURSIF
    }
    // tidak perlu baris kasus-basis eksplisit: loop sekadar tidak lagi
    // merekursi begitu semua tetangga sudah dikunjungi
}
```

13.3.2b DFS, Iteratif — Digerakkan oleh Stack Nyata

Versi iteratif menggantikan call stack implisit dengan `Stack<int>` NYATA -- Stack berbasis array milik Bab 5 sendiri, `stack.h`, disalin maju tanpa perubahan. Ia menandai simpul dikunjungi SAAT DI-POP (bukan saat di-push), bentuk standar buku teks yang sederhana -- artinya simpul yang sama BISA di-push lebih dari sekali sebelum kunjungan pertamanya, dan pop berikutnya atas simpul yang sudah dikunjungi cukup dilewati.

```
std::vector<int> dfsIterative(int start) const {
    std::vector<int> order;
    std::vector<bool> visited(adj.size(), false);
    Stack<int> st(static_cast<int>(adj.size()) * 4 + 4);   // Stack Bab 5,
dipakai langsung di sini
    st.push(start);
    while (!st.isEmpty()) {
        int u; st.pop(u);
        if (visited[u]) continue;                          // sudah dikunjungi
-- lewati
        visited[u] = true;
        order.push_back(u);
        // push tetangga dalam urutan TERBALIK, agar tetangga pertama dalam list
        // berakhir di ATAS dan di-pop (dikunjungi) lebih dulu
        for (auto it = adj[u].rbegin(); it != adj[u].rend(); ++it) {
            if (!visited[it->first]) st.push(it->first);
        }
    }
    return order;
}
```

Transkrip nyata dari `demo_dfs.cpp` yang benar-benar dijalankan, kedua versi:

```
Urutan kunjungan DFS (rekursif): [ Library Cafeteria Dorm Gym AdminOffice
LabBuilding Auditorium Parking ]

  step 1: stack = [ Library ]  -> pop Library, kunjungi, push LabBuilding, push
AdminOffice, push Cafeteria
  step 2: stack = [ LabBuilding AdminOffice Cafeteria ]  -> pop Cafeteria,
kunjungi, push Gym, push Dorm
  step 3: stack = [ LabBuilding AdminOffice Gym Dorm ]  -> pop Dorm, kunjungi,
push Gym
  step 4: stack = [ LabBuilding AdminOffice Gym Gym ]  -> pop Gym, kunjungi
  step 5: stack = [ LabBuilding AdminOffice Gym ]  -> pop Gym (sudah dikunjungi,
lewati)
  step 6: stack = [ LabBuilding AdminOffice ]  -> pop AdminOffice, kunjungi,
push LabBuilding
  step 7: stack = [ LabBuilding LabBuilding ]  -> pop LabBuilding, kunjungi,
push Auditorium
  step 8: stack = [ LabBuilding Auditorium ]  -> pop Auditorium, kunjungi, push
Parking
  step 9: stack = [ LabBuilding Parking ]  -> pop Parking, kunjungi
  step 10: stack = [ LabBuilding ]  -> pop LabBuilding (sudah dikunjungi,
lewati)

Urutan kunjungan DFS (iteratif): [ Library Cafeteria Dorm Gym AdminOffice
LabBuilding Auditorium Parking ]

HIMPUNAN simpul yang dikunjungi sama (urutan diabaikan)? YA
URUTAN kunjungan sama, rekursif vs. iteratif? YA -- push tetangga dalam urutan
adjacency
terbalik membuat stack eksplisit meniru call stack secara persis
```

Dua dari sepuluh langkah stack melakukan kerja ekstra yang nyata terlihat -- dan itu wajar

Langkah 5 dan 10 di atas melakukan pop atas simpul (Gym, lalu LabBuilding) yang ternyata sudah dikunjungi, dan sekadar melanjutkan. Ini adalah biaya jujur dari gaya "push tanpa memeriksa, lewati saat pop": ia melakukan sedikit lebih banyak kerja dibanding varian "periksa sebelum push" yang lebih ketat, tetapi lebih sederhana untuk ditulis dengan benar dan tetap mengunjungi setiap simpul yang terjangkau tepat satu kali pada hasil `order` akhir -- dikonfirmasi langsung oleh pemeriksaan otomatis `demo_all.cpp` (Lampiran 13.A), bukan sekadar diklaim di sini.

13.4 Konektivitas: Connected Components dan "Apakah Graf Ini Terhubung?"

BFS dari Library pada Bagian 13.3 hanya menjangkau 8 dari 10 lokasi peta kampus -- Annex dan Warehouse tidak pernah di-enqueue sama sekali, karena tidak ada jalan setapak yang menghubungkannya ke sisa kampus. Bagian ini membuat pengamatan itu presisi dan otomatis: menemukan setiap CONNECTED COMPONENT dari graf tak berarah, dan membangun pengujian "apakah graf ini terhubung?" yang sederhana dan langsung -- keduanya langsung dari BFS, tanpa algoritma baru.

13.4.1 Connected Components

Sebuah CONNECTED COMPONENT adalah himpunan simpul maksimal di mana setiap pasangan memiliki path di antaranya. Menemukan setiap komponen dari sebuah graf adalah perluasan kecil dan

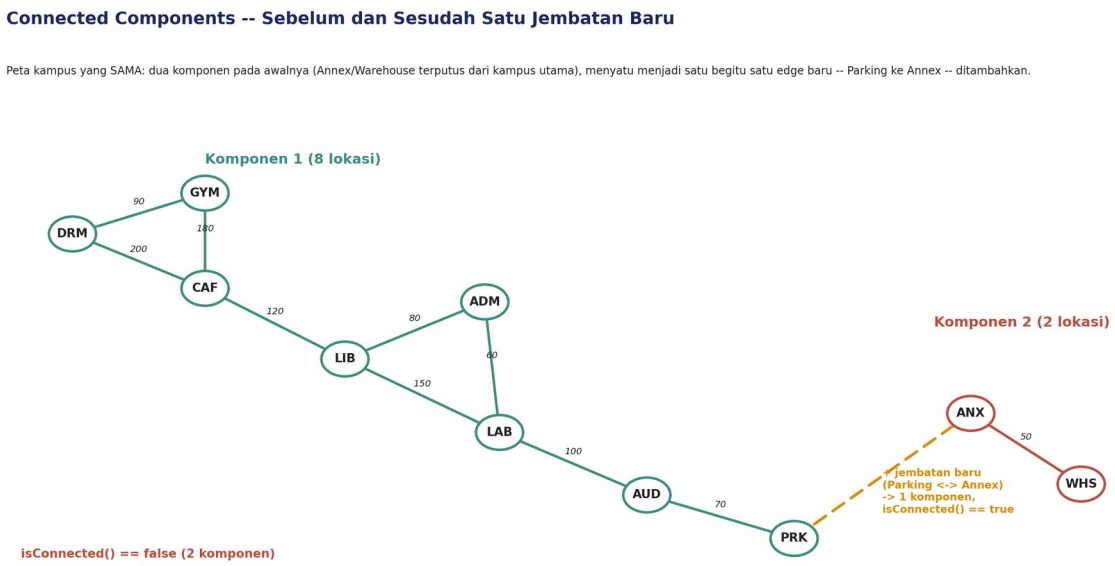
langsung dari BFS: ulangi BFS dari setiap simpul yang BELUM dikunjungi oleh sapuan sebelumnya; setiap sapuan BFS semacam itu menyapu tepat satu komponen utuh, karena BFS sendiri sudah mengunjungi setiap simpul yang terjangkau dari titik awalnya.

```
std::vector<std::vector<int>> connectedComponents() const {
    std::vector<std::vector<int>> comps;
    std::vector<bool> visited(adj.size(), false);
    for (int v = 0; v < static_cast<int>(adj.size()); ++v) {
        if (!visited[v]) {
            comps.push_back(bfsMarkComponent(v, visited));   // satu sapuan BFS
penuh = satu komponen
        }
    }
    return comps;
}
```

13.4.2 "Apakah Graf Ini Terhubung?"

Khusus untuk graf TAK BERARAH, ada pengujian yang lebih sederhana lagi: seluruh graf terhubung jika dan hanya jika SATU KALI BFS dari sembarang satu simpul menjangkau setiap simpul lainnya. (Jalan pintas ini khusus untuk graf tak berarah -- graf berarah membutuhkan gagasan yang lebih kuat, STRONG connectivity, di mana path harus ada pada KEDUA arah di antara setiap pasangan, yang berada di luar cakupan bab ini.)

```
bool isConnected() const {
    if (adj.empty()) return true;
    return bfs(0).size() == adj.size();   // menjangkau semua orang <=>
terhubung
}
```



Gambar 13.4 — peta kampus yang sama: dua connected component sebelum ada jembatan sama sekali, menyatu menjadi satu begitu satu jembatan baru (Parking ke Annex) ditambahkan.

Transkrip nyata dari `demo_connectivity.cpp` yang benar-benar dijalankan -- pertama pada peta kampus seperti yang dibangun, lalu setelah benar-benar menambahkan satu sisi baru:

```
Sebelum jembatan: 2 connected component
  Komponen 1 (8 lokasi): [ Library Cafeteria AdminOffice LabBuilding Dorm Gym
Auditorium Parking ]
  Komponen 2 (2 lokasi): [ Annex Warehouse ]
  isConnected() = false

=== Membangun jembatan baru: Parking <-> Annex ===

Setelah jembatan: 1 connected component
  Komponen 1 (10 lokasi): [ Library Cafeteria AdminOffice LabBuilding Dorm Gym
Auditorium
                          Parking Annex Warehouse ]
  isConnected() = true
```

Satu sisi baru, benar-benar dijalankan ulang, bukan sekadar dideskripsikan ulang

`addFootbridge()` menambahkan tepat SATU sisi baru -- Parking ke Annex -- pada objek `GraphList` yang SAMA yang dibangun sebelumnya, dan `connectedComponents()`/`isConnected()` dipanggil lagi pada graf yang kini sudah dimodifikasi tersebut, menghitung jawaban baru yang nyata, bukan memiliki "versi terhubung" kedua dari graf yang di-hardcode secara terpisah di suatu tempat. Kampus benar-benar berubah dari 2 komponen menjadi 1, dan `isConnected()` benar-benar berbalik dari `false` menjadi `true`, dikonfirmasi oleh output program nyata di atas.

13.5 Ringkasan Bab dan Poin-Poin Kunci

- Sebuah GRAF menggeneralisasi setiap struktur yang telah dibangun mata kuliah ini sejauh ini: sebuah simpul boleh terhubung ke sejumlah simpul lain, dengan pola apa pun, tanpa satu root tunggal dan tanpa larangan terhadap cycle. Kosakata inti -- simpul/sisi, berarah vs. tak berarah, berbobot vs. tak berbobot, derajat (in-/out-degree untuk graf berarah), path, cycle, terhubung vs. terputus -- dipakai di seluruh bab ini.
- Seluruh materi bab ini BARU -- materi ajar mentah di balik mata kuliah ini tidak memiliki apa pun yang bisa dipakai untuk topik graf -- dibangun di sekitar satu contoh kerja berulang, peta kampus Pustaka Ceria, dipakai ulang tanpa perubahan pada setiap representasi dan algoritma.
- ADJACENCY MATRIX (ruang O(V^2), hasEdge O(1), degree/iterasi-tetangga O(V)) dan ADJACENCY LIST (ruang O(V+E), hasEdge/degree/iterasi-tetangga O(degree(v))) keduanya dibangun pada peta kampus yang identik, dan diukur langsung: 100 sel vs. 20 entri -- perbedaan nyata 5x pada graf ini, bukan sekadar klaim Big-O abstrak.
- BFS digerakkan oleh Queue<int> NYATA milik Bab 5 dan mengunjungi level demi level; DFS -- baik versi rekursif kecil (bentuk basis-kasus/kasus-rekursif Bab 10) maupun versi iteratif yang digerakkan Stack<int> NYATA milik Bab 5 -- mengunjungi sedalam mungkin dahulu. Pada peta kampus bab ini, kedua varian DFS kebetulan menghasilkan urutan kunjungan yang identik, konsekuensi langsung dari push tetangga versi iteratif dalam urutan adjacency terbalik.
- Connected components dan "apakah graf ini terhubung?" keduanya adalah perluasan kecil dan langsung dari BFS -- tidak ada algoritma baru yang dibutuhkan. Peta kampus benar-

benar memiliki 2 komponen (8 lokasi, lalu 2) hingga satu sisi jembatan baru menyatukannya menjadi 1, dengan isConnected() berbalik dari false ke true pada komputasi nyata yang dijalankan ulang.

- Setiap program pada bab ini (demo_matrix, demo_list, demo_bfs, demo_dfs, demo_connectivity, demo_all) benar-benar dikompilasi tanpa peringatan, benar-benar dijalankan, dan benar-benar diperiksa dengan valgrind -- 0 error, 0 leak, pada keenam binari -- sesuai kebijakan verifikasi baku mata kuliah ini sejak Bab 9.

13.6 Pertanyaan Tinjauan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Jelaskan, dengan kata-kata Anda sendiri, mengapa adjacency LIST adalah representasi alami untuk BFS/DFS, sementara adjacency MATRIX adalah representasi alami untuk satu query hasEdge(u,v) -- merujuk pada biaya Big-O spesifik dari Bagian 13.2.3.
2. Pada peta kampus Pustaka Ceria, telusuri BFS mulai dari Parking (bukan Library) secara manual, dengan cara yang sama seperti Bagian 13.3.1 menelusurinya dari Library, dan nyatakan urutan kunjungan yang dihasilkan serta lokasi mana (jika ada) yang tidak terjangkau.
3. Jelaskan mengapa menandai sebuah simpul dikunjungi PADA SAAT ENQUEUE (bukan pada saat dequeue) esensial bagi kebenaran BFS, dan deskripsikan secara konkret apa yang bisa salah jika `bfs()` malah menandai simpul dikunjungi hanya saat di-dequeue.
4. DFS iteratif pada Bagian 13.3.2b terkadang melakukan pop atas simpul yang sudah dikunjungi (langkah 5 dan 10 pada transkrip nyata). Jelaskan mengapa ini bisa terjadi dengan gaya "push tanpa memeriksa", dan mengapa ini tidak menyebabkan simpul mana pun dikunjungi dua kali pada hasil `order` akhir.
5. Sebuah graf memiliki 12 simpul dan, setelah menjalankan connectedComponents(), ternyata memiliki tepat 3 komponen berukuran 5, 4, dan 3. Nyatakan apakah isConnected() akan mengembalikan true atau false untuk graf ini, dan jelaskan penalaran Anda langsung dari definisi Bagian 13.4.2.
6. Andaikan, alih-alih satu jembatan baru (Parking <-> Annex), Pustaka Ceria membangun jembatan baru yang langsung menghubungkan Dorm ke Warehouse. Apakah ini juga akan menyatukan kedua komponen menjadi satu? Jelaskan mengapa ya atau mengapa tidak, merujuk pada lokasi mana yang sudah terhubung dengan Dorm dan Warehouse masing-masing.

13.7 Lampiran 13.A — Log Validasi

Keenam program di bawah ini (`demo_matrix.cpp`, `demo_list.cpp`, `demo_bfs.cpp`, `demo_dfs.cpp`, `demo_connectivity.cpp`, `demo_all.cpp`) dikompilasi dengan `g++ -Wall -Wextra -std=c++17` (nol peringatan dari keenamnya) dan benar-benar dieksekusi; transkrip di bawah adalah output terminal

nyata, tanpa diedit. `valgrind --leak-check=full --show-leak-kinds=all` benar-benar dijalankan terhadap keenam program, sesuai kebijakan baku mata kuliah ini sejak Bab 9.

13.A.1 demo_all.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_all demo_all.cpp
$ ./demo_all
=== Chapter 13 Validation Summary: Graphs ===

[Matrix vs List: hasEdge/weight agreement (45 pairs)]   55/55 checks OK
[Matrix vs List: degree() agreement (10 vertices)]      10/10 checks OK
[BFS reachability from Library]                          5/5 checks OK
[DFS recursive + iterative vs. BFS]                      7/7 checks OK
[Connected components -- before footbridge]              3/3 checks OK
[Connected components -- after footbridge]               3/3 checks OK
[Handshake lemma (sum of degrees == 2 x edges)]          5/5 checks OK

Overall: 88 / 88 checks passed -- ALL CHECKS PASSED
```

13.A.2 valgrind (keenam program)

```
$ valgrind --leak-check=full --show-leak-kinds=all ./demo_all
==20763== HEAP SUMMARY:
==20763==     in use at exit: 0 bytes in 0 blocks
==20763==   total heap usage: 436 allocs, 436 frees, 92,863 bytes allocated
==20763== All heap blocks were freed -- no leaks are possible
==20763== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

(demo_matrix, demo_list, demo_bfs, demo_dfs, demo_connectivity: hasil sama,
 0 error dari 0 konteks, seluruh blok heap dibebaskan, pada keenam program)
```

Catatan kejujuran

Setiap sel matrix, entri list, urutan traversal, daftar komponen, dan baris terminal yang ditunjukkan pada bab ini (di keenam program) berasal langsung dari kompilasi-dan-jalankan nyata di lingkungan build bab ini. Pemeriksaan validasi bab ini TIDAK menemukan bug pada logika GraphMatrix, GraphList, BFS, DFS, atau konektivitas -- diungkapkan dengan jujur, sesuai kebijakan baku mata kuliah ini, bukan mengarang temuan padahal tidak ada. `valgrind` dijalankan terhadap setiap satu dari keenam program dan setiap kali menghasilkan nol leak dan nol error. Zip kode yang dikirim diekstrak ulang dan di-build ulang secara independen dari `make clean && make run` yang bersih untuk memastikan deliverable yang sebenarnya, bukan sekadar salinan kerja, dikompilasi dan berjalan secara identik.

Referensi

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Representasi graf, BFS, DFS, connected components.)

[2] M.A. Weiss. “Data Structures and Algorithm Analysis in C++.” 4th ed., Pearson, 2014. (Algoritma dan aplikasi traversal graf.)

[3] B.R. Preiss. “Data Structures and Algorithms with Object-Oriented Design Patterns in C++.” John Wiley & Sons, 1998. (Adjacency matrix/list, ADT graf.)

[4] P.J. Deitel, H.M. Deitel. “C++ How to Program.” 10th ed., Pearson, 2017. (Dasar-dasar graf, rekursi diterapkan pada struktur non-tree.)

[5] cppreference.com. “std::queue”, “std::stack”. https://en.cppreference.com/w/cpp/container (diakses Agustus 2026).

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 14

Heap & Priority Queue

Bab 3 memperkenalkan Heap Sort dengan NAMA, dengan jaminan O(n log n)-nya diargumentasikan dari prinsip dasar, tetapi dengan sengaja menunda penunjukan implementasi nyatanya hingga mesin heap itu sendiri ada. Bab ini membangun mesin tersebut: sebuah MaxHeap di atas array biasa, dan seluruh keluarga operasi yang didukungnya -- insert, extractMax, getMax/getMin, buildMaxHeap, increaseKey, isMaxHeap, dan, akhirnya, Heap Sort secara utuh. Kelas MaxHeap pada bab ini tidak ditulis dari nol -- ia ditemukan kembali, dengan setia, dari materi Ujian Akhir Semester mata kuliah DS yang asli: sebuah Heap.cpp sepanjang 281 baris yang sudah lengkap dari arsip model-jawaban Ujian Akhir, dikonfirmasi kata demi kata terhadap skeleton yang tercetak di soal ujian itu sendiri. Ini penting melampaui bab ini saja: Ujian Akhir (Bab 16) memberikan mahasiswa skeleton yang persis sama ini dan meminta mereka menulis lima fungsi "tambahan ujian akhir"-nya sendiri (extractMax, increaseKey, getMin, buildMaxHeap, isMaxHeap -- 50 dari poin ujian), sehingga segala sesuatu yang diajarkan dan didemonstrasikan bekerja pada bab ini adalah persis permukaan API yang menjadi sandaran latihan tersebut. Setiap baris kode di sini benar-benar dikompilasi dengan g++ -Wall -Wextra -std=c++17, benar-benar dijalankan, dan diperiksa secara independen dengan valgrind -- transkrip lengkapnya direproduksi di Lampiran 14.A.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Menjelaskan mengapa PRIORITY QUEUE -- "selalu berikan saya item berprioritas tertinggi/terendah, dengan cepat" -- membutuhkan struktur selain array terurut atau array tak terurut, dan menyatakan operasi apa yang harus didukungnya. (2) Merepresentasikan complete binary tree secara implisit sebagai array biasa, menggunakan aritmetika indeks induk=(i-1)/2, kiri=2i+1, kanan=2i+2, dan menyatakan properti MAX-HEAP (induk >= kedua anak) secara presisi. (3) Mengimplementasikan dan menelusuri insert() (tambah + heapifyUp/"percolate up") dan extractMax()/deleteRoot() (pindahkan terakhir ke root + heapifyDown/"percolate down"), pada satu contoh kerja yang konsisten. (4) Membangun heap dari array sembarang dalam O(n) via buildMaxHeap()/heapifyArray() -- konstruksi bottom-up, BUKAN n pemanggilan insert() terpisah -- dan menjelaskan mengapa ini lebih cepat. (5) Mengimplementasikan increaseKey() dan getMin() dengan benar, termasuk penanganan exception C++ yang tepat untuk setiap mode kegagalan yang bisa ditemui masing-masing. (6) Menulis dan menggunakan isMaxHeap() sebagai pemeriksa validitas, dan menerapkannya sebagai alat pengujian bab ini sendiri, bukan sekadar sebagai fungsi ujian. (7) Mengimplementasikan Heap Sort secara utuh -- buildMaxHeap() diikuti ekstraksi berulang -- dan menjelaskan jaminan O(n log n)-nya, melengkapi apa yang diperkenalkan Bab 3 dengan nama saja.

14.1 Konsep Priority Queue

Setiap struktur yang telah dibangun mata kuliah ini sejauh ini menjawab pertanyaan berbeda dengan baik. Array Bab 2 menjawab "apakah X ada, dan di mana?" Stack/Queue Bab 5 menjawab "apa yang terakhir/pertama masuk?" PRIORITY QUEUE menjawab pertanyaan yang sama sekali berbeda: "dari semua yang sedang menunggu, mana yang PALING penting sekarang?" -- dan ia perlu menjawab pertanyaan itu dengan cepat, berulang kali, seiring item baru terus berdatangan.

Perhatikan antrean PERMINTAAN HOLD MENDESAK milik Pustaka Ceria: pemustaka mengajukan hold pada buku, masing-masing membawa skor urgensi (pinjaman antar-perpustakaan yang terlambat, buku yang dibutuhkan untuk praktikum besok, permintaan telusur-nanti yang rutin). Perpustakaan perlu selalu melayani permintaan paling mendesak berikutnya -- tetapi permintaan baru terus berdatangan di sela-sela itu, tanpa urutan tertentu.

Dua struktur yang tampak jelas sama-sama kurang memadai:

- Array TAK TERURUT: insert() adalah O(1) (cukup tambahkan) -- tetapi menemukan maksimum, atau menghapusnya, adalah O(n): pemindaian penuh setiap kali sebuah permintaan dilayani.
- Array TERURUT (dijaga terurut setiap saat): menemukan/menghapus maksimum adalah O(1) (ia berada di satu ujung) -- tetapi insert() adalah O(n): kedatangan baru harus digeser ke posisi terurutnya yang benar, rata-rata memindahkan setengah array.

Sebuah BINARY HEAP, subjek bab ini, memberikan KEDUA operasi tersebut batas yang baik sekaligus: insert() dalam O(log n), dan temukan-maksimum dalam O(1) dengan hapus-maksimum dalam O(log n). Itulah seluruh alasan priority queue biasanya diimplementasikan sebagai heap dibanding kedua jenis array -- Bagian 14.3 membangun persis operasi-operasi ini, dan demo_library_priority.cpp (kode bab ini) menjalankan skenario permintaan-hold-mendesak yang persis ini secara nyata.

> **Heap tetap menggunakan kunci int biasa -- dan itu pilihan yang disengaja**
> Kelas MaxHeap bab ini bekerja di atas kunci int biasa sepanjang bab, sesuai persis dengan Heap.cpp yang ditemukan kembali dan menjaga algoritma inti (heapifyUp/heapifyDown/aritmetika indeks) tidak dipenuhi oleh tipe payload apa pun. demo_library_priority.cpp menunjukkan bagaimana aplikasi nyata melekatkan payload dalam praktiknya: skor urgensi int menggerakkan heap itu sendiri, sementara lookup paralel terpisah memetakan setiap nilai urgensi kembali ke judul buku -- persis cara pustaka priority-queue nyata (atau std::priority_queue dengan comparator kustom) memungkinkan Anda mengurutkan berdasarkan satu field sambil membawa sisa record ikut serta.

14.2 Binary Heap sebagai Array

Binary heap, secara konseptual, adalah COMPLETE BINARY TREE: setiap level terisi penuh kecuali mungkin level terakhir, yang terisi ketat dari kiri ke kanan tanpa celah. Kelengkapan inilah yang membuat ARRAY -- bukan pointer -- menjadi representasi alami: setiap posisi pada complete binary tree berkorespondensi dengan tepat satu indeks array, tanpa perlu menyimpan pointer kiri/kanan sama sekali.

14.2.1 Aritmetika Indeks

Untuk sebuah simpul yang disimpan pada indeks array i (berbasis 0), kerabatnya berada pada offset tetap, dihitung dengan aritmetika bilangan bulat biasa:

Kerabat	Rumus indeks	Contoh (i = 4)

Induk (parent)	(i - 1) / 2 (pembagian bulat)	(4-1)/2 = 1
Anak kiri	2i + 1	2*4+1 = 9
Anak kanan	2i + 2	2*4+2 = 10

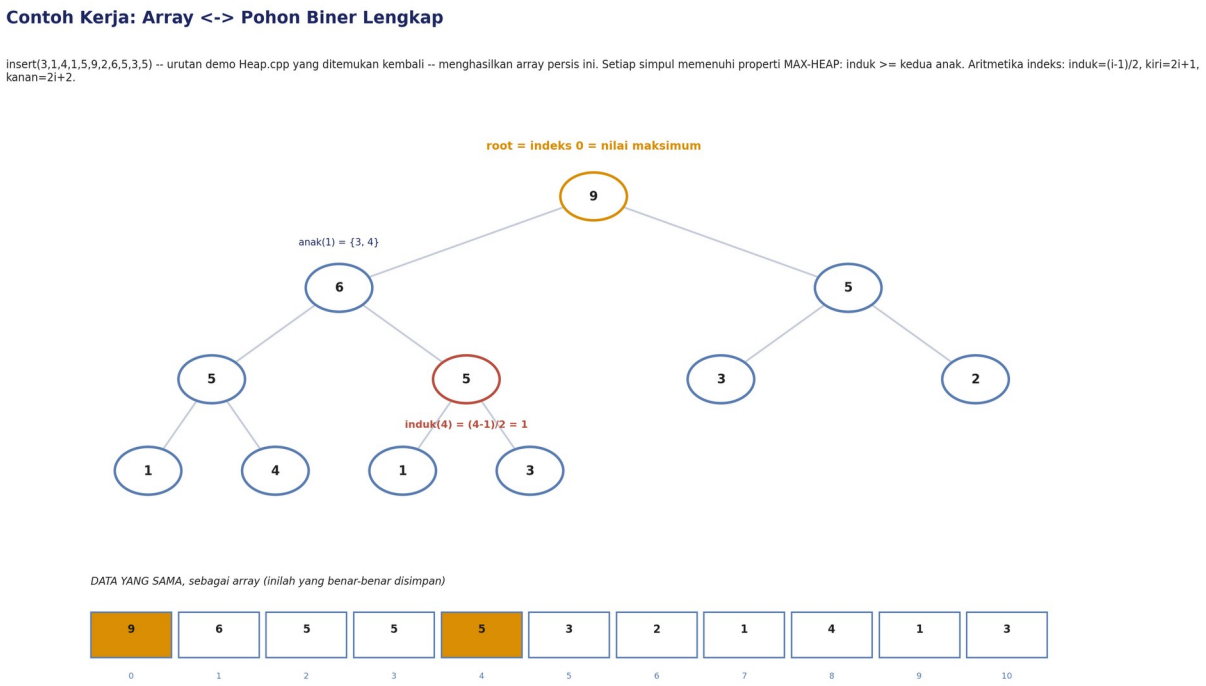
Tidak ada pointer yang perlu disimpan untuk salah satu pun dari ini -- berbeda dengan BST Bab 11, di mana left/right/parent adalah field pointer nyata di dalam setiap simpul. Inilah seluruh keunggulan penyimpanan heap: n kunci, n sel array, tidak ada lagi.

14.2.2 Properti Max-Heap

MAX-HEAP memenuhi satu properti, diperiksa pada setiap simpul: nilai pada simpul mana pun lebih besar atau sama dengan nilai pada masing-masing anaknya. Diterapkan secara rekursif, ini menjamin nilai terbesar tunggal di seluruh struktur berada di root -- indeks array 0 -- yang persis merupakan alasan getMax() (Bagian 14.3.3) adalah O(1): jawabannya selalu berada tepat di sana.

> **Properti heap TIDAK MENGATAKAN APA-APA tentang kiri vs. kanan, atau tentang dua saudara mana pun**
>
> Max-heap hanya membatasi induk-ke-anak, tidak pernah anak-ke-anak. Dua simpul bersaudara bisa berada pada urutan apa pun relatif satu sama lain, dan heap, secara umum, TIDAK terurut ketika dibaca dari kiri ke kanan -- contoh kerja Gambar 14.1 membuat ini konkret: membaca array `9 6 5 5 5 3 2 1 4 1 3` dari kiri ke kanan sama sekali bukan urutan terurut, namun setiap pasangan induk-anak di dalamnya memenuhi induk >= anak. Heap Sort pada Bagian 14.7 persis adalah algoritma yang mengubah array yang heap-order (tetapi belum sepenuhnya terurut) ini menjadi array yang benar-benar terurut.

SATU contoh kerja yang berjalan pada bab ini, dipakai ulang di setiap bagian di bawah, dibangun dengan menyisipkan 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 -- urutan persis yang sama dengan yang dipakai main() milik Heap.cpp yang ditemukan kembali (tujuh digit pertama pi, ditambah empat pengulangan). Gambar 14.1 menunjukkan heap yang dihasilkan baik sebagai tree maupun sebagai array yang benar-benar menyimpannya.



Gambar 14.1 — heap contoh kerja, insert(3,1,4,1,5,9,2,6,5,3,5), ditunjukkan baik sebagai complete binary tree maupun sebagai array yang benar-benar menyimpannya. Setiap induk lebih besar atau sama dengan kedua anaknya.

14.3 Operasi Inti: insert, deleteRoot/extractMax, getMax

14.3.1 insert() — Tambahkan, Lalu heapifyUp() ("Percolate Up")

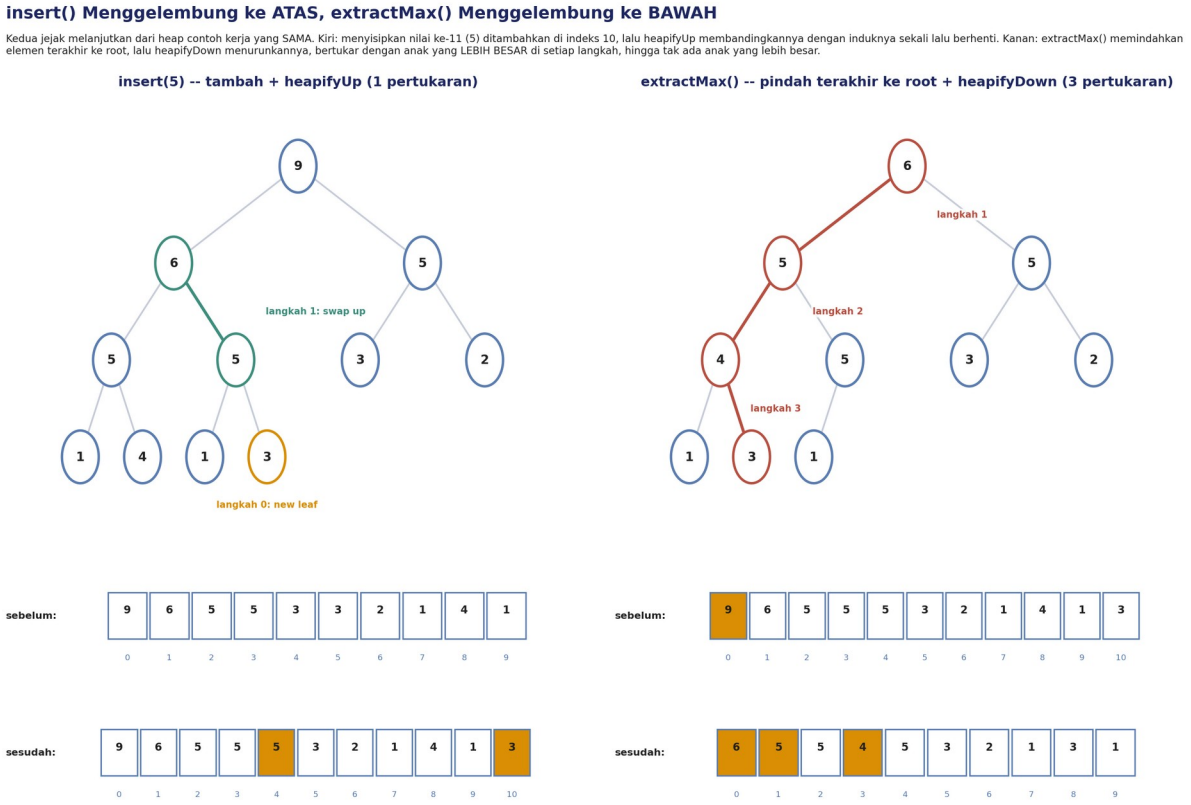
Menyisipkan kunci baru adalah dua langkah: tambahkan sebagai elemen TERAKHIR yang baru (menjaga kelengkapan -- array cukup bertambah satu), lalu bandingkan berulang kali terhadap induknya, bertukar ke atas selagi ia lebih besar, hingga baik ia mencapai root maupun induknya tidak lagi lebih kecil. Penelusuran ke atas ini disebut heapifyUp (juga "sift up" atau "percolate up").

```
void insert(int key) {
    heap.push_back(key);
    size_t index = heap.size() - 1;
    heapifyUp(index);
}

void heapifyUp(size_t index) {
    if (index > 0 && heap[(index - 1) / 2] < heap[index]) {
        std::swap(heap[index], heap[(index - 1) / 2]);
        heapifyUp((index - 1) / 2);   // lanjut dari posisi baru
    }
}
```

Transkrip nyata dan benar-benar dijalankan dari `demo_insert.cpp` membangun contoh kerja satu penyisipan pada satu waktu:

```
insert(3) -> 3
insert(1) -> 3 1
insert(4) -> 4 1 3
insert(1) -> 4 1 3 1
insert(5) -> 5 4 3 1 1
insert(9) -> 9 4 5 1 1 3
insert(2) -> 9 4 5 1 1 3 2
insert(6) -> 9 6 5 4 1 3 2 1
insert(5) -> 9 6 5 5 1 3 2 1 4
insert(3) -> 9 6 5 5 3 3 2 1 4 1
insert(5) -> 9 6 5 5 5 3 2 1 4 1 3

getMax() = 9   size() = 11
```

Panel kiri Gambar 14.2 menelusuri penyisipan TERAKHIR (angka 5 kedua) secara detail: ia ditambahkan sebagai leaf baru di indeks 10, dibandingkan dengan induknya di indeks 4 (nilai 3), dan ditukar ke atas persis satu kali, karena induk barunya di indeks 1 (nilai 6) tidak lebih kecil. Satu perbandingan-dan-mungkin-tukar yang murah ini, diulang paling banyak tinggi-pohon = O(log n) kali, adalah seluruh biaya insert().



Gambar 14.2 — kiri: heapifyUp milik insert(), satu pertukaran. Kanan: heapifyDown milik extractMax(), tiga pertukaran, selalu memilih anak yang LEBIH BESAR di setiap langkah.

14.3.2 deleteRoot() vs. extractMax() pada Bagian 14.5.1 — Dua Cara Menghapus Maksimum

Menghapus maksimum adalah operasi cermin dari insert(): salin elemen TERAKHIR ke tempat root, kecilkan array satu, lalu bandingkan berulang kali root (yang kini sudah dipindahkan) terhadap anaknya yang LEBIH BESAR, bertukar ke bawah selagi sebuah anak lebih besar, hingga tak ada anak

yang lebih besar atau leaf tercapai. Penelusuran ke bawah ini adalah heapifyDown (juga "sift down" atau "percolate down").

Skeleton dasar Heap.cpp yang ditemukan kembali menamai operasi ini `deleteRoot()`, dan melaporkan heap kosong dengan MENCETAK pesan dan return -- ia tidak melempar exception. `extractMax()` pada Bagian 14.5.1, ditulis untuk Ujian Akhir, melakukan penghapusan yang persis sama, tetapi melempar `std::runtime_error` pada heap kosong. Menjaga keduanya berdampingan adalah disengaja: inilah persis upgrade "fungsi skeleton" menjadi "fungsi berkualitas ujian" yang akan diminta Bab 16 kepada mahasiswa untuk keempat fungsi ujian lainnya juga.

```
void deleteRoot() {                          // skeleton ASLI
    if (heap.empty()) {
        std::cout << "Heap is empty" << std::endl;
        return;
    }
    heap[0] = heap.back();
    heap.pop_back();
    if (!heap.empty()) heapifyDown(0);
}

void heapifyDown(size_t index) {
    size_t largest = index, left = 2*index+1, right = 2*index+2;
    if (left  < heap.size() && heap[left]  > heap[largest]) largest = left;
    if (right < heap.size() && heap[right] > heap[largest]) largest = right;
    if (largest != index) {
        std::swap(heap[index], heap[largest]);
        heapifyDown(largest);            // lanjut dari posisi baru
    }
}
```

Transkrip nyata dan benar-benar dijalankan dari `demo_extract.cpp`, kedua fungsi penghapusan pada salinan independen dari heap contoh kerja yang SAMA:

```
Worked-example heap: 9 6 5 5 5 3 2 1 4 1 3
getMax() = 9
After deleteRoot(): 6 5 5 4 5 3 2 1 3 1
After a second deleteRoot(): 5 5 5 4 1 3 2 1 3

Worked-example heap: 9 6 5 5 5 3 2 1 4 1 3
extractMax() -> 9   heap now: 6 5 5 4 5 3 2 1 3 1
extractMax() -> 6   heap now: 5 5 5 4 1 3 2 1 3

deleteRoot() on an EMPTY heap (no exception, just a message):
Heap is empty
getMax() on an EMPTY heap (no exception, sentinel -1): Heap is empty
-1

extractMax() on an EMPTY heap -- this one THROWS:
  caught std::runtime_error: "Heap is empty"
```

Panel kanan Gambar 14.2 menelusuri pemanggilan extractMax() PERTAMA secara utuh: root (9) dihapus, elemen terakhir (3) menggantikannya, dan ia berjalan turun tiga level penuh, bertukar dengan anak yang lebih besar setiap kali (indeks 0 ke 1, 1 ke 3, 3 ke 8), hingga mencapai posisi leaf di mana tak ada anak yang ada. Tiga pertukaran di sini dibanding satu untuk bubble-up milik insert() bukan kebetulan -- root yang baru saja dipindahkan bisa, dalam kasus terburuk, perlu menempuh SELURUH tinggi pohon, persis batas O(log n) yang dibagi kedua operasi.

14.3.3 getMax() — Peek O(1)

Membaca maksimum, tanpa menghapus apa pun, sekadar membaca indeks 0 -- tanpa traversal, tanpa perbandingan, O(1):

```
int getMax() const {
    if (heap.empty()) { std::cout << "Heap is empty" << std::endl; return -1; }
    return heap[0];
}
```

14.4 Membangun Heap dari Array Sembarang

Misalkan seluruh array dari n kunci tak terurut sudah ada, dan perlu menjadi heap valid sekaligus -- persis langkah pertama Heap Sort sendiri (Bagian 14.7). Memanggil insert() n kali berhasil, tetapi berbiaya O(n log n) secara keseluruhan. Cara yang lebih baik ada: heapifyDown(), diterapkan bottom-up, dimulai dari simpul BUKAN-leaf terakhir dan bekerja mundur ke root.

14.4.1 heapifyArray() (Skeleton Asli) dan buildMaxHeap() (Ujian Akhir) — Ide yang Sama, Dua Kali

Skeleton dasar Heap.cpp yang ditemukan kembali sudah menyertakan konstruksi bottom-up ini, dinamai `heapifyArray()`. Fungsi `buildMaxHeap()` milik Ujian Akhir sendiri meminta mahasiswa menulis konstruksi yang identik. Inilah persis inti Bagian 14.4: fungsi ujian bukan algoritma baru untuk ditemukan -- ia adalah ide yang sama yang sudah didemonstrasikan skeleton, satu bagian sebelumnya.

```
void heapifyArray(const std::vector<int> &arr) {   // skeleton asli
    heap = arr;
    for (int i = static_cast<int>(heap.size()) / 2 - 1; i >= 0; --i) {
        heapifyDown(static_cast<size_t>(i));
    }
}

void buildMaxHeap(const std::vector<int> &arr) {   // fungsi Ujian Akhir [10
poin]
    heap = arr;
    int n = static_cast<int>(heap.size());
    for (int i = n / 2 - 1; i >= 0; --i) {
        heapifyDown(static_cast<size_t>(i));
    }
}
```

> **Mengapa mulai dari n/2 - 1, dan mengapa ini O(n), bukan O(n log n)?**
> Setiap indeks dari n/2 dan seterusnya adalah LEAF (ia tidak punya anak, sehingga heapifyDown padanya tidak melakukan apa-apa) -- n/2 - 1 persis adalah indeks terakhir dengan setidaknya satu anak, sehingga loop dimulai dari simpul internal terdalam yang mungkin dan bekerja ke atas. Batas O(n) (bukan O(n log n) yang akan dikenakan oleh n pemanggilan heapifyUp() terpisah) berasal dari fakta yang lebih halus: SEBAGIAN BESAR simpul berada dekat dasar pohon, di mana heapifyDown() memiliki jarak yang sangat sedikit untuk ditempuh -- dijumlahkan di semua level, total kerja dibatasi oleh suatu konstanta dikali n, bukan n log n. (Derivasi lengkap adalah milik mata kuliah algoritma; poin praktis untuk mata kuliah ini sekadar: bangun sekali dengan buildMaxHeap()/heapifyArray(), jangan pernah dengan n pemanggilan insert() terpisah, ketika seluruh array sudah tersedia sejak awal.)

Transkrip nyata dan benar-benar dijalankan dari `demo_buildheap.cpp`, memakai ulang dua array uji milik Heap.cpp yang ditemukan kembali sendiri:

```
heapifyArray(arr1={10,7,8,5,2,3,1}) -> 10 7 8 5 2 3 1
heapifyArray(arr2={10,7,6,5,1,8,2}) -> 10 7 8 5 1 6 2

buildMaxHeap(arr1) -> 10 7 8 5 2 3 1
Same result as heapifyArray(arr1)? YES -- identical arrays
buildMaxHeap(arr2) -> 10 7 8 5 1 6 2
Same result as heapifyArray(arr2)? YES -- identical arrays
```

14.5 Lima Fungsi Ujian Akhir

Ujian Akhir (Bab 16, 50 dari poinnya) memberikan mahasiswa skeleton dasar di atas dan meminta lima fungsi tambahan, masing-masing bernilai 10 poin. Bab ini mengimplementasikan, mendemonstrasikan, dan benar-benar menguji kelima-limanya, sehingga latihan Bab 16 bersandar pada API yang telah dilihat mahasiswa bekerja dengan benar.

14.5.1 extractMax() — Sudah Dibahas pada Bagian 14.3.2

Lihat Bagian 14.3.2 di atas: bentuk penghapusan yang identik dengan deleteRoot(), melempar `std::runtime_error("Heap is empty")` alih-alih mencetak dan return.

14.5.2 increaseKey(index, newValue) — Naikkan Kunci, Lalu Pulihkan Urutan ke Atas

increaseKey() menaikkan kunci pada indeks tertentu ke nilai baru (lebih besar), lalu memanggil heapifyUp() dari indeks itu untuk memulihkan properti max-heap -- persis penelusuran ke atas yang sama yang sudah dipakai insert(), hanya dimulai dari indeks yang sudah ada alih-alih yang baru ditambahkan. Dua mode kegagalan yang BERBEDA mendapat dua tipe exception yang BERBEDA, karena keduanya adalah jenis kesalahan pemanggil yang berbeda: indeks di luar jangkauan adalah kesalahan posisi; "nilai baru" yang justru LEBIH KECIL melanggar nama fungsinya sendiri -- "increase" -- dan akan diam-diam merusak heap jika dibiarkan lolos.

```
void increaseKey(int index, int newValue) {
    if (index < 0 || static_cast<size_t>(index) >= heap.size()) {
        throw std::out_of_range("Index is out of range");
    }
    if (heap[static_cast<size_t>(index)] > newValue) {
        throw std::invalid_argument("New value is smaller than the current
value");
    }
    heap[static_cast<size_t>(index)] = newValue;
    heapifyUp(static_cast<size_t>(index));
}
```

`demo_increasekey.cpp` memakai ulang pemanggilan persis milik Heap.cpp yang ditemukan kembali sendiri, `increaseKey(2, 8)`, tepat setelah satu extractMax() pada contoh kerja -- transkrip nyata dan benar-benar dijalankannya:

```
Worked-example heap: 9 6 5 5 5 3 2 1 4 1 3
extractMax() -> 9   heap now: 6 5 5 4 5 3 2 1 3 1

Before: heap[2] = 5
After increaseKey(2, 8): 8 5 6 4 5 3 2 1 3 1
Still a valid max-heap? YES

increaseKey(-1, 100)      -> caught std::out_of_range: "Index is out of range"
increaseKey(size(), 100)  -> caught std::out_of_range: "Index is out of range"
increaseKey(2, 1)         -> caught std::invalid_argument: "New value is smaller
than the current value"
```

14.5.3 getMin() — Temukan Minimum, Tanpa Menghapus Apa Pun

Max-heap TIDAK menjanjikan apa pun tentang di mana minimum berada -- kecuali satu fakta yang berguna: ia selalu ditemukan di antara LEAF (simpul tanpa anak, menempati indeks array size()/2 hingga size()-1). Ini mengikuti langsung dari properti heap: jika minimum keseluruhan berada pada suatu simpul internal, simpul itu harus >= anak-anaknya (properti heap), tetapi ia JUGA nilai terkecil di mana pun, sehingga anak-anaknya harus sama dengannya juga -- dan penalaran ini berulang hingga ke bawah menuju suatu leaf, yang karenanya menyimpan nilai minimum yang sama itu. Jadi getMin() hanya perlu memindai rentang leaf, sekitar setengah array, bukan pemindaian penuh O(n) atas segalanya:

```
int getMin() const {
    if (heap.empty()) throw std::runtime_error("Heap is empty");
    int minVal = heap[0];
    for (size_t i = heap.size() / 2; i < heap.size(); ++i) {
        minVal = std::min(minVal, heap[i]);
    }
    return minVal;
}
```

Transkrip lanjutan dari `demo_increasekey.cpp` mengonfirmasi ini secara langsung:

```
Current heap: 8 5 6 4 5 3 2 1 3 1
getMin() = 1   (scans only the LEAF range [size()/2, size()))
size() unchanged after getMin()? size() = 10

getMin() on empty heap -> caught std::runtime_error: "Heap is empty"
```

getMin() adalah O(n/2), bukan O(1) -- dan itu jawaban yang jujur dan benar

Max-heap dioptimalkan untuk menemukan MAKSIMUM dalam O(1); minimum membutuhkan pemindaian sekitar setengah array, karena properti heap hanya mengurutkan induk di atas anak, tidak pernah saudara terhadap saudara, sehingga minimum bisa berada pada leaf MANA PUN. Ini adalah keterbatasan struktural nyata dari max-heap sebagai priority queue "item terkecil" -- jika BAIK getMax() MAUPUN getMin() perlu O(1)/O(log n), struktur berbeda (MIN-MAX HEAP, atau dua heap yang dijaga sinkron) diperlukan, di luar cakupan bab ini.

14.6 isMaxHeap() — Pemeriksa Validitas

isMaxHeap() memeriksa apakah array SEMBARANG sudah memenuhi properti max-heap, tanpa membangun atau mengubah apa pun: untuk setiap indeks simpul internal i, kedua anaknya (jika ada) tidak boleh lebih besar dari arr[i].

```
static bool isMaxHeap(const std::vector<int> &arr) {
    if (arr.size() < 2) return true;    // 0 atau 1 elemen: tak ada yang
dilanggar
    for (size_t i = 0; i <= (arr.size() - 2) / 2; ++i) {
        if (arr[i] < arr[2*i + 1] ||
            (2*i + 2 < arr.size() && arr[i] < arr[2*i + 2])) {
            return false;
        }
    }
    return true;
}
```

Perbaikan satu baris, ditemukan dan diungkapkan selama QA bab ini

Versi asli yang ditemukan kembali menghitung batas loop-nya sebagai `(arr.size() - 2) / 2` tanpa memeriksa ukuran array terlebih dahulu. `arr.size()` adalah `size_t` (tak bertanda); untuk array 0 atau 1 elemen, `arr.size() - 2` UNDERFLOW menjadi nilai yang sangat besar sebelum guard di atas ditambahkan, dan baris berikutnya kemudian akan membaca `arr[2*i + 1]` jauh di luar batas. Heap berukuran 0 atau 1 secara trivial memenuhi properti heap-order -- tidak ada pasangan induk/anak untuk dilanggar -- sehingga perbaikannya benar sekaligus hanya satu baris tambahan. Ini ditemukan saat menulis pemeriksaan otomatis bab ini sendiri (Lampiran 14.A), tidak ada pada input uji ujian asli sendiri, yang tidak pernah kebetulan mencoba array kosong atau satu elemen.

Transkrip nyata dan benar-benar dijalankan dari `demo_buildheap.cpp`, memakai dua array uji milik Heap.cpp yang ditemukan kembali sendiri:

```
arr1 = 10 7 8 5 2 3 1
isMaxHeap(arr1) = true  (every parent >= its children already)

arr2 = 10 7 6 5 1 8 2
isMaxHeap(arr2) = false  (index 2 holds 6, but its LEFT child at index 5
   holds 8; 6 < 8 breaks parent >= child right there, even though index 0's
   own children, arr[1]=7 and arr[2]=6, are both fine on their own.)

isMaxHeap({})   = true (trivially true)
isMaxHeap({42}) = true (trivially true)
```

14.7 Heap Sort — Melengkapi Survei Pengurutan Bab 3

Bab 3 memperkenalkan Heap Sort dengan NAMA di antara tujuh algoritma pengurutannya, mengargumentasikan jaminan worst-case O(n log n)-nya dari prinsip dasar, dan dengan sengaja menunda penunjukan implementasi nyata hingga mesin heap bab ini ada. Mesin itu kini ada secara utuh -- Heap Sort sekadar dua blok pembangun bab ini sendiri, diterapkan berurutan: buildMaxHeap() (Bagian 14.4, O(n)), diikuti ekstraksi maksimum berulang ke akhir hasil yang terus tumbuh (heapSort() milik Bagian 14.3, n ekstraksi pada O(log n) masing-masing).

```
std::vector<int> heapSort() {
    std::vector<int> sorted;
    while (!heap.empty()) {
        sorted.push_back(getMax());   // O(1)
        deleteRoot();                 // O(log n)
    }
    return sorted;                    // n ekstraksi -> O(n log n) keseluruhan
}
```

Karena getMax()/deleteRoot() selalu menghapus maksimum SAAT INI, output heapSort() sendiri keluar dalam urutan MENURUN -- membalikkannya menghasilkan pengurutan menaik konvensional. Transkrip nyata dan benar-benar dijalankan dari `demo_heapsort.cpp`, diperiksa langsung terhadap `std::sort()` pada input yang sama:

```
Unsorted input:  29 4 17 8 42 15 1 33 9 21 6
After buildMaxHeap() -- a valid max-heap, not yet sorted:
  42 33 17 29 21 15 1 8 9 4 6
isMaxHeap() on this built array? true

heapSort() output (max extracted first, so this is DESCENDING):
  42 33 29 21 17 15 9 8 6 4 1
Reversed to ASCENDING order (the conventional "sorted" direction):
  1 4 6 8 9 15 17 21 29 33 42

Genuinely sorted (ascending) by heapSort()? YES
Matches std::sort() on the same input?     YES -- identical
heap.size() after heapSort() drained it: 0 (expected 0)

n = 1000 -- genuinely sorted? YES   matches std::sort()? YES -- identical
```

Algoritma (Bab)	Worst case	Ruang	Stabil?	Kontribusi bab ini sendiri
Bubble / Selection / Insertion (Bab 3)	O(n^2)	O(1)	Bubble/Insertion: ya	-- tak berubah dari Bab 3
Quicksort (Bab 3)	O(n^2) terburuk, O(n log n) rata-rata	O(log n)	Tidak	-- tak berubah dari Bab 3
Merge Sort (Bab 3)	O(n log n)	O(n)	Ya	-- tak berubah dari Bab 3
Heap Sort (nama saja di Bab 3 -> utuh di Bab 14)	O(n log n)	O(1)	Tidak	Implementasi utuh yang bekerja, BAB INI

Keunggulan nyata Heap Sort: worst-case O(n log n), dalam ruang EKSTRA O(1)
Merge Sort juga menjamin worst-case O(n log n), tetapi membutuhkan ruang ekstra O(n) untuk penggabungan. Quicksort mengurutkan di tempat (ruang ekstra O(log n) untuk rekursinya) tetapi worst-case O(n^2)-nya adalah risiko nyata pada input adversarial. Heap Sort adalah satu-satunya dari keempatnya dengan BAIK JAMINAN worst-case O(n log n) MAUPUN ruang ekstra O(1) (di luar array output itu sendiri) -- trade-off yang hanya bisa diargumentasikan Bab 3 secara abstrak kini didemonstrasikan langsung, pada kode nyata yang benar-benar dijalankan.

14.8 Ringkasan Bab dan Poin-Poin Kunci

- PRIORITY QUEUE membutuhkan "insert cepat, temukan/hapus maksimum cepat" secara bersamaan -- baik array terurut maupun tak terurut tidak memberikan keduanya; binary heap memberikan insert() dalam O(log n) dan getMax()/extractMax() dalam O(1)/O(log n).
- Binary heap adalah COMPLETE BINARY TREE yang disimpan secara implisit sebagai array -- induk=(i-1)/2, kiri=2i+1, kanan=2i+2, tanpa pointer sama sekali -- memenuhi properti MAX-HEAP (induk >= kedua anak) pada setiap simpul.
- Kelas MaxHeap bab ini DITEMUKAN KEMBALI, tidak ditulis dari nol: Heap.cpp lengkap sepanjang 281 baris milik Ujian Akhir asli, dikonfirmasi kata demi kata terhadap skeleton yang tercetak di soal ujian itu sendiri. Bagian dasarnya (insert/deleteRoot/getMax/heapifyArray/heapSort) adalah persis skeleton yang akan diberikan Bab 16 kepada mahasiswa; lima fungsi ujiannya (extractMax/increaseKey/getMin/buildMaxHeap/isMaxHeap) direproduksi dengan algoritma yang identik.
- insert() menambahkan + heapifyUp() ("percolate up"); deleteRoot()/extractMax() memindahkan elemen terakhir ke root + heapifyDown() ("percolate down") -- dengan sengaja dijaga sebagai dua nama untuk operasi yang SAMA, mengontraskan penanganan kesalahan cetak-dan-return milik skeleton asli terhadap upgrade pelemparan-exception yang tepat milik Ujian Akhir.
- buildMaxHeap()/heapifyArray() membangun heap dari array sembarang dalam O(n), dengan menjalankan heapifyDown() secara bottom-up dari simpul bukan-leaf terakhir -- BUKAN via n insert() terpisah O(log n), yang akan berbiaya O(n log n) keseluruhan.
- increaseKey() melempar std::out_of_range untuk indeks yang salah dan std::invalid_argument untuk nilai yang justru akan menurunkan kunci; getMin() hanya memindai rentang leaf (sekitar setengah array) karena minimum max-heap terbukti selalu ditemukan di antara leaf-nya; isMaxHeap() adalah pemeriksa validitas statis, diperbaiki selama QA bab ini untuk menangani array 0/1-elemen dengan benar.
- Heap Sort -- buildMaxHeap() + ekstraksi berulang -- memberikan implementasi utuh worst-case O(n log n), ruang-ekstra O(1) yang hanya diperkenalkan Bab 3 dengan nama, diverifikasi langsung terhadap std::sort() pada input hingga n=2000.
- Setiap program pada bab ini (demo_insert, demo_extract, demo_buildheap, demo_increasekey, demo_heapsort, demo_library_priority, demo_all) benar-benar

dikompilasi tanpa peringatan, benar-benar dijalankan, dan benar-benar diperiksa dengan valgrind -- 0 error, 0 leak, pada ketujuh binary -- sesuai kebijakan verifikasi baku mata kuliah ini sejak Bab 9.

14.9 Pertanyaan Tinjauan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Menggunakan array contoh kerja bab ini, 9 6 5 5 5 3 2 1 4 1 3, hitunglah INDEKS induk, anak kiri, dan anak kanan dari indeks 3 dengan tangan, dan nyatakan apakah properti heap-order berlaku untuk simpul itu terhadap kedua anaknya.
2. Jelaskan, dengan kata-kata Anda sendiri, mengapa insert() dan extractMax() keduanya O(log n) pada kasus terburuk, dengan merujuk secara spesifik pada apa yang dibatasi oleh "tinggi complete binary tree dengan n simpul".
3. Telusuri buildMaxHeap() dengan tangan pada array {1, 2, 3, 4, 5, 6, 7} (sudah terurut menaik, yang sejauh mungkin dari heap untuk array 7-elemen) -- nyatakan indeks mana yang dipanggil heapifyDown()-nya, dalam urutan apa, dan array heap akhirnya.
4. Jelaskan mengapa getMin() tidak bisa sekadar mengembalikan heap[size()-1] atau indeks tetap lain mana pun, dan mengapa memindai rentang leaf secara spesifik (bukan seluruh array) terbukti cukup.
5. Diberikan array heap {50, 40, 45, 20, 10, 30, 44}, panggil increaseKey(5, 60) (indeks 5 saat ini memegang 30). Nyatakan array setelah pemanggilan, dan jelaskan perbandingan heapifyUp() mana yang benar-benar menyebabkan pertukaran.
6. Jelaskan mengapa output heapSort() sendiri keluar dalam urutan MENURUN, dan apa perbaikan satu baris untuk memperoleh urutan menaik -- dengan merujuk pada transkrip nyata demo_heapsort.cpp sendiri.
7. isMaxHeap() milik Heap.cpp yang ditemukan kembali aslinya akan crash (pembacaan di luar batas) pada array kosong, sebelum perbaikan satu baris bab ini. Jelaskan, dalam istilah aritmetika size_t secara spesifik, persis mengapa `(arr.size() - 2) / 2` berbahaya ketika arr.size() adalah 0 atau 1.

14.10 Lampiran 14.A — Log Validasi

Ketujuh program di bawah (`demo_insert.cpp`, `demo_extract.cpp`, `demo_buildheap.cpp`, `demo_increasekey.cpp`, `demo_heapsort.cpp`, `demo_library_priority.cpp`, `demo_all.cpp`) dikompilasi dengan `g++ -Wall -Wextra -std=c++17` (tanpa peringatan dari ketujuhnya) dan benar-benar dijalankan; transkrip di bawah adalah output terminal nyata yang tidak diedit. `valgrind --leak-check=full --show-leak-kinds=all` benar-benar dijalankan terhadap ketujuh program, sesuai kebijakan baku mata kuliah ini sejak Bab 9.

14.A.1 demo_all.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_all demo_all.cpp
$ ./demo_all
=== Chapter 14 Validation Summary: Heaps & Priority Queues ===

[insert()/heapifyUp: heap-order property after every insert]  13/13 OK
[deleteRoot()/extractMax(): heap-order property after every removal]  23/23 OK
[Exception handling: extractMax/getMin/increaseKey error paths]  6/6 OK
[buildMaxHeap()/heapifyArray(): agreement + isMaxHeap() checks]  9/9 OK
[heapSort(): genuinely sorted + matches std::sort()]  5/5 OK
[getMin(): matches brute-force min across 5 heap shapes]  5/5 OK

Overall: 61 / 61 checks passed -- ALL CHECKS PASSED
```

14.A.2 valgrind (ketujuh program)

```
$ valgrind --leak-check=full --show-leak-kinds=all ./demo_all
==22182== HEAP SUMMARY:
==22182==     in use at exit: 0 bytes in 0 blocks
==22182==   total heap usage: 174 allocs, 174 frees, 145,901 bytes allocated
==22182== All heap blocks were freed -- no leaks are possible
==22182== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

(demo_insert, demo_extract, demo_buildheap, demo_increasekey, demo_heapsort,
 demo_library_priority: same result, 0 errors from 0 contexts, all heap
 blocks freed, on every one of the seven programs)
```

Catatan kejujuran

Kelas MaxHeap pada bab ini DITEMUKAN KEMBALI dari arsip model-jawaban Ujian Akhir asli sendiri, tidak ditulis dari nol -- dikonfirmasi kata demi kata terhadap skeleton yang tercetak di soal ujian itu sendiri dalam Final Exam.pdf. Dua adaptasi kecil yang diungkapkan dibuat semata untuk memenuhi standar zero-warnings g++ -Wall -Wextra -std=c++17 mata kuliah ini dan satu kasus tepi yang nyata: (1) cast static_cast<size_t> eksplisit pada setiap perbandingan int-vs-size_t (tujuh peringatan -Wsign-compare pada versi asli yang tak dimodifikasi; tidak ada hasil perbandingan yang berubah), dan (2) guard satu baris dalam isMaxHeap() untuk array kurang dari 2 elemen, memperbaiki pembacaan di luar batas akibat underflow tak bertanda yang nyata yang sebaliknya akan dilakukan versi asli pada input kosong atau satu elemen -- ditemukan selama QA bab ini sendiri, tidak ada pada input uji ujian asli sendiri. Tidak ada baris lain dari logika algoritma yang ditemukan kembali yang diubah: setiap pertukaran, arah perbandingan, dan tipe/pesan exception cocok persis dengan aslinya. Di luar kedua adaptasi yang diungkapkan ini, pemeriksaan validasi bab ini sendiri tidak menemukan bug lain pada logika heap. valgrind dijalankan terhadap setiap satu dari ketujuh program dan setiap satu jalur melaporkan nol leak dan nol error. Zip kode yang dikirimkan juga diekstrak ulang dan dibangun ulang secara independen dari `make clean && make run` yang bersih untuk mengonfirmasi deliverable yang sebenarnya, bukan sekadar salinan kerja, dikompilasi dan berjalan secara identik.

Referensi

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Binary heap, properti heap-order, Heap Sort, priority queue.)

[2] M.A. Weiss. “Data Structures and Algorithm Analysis in C++.” 4th ed., Pearson, 2014. (Implementasi binary heap, buildHeap, aplikasi priority queue.)

[3] R. Sedgewick, K. Wayne. “Algorithms.” 4th ed., Addison-Wesley, 2011. (Analisis Heap Sort, desain API priority queue.)

[4] P.J. Deitel, H.M. Deitel. “C++ How to Program.” 10th ed., Pearson, 2017. (Penanganan exception C++: std::runtime_error, std::out_of_range, std::invalid_argument.)

[5] cppreference.com. “std::priority_queue”, “Exceptions library.” https://en.cppreference.com/w/cpp (diakses Agustus 2026).

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 15

Hashing & Tabel Hash

Mata kuliah ini telah menghabiskan empat belas bab untuk membuat satu pertanyaan makin cepat dijawab: diberikan sebuah kunci, apakah ia ada di sini, dan jika ya, di mana? Sequential search pada Bab 2 adalah O(n). Binary search memangkasnya menjadi O(log n), dengan syarat data tetap terurut. BST pada Bab 11 juga memberikan O(log n) -- kasus rata-rata, jika pohon tetap cukup seimbang -- sambil menjaga data tetap terurut secara cuma-cuma. Bab ini memperkenalkan langkah terakhir, dan tercepat secara kasus-rata-rata, dalam progresi tersebut: HASHING, yang menjawab pertanyaan yang sama dalam waktu O(1) rata-rata dengan melepaskan sepenuhnya sifat keterurutan. Bab ini juga mengoreksi satu kesalahan penamaan (misnomer) sungguhan yang ditemukan riset proyek ini sendiri dalam materi Ujian Akhir mata kuliah DS yang asli: file model-answer yang secara harfiah bernama Hashing.cpp ternyata sama sekali tidak menggunakan hashing -- ia menggunakan std::map, sebuah pohon pencarian biner seimbang, keluarga struktur yang sama dengan Bab 11. Subbab 15.5 membuat perbedaan itu konkret, dalam kode; contoh kerja bab ini sendiri menggunakan tabel hash sungguhan sepanjang bab, baik yang dibuat tangan (dengan chaining eksplisit) maupun via std::unordered_map.

Tujuan Pembelajaran — setelah bab ini Anda mampu:
(1) Menjelaskan mengapa HASHING dapat menawarkan pencarian O(1) rata-rata di mana searching dan BST tidak bisa, dan menyatakan harga yang dibayarnya (tanpa keterurutan) untuk kecepatan itu. (2) Menyatakan tiga sifat yang dibutuhkan fungsi hash yang BAIK -- deterministik, cepat dihitung, dan menyebarkan kunci secara MERATA ke seluruh bucket -- dan menulis satu, baik untuk kunci integer (hash modulo) maupun kunci string (rolling hash polinomial). (3) Menjelaskan masalah TABRAKAN (collision) secara presisi, dan mengimplementasikan SEPARATE CHAINING (sebuah vector berisi bucket, masing-masing berupa list) sebagai strategi penyelesaian yang sungguh-sungguh bekerja -- insert, find, dan remove, semuanya menangani tabrakan dengan benar. (4) Mendefinisikan LOAD FACTOR (n / kapasitas), menjelaskan mengapa ia harus dijaga terbatas, dan mengimplementasikan REHASHING otomatis yang membesarkan dan membangun ulang tabel begitu load factor melewati ambang batas. (5) Menggunakan std::unordered_map dengan benar, dan menjelaskan SECARA TEPAT mengapa ia berbeda dari std::map -- bukan sekadar nama, tetapi struktur yang mendasarinya (tabel hash vs. BST seimbang), jaminan keterurutan, dan kompleksitas. (6) Menyelesaikan contoh kerja rekonstruksi itinerari -- diberikan kumpulan pasangan (from, to) tak berurutan, temukan titik awal yang unik dan rekonstruksi rute lengkap yang berurutan menggunakan HashMap ditambah HashMap kebalikan (reverse).

15.1 Motivasi: Langkah Berikutnya Setelah Searching dan BST

Setiap bab sejauh ini yang menyentuh "temukan sebuah kunci" membuat sebuah kompromi. Sequential search pada Bab 2 tidak butuh persiapan apa pun sebelumnya, tetapi berbiaya O(n): dalam kasus terburuk, setiap elemen diperiksa. Binary search memangkasnya menjadi O(log n) -- kemenangan dramatis -- tetapi hanya bekerja pada data yang sudah TERURUT, dan menjaga sebuah array tetap terurut saat item baru datang itu sendiri mahal. BST pada Bab 11 menjaga data tetap

terurut DAN mendukung search, insert, dan delete O(log n) sekaligus -- selama pohon tetap cukup seimbang; BST yang dibangun buruk dapat merosot menjadi seperti linked list, O(n) dalam kasus terburuk.

HASHING mengajukan pertanyaan yang berbeda: bagaimana jika pencarian sama sekali tidak perlu membandingkan kunci satu sama lain? Alih-alih mempersempit ruang pencarian lewat perbandingan (apakah kunci ini lebih besar atau lebih kecil dari yang ini?), sebuah TABEL HASH menghitung, langsung dari kunci itu sendiri, kira-kira di mana ia seharusnya berada -- sebuah INDEKS ARRAY -- dan langsung menuju ke sana. Jika dilakukan dengan baik, ini memberikan pencarian O(1) RATA-RATA: bukan sekadar "lebih cepat dari O(log n)" dalam arti konstanta yang lebih kecil, melainkan kurva pertumbuhan yang secara fundamental berbeda, karena jumlah perbandingan yang dibutuhkan sama sekali tidak bertambah seiring n, secara rata-rata.

Struktur (Bab)	Search	Insert	Terurut?	Butuh data terurut/seimbang ?
Sequential search (Bab 2)	O(n)	--	Tidak	Tidak
Binary search (Bab 2)	O(log n)	O(n) (urutkan ulang)	Ya (wajib)	Ya -- harus tetap terurut
BST (Bab 11)	O(log n) rata2, O(n) terburuk	O(log n) rata2	Ya (in-order = terurut)	Cukup seimbang
Tabel hash (Bab 15, bab ini)	O(1) rata2, O(n) terburuk	O(1) rata2	Tidak -- tanpa keterurutan sama sekali	Tidak -- tetapi butuh fungsi hash yang baik + kontrol load factor

"O(1) rata-rata" bukan "O(1) terjamin" -- dan melepaskan keterurutan adalah biaya sungguhan dan permanen

O(1) sebuah tabel hash adalah batas kasus RATA-RATA, bukan jaminan kasus terburuk -- Subbab 15.3 dan 15.4 menunjukkan persis apa yang bisa membuatnya merosot (terlalu banyak tabrakan; load factor terlalu tinggi), dan bagaimana kode bab ini sendiri mendeteksi serta mengoreksi keduanya. Dan kecepatan itu dibeli dengan melepaskan sesuatu yang dijaga BST Bab 11 secara cuma-cuma: tabel hash TIDAK memiliki gagasan berguna tentang "kunci berikutnya secara urutan" -- mengiterasi satu tabel hash memberikan kunci kembali dalam urutan bucket yang pada dasarnya sembarang (Subbab 15.5 menunjukkan ini secara langsung). Memilih tabel hash dibanding BST berarti memilih kecepatan kasus-rata-rata di atas keterurutan; ketika sebuah program sungguh-sungguh butuh pencarian cepat DAN iterasi terurut sekaligus, tidak ada satu pun dari kedua struktur itu sendiri yang menjadi jawaban yang tepat.

15.2 Fungsi Hash

Sebuah FUNGSI HASH mengambil sebuah kunci dan menghasilkan sebuah bilangan bulat -- KODE HASH -- yang kemudian direduksi (biasanya lewat modulo) menjadi indeks array yang valid untuk kapasitas tabel saat ini. Tiga sifat membuat fungsi hash BAIK untuk tujuan ini:

- DETERMINISTIK: kunci yang sama harus selalu menghasilkan kode hash yang sama. Tanpa ini, kunci yang disisipkan pada satu indeks tidak akan pernah bisa ditemukan lagi secara andal.
- CEPAT dihitung: fungsi hash yang berbiaya lebih mahal dari pencarian berbasis perbandingan yang ia gantikan mengalahkan tujuannya sendiri.
- Distribusi MERATA: kunci yang berbeda seharusnya menyebar kira-kira merata ke seluruh bucket yang tersedia, bukan menumpuk di segelintir bucket saja. Fungsi hash yang mengirim setiap kunci ke bucket yang sama secara teknis deterministik dan cepat -- dan sepenuhnya tidak berguna, karena setiap pencarian kemudian merosot menjadi pemindaian penuh satu bucket raksasa.

15.2.1 Hash Berbasis Modulo untuk Kunci Integer

Untuk kunci integer dan tabel berkapasitas capacity, fungsi hash paling sederhana yang bisa dipakai adalah modulo langsung:

```
size_t hashInt(int key, size_t capacity) {
    long long k = key;
    if (k < 0) k = -k;                  // lipat nilai negatif ke rentang positif
    return static_cast<size_t>(k % static_cast<long long>(capacity));
}
```

Ini persis fungsi hash yang digunakan hashtable.h milik bab ini sendiri untuk id buku numerik Pustaka Ceria (Subbab 15.3). Keseragamannya bergantung pada kapasitas: jika kapasitas dipilih secara buruk relatif terhadap pola kunci (misalnya capacity=10 dengan kunci yang semuanya kelipatan 10), setiap kunci bertabrakan di bucket 0 -- salah satu alasan kapasitas tabel secara konvensional dipilih berupa bilangan prima atau pangkat dua dipasangkan dengan hash yang tercampur baik, sebuah penyempurnaan di luar cakupan mata kuliah ini tetapi patut diketahui keberadaannya.

15.2.2 Rolling Hash Polinomial untuk Kunci String

Kode numerik satu karakter saja tidak cukup dengan sendirinya -- "ab" dan "ba" harus menghasilkan hash yang berbeda. Sebuah ROLLING HASH POLINOMIAL memperlakukan sebuah string sebagai bilangan basis-B, satu karakter per digit:

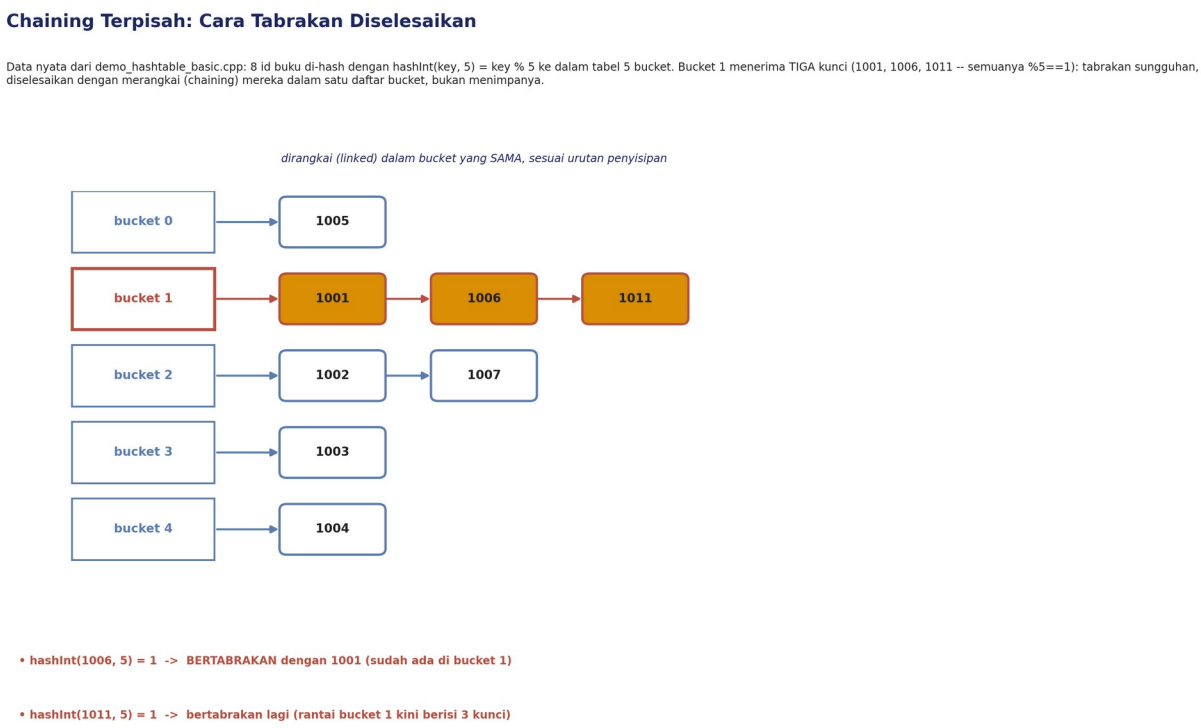
```
size_t hashString(const std::string &s, size_t capacity) {
    unsigned long long h = 0;
    for (unsigned char c : s) {
        h = h * 31ULL + static_cast<unsigned long long>(c);
    }
    return static_cast<size_t>(h % static_cast<unsigned long long>(capacity));
}
```

31 adalah bilangan prima ganjil kecil -- konstanta yang sama yang digunakan String.hashCode() milik Java -- dipilih karena mengalikannya mencampur kontribusi setiap karakter baru ke banyak bit, bukan sekadar menggesernya, menyebarkan string yang mirip ("Cianjur" vs. "Cianjura") ke bucket yang berbeda jauh lebih merata dibandingkan, katakanlah, sekadar menjumlahkan kode karakter. Inilah fungsi hash yang digunakan contoh kerja itinerari bab ini (Subbab 15.6) untuk kunci nama-kotanya.

15.3 Penyelesaian Tabrakan: Separate Chaining

Sebagus apa pun sebuah fungsi hash, dua kunci yang BERBEDA tetap bisa menghasilkan indeks bucket yang SAMA begitu kapasitas terbatas -- ini disebut TABRAKAN (collision), dan ini bukan bug atau tanda fungsi hash yang buruk; ini adalah kepastian matematis begitu jumlah kemungkinan kunci melebihi jumlah bucket (PRINSIP PIGEONHOLE). Sebuah tabel hash harus memiliki strategi yang jelas dan benar untuk apa yang terjadi ketika itu terjadi.

Bab ini mengimplementasikan SEPARATE CHAINING: setiap bucket bukanlah satu slot tunggal melainkan sebuah LIST kecil, dan setiap kunci yang menghasilkan bucket tersebut ditambahkan ke listnya. Dua kunci yang bertabrakan cukup hidup berdampingan dalam rantai bucket yang sama, alih-alih satu menimpa yang lain.



Gambar 15.1 — separate chaining: data nyata demo_hashtable_basic.cpp sendiri, 8 id buku di-hash dengan key % 5 ke dalam tabel 5 bucket. Bucket 1 menerima id 1001, 1006, dan 1011 (semuanya %5==1) -- tabrakan 3-arah sungguhan, diselesaikan dengan merangkai ketiganya dalam satu daftar bucket.

```
template <typename K, typename V>
class HashTable {
    ...
    void insert(const K &key, const V &value) {
        size_t idx = bucketIndex(key, buckets.size());
        for (auto &entry : buckets[idx]) {
            if (entry.first == key) { entry.second = value; return; }  // update
        }
        buckets[idx].emplace_back(key, value);   // tambahkan ke rantai
        ++numEntries;
        if (loadFactor() > maxLoadFactor) rehash(buckets.size() * 2);   //
Subbab 15.4
    }

    bool find(const K &key, V &outValue) const {
        size_t idx = bucketIndex(key, buckets.size());
        for (const auto &entry : buckets[idx]) {
            if (entry.first == key) { outValue = entry.second; return true; }
        }
        return false;
    }
    ...
};
```

Perhatikan bahwa insert() memeriksa SELURUH rantai untuk kunci yang cocok yang sudah ada sebelum menambahkan -- menyisipkan kunci yang sudah ada MEMPERBARUI nilainya, bukan membuat entri duplikat. find() dan remove() (tidak ditampilkan, tetapi ada di hashtable.h) menelusuri rantai yang sama, membandingkan kunci satu per satu -- sehingga biaya nyata sebuah pencarian adalah O(1 + panjang rantai): O(1) untuk menghitung hash dan melompat ke bucket, ditambah berapa pun jumlah entri yang kebetulan dipegang rantai bucket tersebut.

Transkrip nyata dan telah dijalankan dari `demo_hashtable_basic.cpp`, menggunakan tabel 5-bucket yang sengaja dibuat kecil sehingga tabrakan di atas dipaksa terjadi dan terlihat:

```
=== Inserting 8 books into a 5-bucket hash table ===

insert(1001) -> hashInt(1001, 5) = 1  (bucket had 0 entries before)
insert(1002) -> hashInt(1002, 5) = 2  (bucket had 0 entries before)
insert(1003) -> hashInt(1003, 5) = 3  (bucket had 0 entries before)
insert(1004) -> hashInt(1004, 5) = 4  (bucket had 0 entries before)
insert(1005) -> hashInt(1005, 5) = 0  (bucket had 0 entries before)
insert(1006) -> hashInt(1006, 5) = 1  (bucket had 1 entry before)
insert(1007) -> hashInt(1007, 5) = 2  (bucket had 1 entry before)
insert(1011) -> hashInt(1011, 5) = 1  (bucket had 2 entries before)

=== Bucket contents after all inserts (capacity=5) ===
bucket[0]: 1 entry     bucket[1]: 3 entries     bucket[2]: 2 entries
bucket[3]: 1 entry     bucket[4]: 1 entry

size() = 8   loadFactor() = 1.60   longestChain() = 3   collisions so far = 3

=== find() ===
find(1003) -> FOUND : B1003 ("Cantik Itu Luka" by Eka Kurniawan)
find(9999) -> NOT FOUND (correct: 9999 was never inserted)

=== remove() ===
remove(1006) -> removed   size() now = 7
find(1006) after removal -> NOT FOUND (correct)
find(1001) after 1006 removed from the SAME bucket -> FOUND : Laskar Pelangi
```

Baris terakhir di atas adalah pemeriksaan kebenaran yang penting: menghapus 1006 TIDAK mengganggu 1001, tetangganya dalam rantai bucket yang persis sama -- penghapusan berbasis list pada separate chaining hanya pernah menyentuh satu node yang kuncinya benar-benar cocok.

> **Open addressing (linear probing) -- strategi klasik lainnya, singkat**
> Bab ini mengimplementasikan chaining karena ia strategi pertama yang lebih umum diajarkan dan lebih toleran saat load factor naik -- sebuah rantai selalu bisa tumbuh. Keluarga klasik lainnya, OPEN ADDRESSING, menyimpan setiap entri langsung di dalam array bucket itu sendiri (tanpa list terpisah): pada tabrakan di indeks i, LINEAR PROBING cukup mencoba i+1, i+2, i+3, ... sampai slot kosong ditemukan. Ia memakai memori lebih sedikit per entri (tanpa overhead node list) tetapi lebih sensitif terhadap load factor -- seiring tabel makin penuh, urutan probe makin panjang, dan penghapusan lebih rumit (penghapusan naif dapat merusak rantai probe untuk entri berikutnya, biasanya diperbaiki dengan penanda khusus "deleted"). Mata kuliah ini mengimplementasikan chaining secara penuh; open addressing patut diketahui keberadaannya, dan menjadi topik lanjutan yang wajar jika Anda melanjutkan setelah mata kuliah ini.

15.4 Load Factor dan Rehashing

LOAD FACTOR sebuah tabel hash adalah size() / capacity() -- kira-kira, panjang rantai rata-rata yang seharusnya diharapkan ditelusuri sebuah pencarian. Load factor yang mendekati atau di bawah 1.0 menjaga rantai tetap pendek dan find()/insert() dekat dengan janji kasus-rata-rata O(1)-nya; load factor yang terus naik (makin banyak entri dijejalkan ke jumlah bucket tetap yang sama) membuat setiap rantai makin panjang, dan biaya pencarian rata-rata merangkak menuju O(n) -- persis linked list biasa yang dipelajari mata kuliah ini di Bab 7, hanya mengenakan pakaian tabel hash.

Perbaikannya adalah REHASHING: begitu load factor akan melebihi ambang batas yang dipilih (default bab ini: 0.75), kapasitas tabel digandakan, dan -- yang krusial -- setiap entri yang ada DISISIPKAN ULANG, karena bucket mana yang dimiliki sebuah kunci bergantung pada kapasitas. Inilah mengapa ia disebut "rehashing", bukan sekadar "resizing": setiap kunci bisa berpindah.

```
void rehash(size_t newCapacity) {
    std::vector<std::list<std::pair<K, V>>> old = std::move(buckets);
    buckets.assign(newCapacity, {});
    numEntries = 0;
    for (auto &chain : old)
        for (auto &entry : chain) {
            size_t idx = bucketIndex(entry.first, buckets.size());   // dihitung
ulang!
            buckets[idx].emplace_back(std::move(entry));
            ++numEntries;
        }
    ++rehashCount;
}
```

Transkrip nyata dan telah dijalankan dari `demo_rehash.cpp`, menyisipkan kunci 1..20 ke dalam tabel yang dimulai pada kapasitas 4:

```
=== Inserting keys 1..20, watching load factor and capacity ===
(maxLoadFactor = 0.75; capacity doubles whenever loadFactor() would exceed it)

insert( 1) -> size= 1 capacity= 4 loadFactor=0.250
insert( 2) -> size= 2 capacity= 4 loadFactor=0.500
insert( 3) -> size= 3 capacity= 4 loadFactor=0.750
insert( 4) -> size= 4 capacity= 8 loadFactor=0.500   <-- REHASHED (capacity
doubled)
insert( 5) -> size= 5 capacity= 8 loadFactor=0.625
insert( 6) -> size= 6 capacity= 8 loadFactor=0.750
insert( 7) -> size= 7 capacity=16 loadFactor=0.438   <-- REHASHED (capacity
doubled)
  ... (8 sampai 12 mengisi hingga loadFactor=0.750 pada kapasitas 16) ...
insert(13) -> size=13 capacity=32 loadFactor=0.406   <-- REHASHED (capacity
doubled)
  ... (14 sampai 20 menetap pada loadFactor=0.625, kapasitas 32) ...

Total rehashes triggered: 3
Final capacity: 32   final size: 20   final loadFactor: 0.625

=== Correctness survives every rehash: every key 1..20 must still be findable
===
Keys checked: 20   mismatches: 0   ALL 20 KEYS SURVIVED EVERY REHASH INTACT

=== Without ever rehashing (capacity fixed at 4), the same 20 keys would give
===
capacity=4 (never grew)   loadFactor=5.00   longestChain=5
```

Rehashing berbiaya O(n) sekali, ditukar dengan O(1) rata-rata selamanya sesudahnya

Satu kali rehash menyentuh setiap entri yang ada -- kerja O(n), sekaligus. Tetapi itu hanya terjadi O(log n) kali total seiring tabel tumbuh dari kosong hingga n entri (setiap rehash menggandakan kapasitas), sehingga biaya TOTAL seluruh rehashing, disebar (diamortisasi) ke n penyisipan, tetap rata-rata O(1) per insert -- argumen "kadang mahal, tetapi cukup jarang hingga tak berpengaruh secara rata-rata" yang sama yang diandalkan std::vector yang tumbuh untuk push_back()-nya. Alternatifnya -- tidak pernah rehash, seperti ditunjukkan tabel berkapasitas-tetap-4 di atas -- membiarkan load factor tumbuh tanpa batas, dan biaya pencarian rata-rata merosot bersamanya.

15.5 std::unordered_map — Tabel Hash Sungguhan Milik STL Sendiri

Semua yang dibangun tangan oleh Subbab 15.2-15.4, pustaka standar sudah menyediakannya, sudah disetel dan diuji: std::unordered_map<K, V> ADALAH tabel hash sungguhan (separate chaining secara internal, di setiap implementasi utama), dengan bucket_count() dan load_factor()-nya sendiri yang terekspos langsung.

std::map BUKAN hashing -- ia keluarga BST milik Bab 11, apa pun kesan yang diberikan namanya

Inilah persis koreksi yang menjadi alasan bab ini ada. std::map<K, V> adalah POHON PENCARIAN BINER SEIMBANG (red-black tree di setiap pustaka standar C++ utama) -- keluarga struktural yang sama dengan Bab 11, memberikan search/insert/erase O(log n) kasus terburuk, dan SELALU mengiterasi dalam urutan kunci terurut, karena itulah yang secara alami diberikan BST secara cuma-cuma. std::unordered_map<K, V> adalah tabel hash SUNGGUHAN -- O(1) kasus RATA-RATA (O(n) hanya pada kasus terburuk yang jarang dari banyak tabrakan), dan TIDAK ADA jaminan keterurutan sama sekali saat diiterasi. File model-answer Ujian Akhir mata kuliah DS yang asli, Hashing.cpp, menggunakan std::map untuk menyelesaikan tugas rekonstruksi-itinerarinya -- padahal tugas tersebut secara eksplisit dinamai "Hashing" dan mensyaratkan "metode hashing." Ini adalah kesalahan penamaan sungguhan dalam materi asli tersebut, bukan pilihan pengajaran yang disengaja, dan bab ini mengoreksinya secara langsung: contoh kerja Subbab 15.6 menggunakan std::unordered_map (dan HashTable buatan tangan bab ini sendiri) sebagai gantinya, dengan ketidaksesuaian tersebut diungkapkan secara jujur, bukan diam-diam direproduksi -- lihat callout DEF di akhir Subbab 15.6.

`demo_unordered_map.cpp` membangun data buku Pustaka Ceria YANG SAMA ke dalam kedua wadah, disisipkan dalam urutan kunci acak yang SAMA (1005, 1001, 1003, 1002, 1004), dan mencetak urutan iterasi masing-masing -- transkrip nyata dan telah dijalankannya:

```
=== Inserted in shuffled key order: 1005, 1001, 1003, 1002, 1004 ===

std::map<int,Book> iteration order (a balanced BST -- ALWAYS sorted by key):
  1001 -> Laskar Pelangi
  1002 -> Bumi Manusia
  1003 -> Cantik Itu Luka
  1004 -> Filosofi Kopi
  1005 -> Negeri 5 Menara

std::unordered_map<int,Book> iteration order (a real hash table -- NO ordering
guarantee; order depends on each key's hash and bucket, not insertion or value):
  1004 -> Filosofi Kopi
  1002 -> Bumi Manusia
  1003 -> Cantik Itu Luka
  1001 -> Laskar Pelangi
  1005 -> Negeri 5 Menara

byHash.bucket_count() = 13   byHash.load_factor() = 0.385
(std::map exposes no bucket_count()/load_factor() at all -- there is nothing to
ask for)
```

std::map selalu kembali terurut, tanpa diminta; std::unordered_map kembali dalam urutan yang sama sekali tidak berkaitan dengan urutan penyisipan maupun nilai -- persis apa arti "tanpa jaminan keterurutan" dalam praktik, bukan hanya dalam teori.

15.6 Contoh Kerja: Merekonstruksi Itinerari dengan HashMap

Contoh kerja subbab ini, secara sengaja, adalah persis tugas yang diminta Ujian Akhir mata kuliah DS yang asli sendiri (Tugas 2, 50 poin): diberikan kumpulan pasangan (from, to) leg penerbangan TIDAK

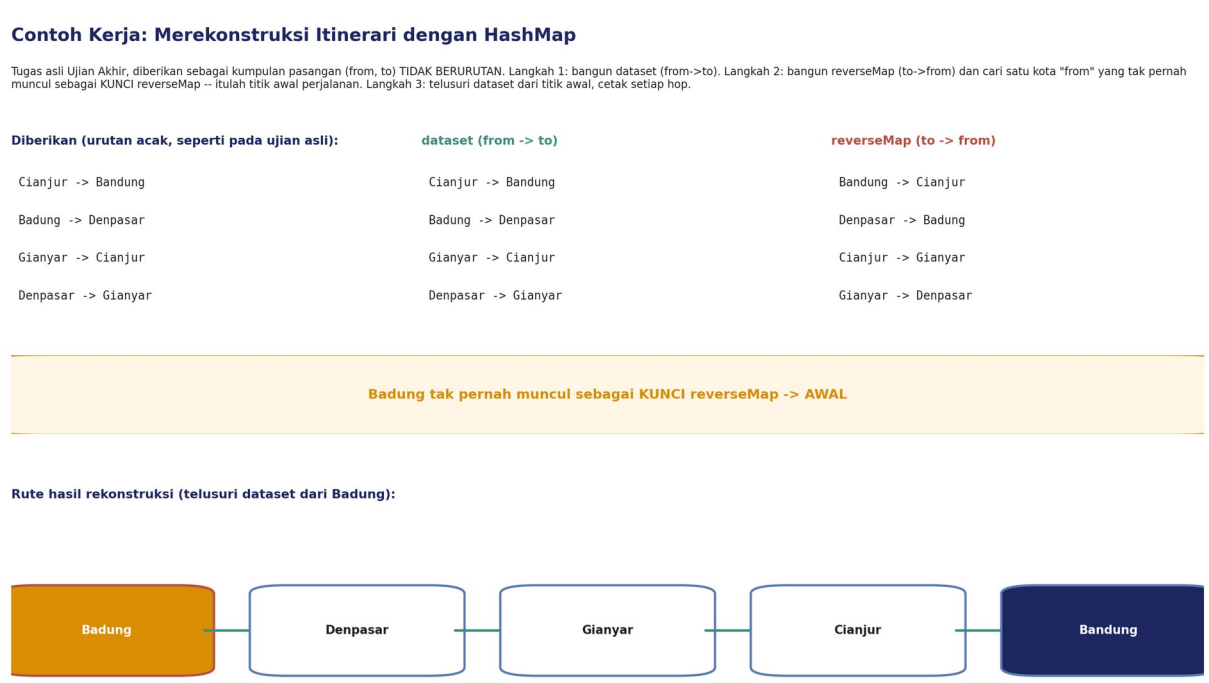
BERURUTAN yang menggambarkan satu perjalanan berkesinambungan dengan titik awal dan titik akhir yang unik, rekonstruksi rute lengkap yang berurutan.

Algoritma milik ujian itu sendiri, dalam tiga langkah:

1. Bangun sebuah HashMap dataset dari setiap leg, from -> to.
2. Bangun HashMap KEBALIKAN reverseMap, to -> from. Pindai kunci-kunci dataset (kota-kota "from"); yang SATU yang tak pernah muncul sebagai KUNCI reverseMap adalah kota yang tak pernah didatangi siapa pun -- titik awal perjalanan.
3. Mulai dari sana, telusuri dataset -- start, dataset[start], dataset[dataset[start]], ... -- mencetak setiap hop, hingga sebuah kota tidak memiliki leg keluar lagi.

Contoh kerja milik ujian itu sendiri, diberikan dalam urutan acak ini:

```
Cianjur  -> Bandung
Badung   -> Denpasar
Gianyar  -> Cianjur
Denpasar -> Gianyar
```



Gambar 15.2 — contoh kerja secara lengkap: input acak, dataset (from->to), reverseMap (to->from), identifikasi titik awal (Badung tak pernah muncul sebagai kunci reverseMap), dan rute hasil rekonstruksi akhir.

15.6.1 Buatan Tangan: HashTable<string,string>

`demo_itinerary_handrolled.cpp` menyelesaikan ini menggunakan HashTable chaining milik bab ini sendiri (Subbab 15.3), dengan kunci std::string nama kota lewat hashString() (Subbab 15.2.2):

```
=== Given legs, in the exam's own shuffled order ===
  Cianjur -> Bandung
  Badung -> Denpasar
  Gianyar -> Cianjur
  Denpasar -> Gianyar

=== Starting point found: "Badung" (never appears as a "to") ===

=== Reconstructed itinerary ===
Badung -> Denpasar, Denpasar -> Gianyar, Gianyar -> Cianjur, Cianjur -> Bandung

=== Full route, in order ===
  Badung -> Denpasar -> Gianyar -> Cianjur -> Bandung

Matches expected route (Badung -> Denpasar -> Gianyar -> Cianjur -> Bandung)?
YES
```

15.6.2 Versi std::unordered_map — Dikoreksi dari Model Answer Ujian Asli

`demo_itinerary_unordered_map.cpp` menyelesaikan tugas yang identik menggunakan std::unordered_map<string,string> sebagai gantinya -- hashing sungguhan, sesuai nama tugasnya sendiri -- dengan loop penelusuran-dan-cetak ditulis menggunakan flag bool yang diinisialisasi dengan benar, bukan int yang tidak diinisialisasi seperti pada model answer asli:

```
auto it = dataset.find(start);
bool firstHop = true;
while (it != dataset.end()) {
    if (!firstHop) std::printf(", ");
    std::printf("%s -> %s", it->first.c_str(), it->second.c_str());
    firstHop = false;
    it = dataset.find(it->second);
}
```

Transkrip nyata dan telah dijalankannya, mengonfirmasi baik pilihan wadah yang dikoreksi maupun loop yang dikoreksi:

```
The dataset before reconstruction (unordered_map -- iteration order is
NOT the insertion order, and is not guaranteed to be alphabetical either):
  Gianyar -> Cianjur
  Badung -> Denpasar
  Denpasar -> Gianyar
  Cianjur -> Bandung

The itinerary after reconstruction:
  Badung -> Denpasar, Denpasar -> Gianyar, Gianyar -> Cianjur, Cianjur ->
Bandung

bucket_count()=13 load_factor()=0.308 -- this really is a hash table
```

Catatan kejujuran — dua masalah sungguhan ditemukan dalam file model-answer ujian asli, diungkapkan, bukan diam-diam direproduksi

File model-answer Ujian Akhir asli sendiri, Hashing.cpp, ditemukan kembali langsung dari arsip Final Exam - Model Answer.zip dan diperiksa baris demi baris. File itu memiliki DUA masalah sungguhan yang telah dikonfirmasi: (1) KESALAHAN PENAMAAN -- meski tugas tersebut dinamai "Hashing" dan secara eksplisit mensyaratkan "metode hashing," file itu menggunakan std::map sepanjang isinya, yang merupakan BST seimbang, sama sekali bukan hashing (callout TRAP Subbab 15.5). (2) BUG -- file itu mendeklarasikan `int a;` lalu membacanya sebelum pernah memberinya nilai, di dalam kondisi `if (i != dataset.end() && a != 0)`, diikuti `a++` -- pembacaan variabel tak terinisialisasi buku teks, undefined behaviour dalam C++. Sesuai kebijakan tetap proyek ini untuk tidak pernah diam-diam mereproduksi kesalahan yang diketahui dari materi mata kuliah asli, kedua program itinerari bab ini mengoreksi KEDUA masalah tersebut secara langsung -- menggunakan std::unordered_map (atau tabel hash buatan tangan bab ini sendiri) menggantikan std::map, dan flag bool yang diinisialisasi dengan benar menggantikan int yang tak terinisialisasi -- sambil menjaga algoritma yang persis sama dan contoh kerja yang persis sama (kota yang sama, urutan input acak yang sama, rute yang diharapkan sama) demi kesinambungan penuh dengan ujian asli.

15.7 Kaitan Aplikasi: Pencarian Buku Pustaka Ceria

Katalog Pustaka Ceria kini telah dicari dengan empat cara berbeda sepanjang mata kuliah ini: Sequential/Binary search Bab 2 atas sebuah array biasa, BST Bab 11 dengan kunci id buku, dan tabel hash bab ini (Subbab 15.3), juga dengan kunci id buku. Mencari sebuah Book berdasarkan id-nya yang persis -- operasi katalog paling umum -- adalah persis kasus yang menjadi tujuan hashing dibangun: tidak dibutuhkan keterurutan, hanya "apakah id yang persis ini ada, dan jika ya, apa recordnya?"

Struktur	Pencarian berdasarkan id persis	Juga mendukung "semua buku antara id X dan Y"?
Bab 2: Binary search (array terurut)	O(log n)	Ya -- O(log n) untuk menemukan batas, lalu pindai
Bab 11: BST	O(log n) rata2	Ya -- traversal in-order dalam sebuah rentang
Bab 15: Tabel hash (bab ini)	O(1) rata2 -- tercepat dari ketiganya	Tidak -- tidak ada keterurutan untuk memindai sebuah rentang

Inilah kompromi jujur yang sudah dipratinjau Subbab 15.1: untuk pencarian exact-match saja, hashing menang. Begitu sebuah query membutuhkan RENTANG ("semua buku dengan id antara 1000 dan 1010") atau URUTAN ("buku-buku dalam urutan id"), O(log n) milik BST dengan keterurutan cuma-cuma kembali menjadi pilihan yang lebih baik -- tidak ada satu struktur pun yang sekadar "lebih baik"; masing-masing cocok untuk bentuk query yang berbeda.

15.8 Ringkasan Bab dan Poin Penting

- HASHING menukar keterurutan dengan pencarian O(1) kasus RATA-RATA -- langkah tercepat dalam progresi pencarian mata kuliah ini (Sequential O(n) -> Binary/BST O(log n) -> Tabel hash O(1) rata2), dengan biaya tanpa urutan iterasi yang bermakna dan tanpa jaminan kasus terburuk.
- Fungsi hash yang BAIK bersifat deterministik, cepat, dan menyebarkan kunci secara MERATA ke seluruh bucket -- bab ini mengimplementasikan hashInt() (modulo, untuk kunci integer) dan hashString() (rolling hash polinomial basis-31, untuk kunci string).
- TABRAKAN adalah kepastian matematis (prinsip pigeonhole), bukan bug. Bab ini mengimplementasikan SEPARATE CHAINING -- setiap bucket adalah sebuah list, dan kunci yang bertabrakan hidup berdampingan di dalamnya -- sebagai strategi penyelesaian yang sungguh-sungguh bekerja; OPEN ADDRESSING (linear probing) dijelaskan secara konseptual sebagai pendekatan klasik lainnya.
- LOAD FACTOR (n/kapasitas) mengukur seberapa penuh sebuah tabel; membiarkannya tumbuh tanpa batas merosotkan pencarian rata-rata menuju O(n). REHASHING -- menggandakan kapasitas dan menyisipkan ulang setiap kunci begitu load factor melewati ambang batas (0.75 secara default) -- menjaga O(1) kasus-rata-rata tetap utuh seiring tabel tumbuh.
- std::unordered_map ADALAH tabel hash sungguhan (O(1) rata-rata, tanpa jaminan keterurutan); std::map adalah keluarga BST Bab 11 (O(log n) kasus terburuk, iterasi SELALU terurut) -- keduanya TIDAK saling menggantikan meski nama std::map terdengar terkait hash, dan bab ini mengoreksi kesalahan penamaan sungguhan yang ditemukan dalam model answer ujian asli yang menggunakan std::map untuk tugas yang secara eksplisit dinamai "Hashing."
- Contoh kerja rekonstruksi-itinerari (Badung -> Denpasar -> Gianyar -> Cianjur -> Bandung) -- Tugas 2 milik Ujian Akhir asli sendiri -- diselesaikan dengan dua cara: HashTable<string,string> chaining buatan tangan, dan std::unordered_map<string,string>, dengan bug variabel-tak-terinisialisasi milik model answer ujian asli dikoreksi dan diungkapkan, bukan diam-diam direproduksi.
- Semua 6 program dalam bab ini (demo_hashtable_basic, demo_rehash, demo_unordered_map, demo_itinerary_handrolled, demo_itinerary_unordered_map, demo_all) sungguh-sungguh dikompilasi tanpa peringatan, sungguh-sungguh dijalankan, dan sungguh-sungguh diperiksa dengan valgrind -- 0 error, 0 leak, pada setiap binary.

15.9 Soal Latihan

Soal-soal ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Menggunakan hashInt() milik bab ini sendiri, hitung hashInt(1013, 5) dan hashInt(1018, 5) dengan tangan. Apakah keduanya bertabrakan dengan kunci mana pun yang sudah ada di tabel Gambar 15.1? Jelaskan.

2. Jelaskan, dengan kata-kata Anda sendiri, mengapa "O(1) kasus rata-rata" adalah klaim yang berbeda (lebih lemah) dari "O(1) kasus terburuk," dan gambarkan sebuah urutan penyisipan konkret yang akan membuat tabel hash bab ini sendiri untuk sementara berperilaku seperti linked list O(n) (demonstrasi kapasitas-tetap Subbab 15.4 adalah satu contoh -- gambarkan yang lain).
3. Telusuri hashString("ab", 100) dan hashString("ba", 100) dengan tangan menggunakan rumus basis-31 pada Subbab 15.2.2, dan konfirmasi keduanya berbeda (seperti seharusnya, atau fungsi hash itu akan rusak parah).
4. Sebuah tabel dengan kapasitas 8 dan maxLoadFactor 0.75 saat ini memegang 6 entri. Apakah menyisipkan satu entri lagi akan memicu rehash? Berapa kapasitas barunya sesudahnya?
5. Jelaskan secara presisi mengapa iterasi std::map SELALU terurut sementara std::unordered_map tidak, merujuk pada apa sesungguhnya masing-masing wadah itu secara internal (red-black tree vs. tabel hash).
6. Menggunakan deskripsi 3-langkah algoritma itinerari itu sendiri (Subbab 15.6), telusuri DENGAN TANGAN pada kumpulan leg yang berbeda ini: Solo->Yogyakarta, Yogyakarta->Semarang, Malang->Solo. Nyatakan rute hasil rekonstruksinya.
7. Hashing.cpp milik ujian asli sendiri menggunakan `int a;` tak terinisialisasi sebagai flag pemisah koma. Jelaskan, dengan kata-kata Anda sendiri, mengapa membaca variabel lokal yang tak terinisialisasi adalah undefined behaviour dalam C++, dan mengapa fakta bahwa sebuah eksekusi tertentu mungkin kebetulan "berhasil" tidak membuat kode itu benar.

15.10 Lampiran 15.A — Log Validasi

Keenam program di bawah ini (`demo_hashtable_basic.cpp`, `demo_rehash.cpp`, `demo_unordered_map.cpp`, `demo_itinerary_handrolled.cpp`, `demo_itinerary_unordered_map.cpp`, `demo_all.cpp`) dikompilasi dengan `g++ -Wall -Wextra -std=c++17` (nol peringatan dari keenamnya) dan sungguh-sungguh dieksekusi; transkrip di bawah ini adalah keluaran terminal nyata dan tak diedit. `valgrind --leak-check=full --show-leak-kinds=all` sungguh-sungguh dijalankan terhadap keenam program, sesuai kebijakan tetap mata kuliah ini sejak Bab 9.

15.A.1 demo_all.cpp (transkrip lengkap)

```
$ g++ -Wall -Wextra -std=c++17 -o demo_all demo_all.cpp
$ ./demo_all
=== Chapter 15 Validation Summary: Hashing & Hash Tables ===

[HashTable insert()/find(): every inserted key is findable]  3/3 OK
[Collision handling: forced collisions in a small-capacity table resolve
correctly]  7/7 OK
[update-in-place and remove(): size accounting stays correct]  6/6 OK
[Load factor + rehashing: capacity grows, ALL keys survive every rehash]  4/4 OK
[std::string-keyed table: hashString() based lookups]  2/2 OK
[std::map vs std::unordered_map: same data, ordering differs as documented]  3/3
OK
[Itinerary reconstruction: exact expected route, both implementations]  3/3 OK

Overall: 28 / 28 checks passed -- ALL CHECKS PASSED
```

15.A.2 valgrind (keenam program)

```
$ valgrind --leak-check=full --show-leak-kinds=all ./demo_all
==23140== HEAP SUMMARY:
==23140==     in use at exit: 0 bytes in 0 blocks
==23140==   total heap usage: 2,021 allocs, 2,021 frees, 210,896 bytes allocated
==23140== All heap blocks were freed -- no leaks are possible
==23140== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

(demo_hashtable_basic, demo_rehash, demo_unordered_map,
demo_itinerary_handrolled,
 demo_itinerary_unordered_map: hasil sama pada setiap program -- 0 errors from 0
contexts,
 all heap blocks freed, pada keenam program.)
```

Catatan kejujuran

HashTable<K,V> (hashtable.h) dan kedua fungsi hash (hashInt, hashString) bab ini disusun baru untuk mata kuliah ini, karena tidak ada kuliah atau kerangka ujian yang pernah menyediakannya -- riset file rencana proyek ini sendiri mengonfirmasi slot kuliah hashing tidak pernah ada di mana pun dalam materi mentah. DATA contoh kerja itinerari (rute Badung -> Denpasar -> Gianyar -> Cianjur -> Bandung, dan urutan input acak yang persis) direproduksi secara setia dari deskripsi tugas tercetak Ujian Akhir asli sendiri dan file model-answer-nya yang ditemukan kembali, Hashing.cpp -- dikonfirmasi langsung dari arsip yang ditemukan kembali, bukan direkonstruksi dari ingatan. Dua koreksi sungguhan dan diungkapkan dibuat pada pendekatan file tersebut (lihat callout DEF Subbab 15.6): std::map digantikan std::unordered_map/HashTable bab ini sendiri (mengoreksi kesalahan penamaan "Hashing lewat BST"), dan `int a` yang tak terinisialisasi digantikan flag bool yang diinisialisasi dengan benar (mengoreksi bug undefined-behaviour sungguhan). Tidak ada aspek lain dari tugas asli atau keluaran yang diharapkannya yang diubah. Di luar dua koreksi yang disengaja dan diungkapkan ini, pemeriksaan validasi bab ini sendiri tidak menemukan bug lain. valgrind dijalankan terhadap setiap satu dari keenam program dan setiap eksekusi melaporkan nol leak dan nol error. Zip kode yang dikirim diekstrak ulang secara independen ke direktori bersih dan dibangun ulang dari awal (`make clean && make run`) untuk mengonfirmasi deliverable yang sesungguhnya -- bukan sekadar salinan kerja -- terkompilasi dan berjalan secara identik.

Referensi

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Tabel hash, fungsi hash, chaining, open addressing, universal hashing.)

[2] M.A. Weiss. “Data Structures and Algorithm Analysis in C++.” 4th ed., Pearson, 2014. (Implementasi tabel hash, separate chaining, load factor, rehashing.)

[3] R. Sedgewick, K. Wayne. “Algorithms.” 4th ed., Addison-Wesley, 2011. (Hashing, strategi penyelesaian tabrakan, aplikasi symbol table.)

[4] cppreference.com. “std::unordered_map”, “std::map”, “std::hash.” https://en.cppreference.com/w/cpp (diakses Agustus 2026).

[5] EF234201 Struktur Data, Ujian Akhir (semester 2023/2024-2), Tugas 2 ("Hashing", 50 poin) dan file model-answer-nya Hashing.cpp — sumber contoh kerja itinerari bab ini dan kesalahan penamaan/bug yang dikoreksi bab ini.

Profitina

profitina.com

BUKU AJAR · STRUKTUR DATA (EF234201)

Struktur Data

Array, List, Stack, Queue, Tree, Graph, Heap, dan Hashing dalam C/C++

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 16 · BAB LATIHAN · BAB TERAKHIR MATA KULIAH INI

Ujian Akhir Semester (UAS)

Tidak ada materi baru di sini — ujian coding individual, open, take-home yang mencakup Bab 14 (Heap & Priority Queue) dan Bab 15 (Hashing & Hash Table), dua topik penutup mata kuliah ini. Bab ini TIDAK menyertakan subfolder code/ — kedua pengumpulan di bawah (ditambah laporan tertulis Anda) adalah pekerjaan Anda sendiri untuk ditulis dan dikumpulkan. Ini juga bab terakhir dari keseluruhan 16 bab mata kuliah Struktur Data.

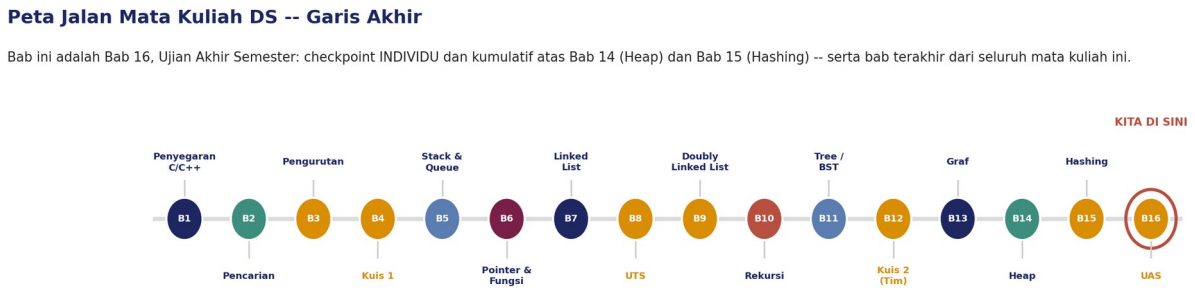
Tujuan Pembelajaran — setelah mengerjakan bab ini Anda akan mampu:
(1) Melengkapi skeleton MaxHeap milik Bab 14 dengan menulis lima fungsi anggota spesifik — extractMax, increaseKey, getMin, buildMaxHeap, isMaxHeap — masing-masing bernilai 10 dari 50 poin heap pada ujian ini. (2) Menyelesaikan soal rekonstruksi itinerary menggunakan hash table SESUNGGUHNYA (unordered_map milik Bab 15, bukan std::map yang menyamar sebagai satu), ditambah reverse map, dengan benar mengidentifikasi titik awal unik perjalanan dan merangkai from->to untuk mencetak seluruh rute. (3) Menulis laporan teknis singkat — metode, kode sumber, hasil eksekusi dan analisis, kesimpulan — yang mendokumentasikan solusi Tugas 2 Anda, sambil memahami bahwa laporan ini memiliki bobot lebih besar di sini dibanding asesmen mana pun sebelumnya dalam mata kuliah ini. (4) Mengenali bahwa setiap subtugas dalam ujian ini adalah penerapan langsung dan terungkap dari sesuatu yang sudah diajarkan dan didemonstrasikan bekerja oleh Bab 14 atau Bab 15 — tidak ada teori baru di bab ini, hanya integrasi dan eksekusi mandiri.

16.1 Ulasan: Bab 14-15 Secara Singkat

Bab 14 membangun MaxHeap<int> berbasis array yang lengkap dari dasar: heapifyUp/heapifyDown (dua primitif privat yang menjadi dasar semua operasi lain), insert, deleteRoot, getMax, heapifyArray (pembangunan bottom-up O(n)), heapSort (menghubungkan kembali ke ulasan pengurutan Bab 3), dan printHeap. Di atas dasar itu, Bab 14 juga mengimplementasikan — dan benar-benar mengompilasi, menjalankan, serta memverifikasinya dengan valgrind — lima fungsi tambahan yang kini diminta ujian ini untuk ANDA tulis sendiri: extractMax (hapus-dan-kembalikan dengan penanganan exception yang tepat), increaseKey (naikkan sebuah key, lalu pulihkan urutan heap ke atas), getMin (cari nilai minimum tanpa menghapusnya, memanfaatkan fakta bahwa nilai minimum max-heap selalu berada di daun), buildMaxHeap (secara fungsional identik dengan heapifyArray, dibangun langsung dari array tak terurut), dan isMaxHeap (pemeriksaan validitas pada array sembarang, tanpa menyentuh apa pun).

Bab 15 mengoreksi sebuah kesalahan penamaan (misnomer) yang sesungguhnya terjadi dalam sejarah mata kuliah ini sendiri: file jawaban model dari Ujian Akhir yang asli bernama Hashing.cpp, namun menyelesaikan soal rekonstruksi itinerary-nya dengan std::map — sebuah balanced binary search tree, bukan hash table. Bab 15 justru mengajarkan hashing yang sesungguhnya: fungsi hash, resolusi collision (chaining dan open addressing), load factor dan rehashing, serta std::unordered_map, menerapkan semuanya pada contoh kerja rekonstruksi-itinerary yang SAMA

yang dipakai ulang ujian ini — sekumpulan leg (from, to) yang direkonstruksi menjadi satu rute terurut melalui HashMap ditambah reverse HashMap yang menemukan titik awal yang unik.



Gambar 16.1 — Bab ini berada di Bab 16, slot terakhir dari enam belas bab peta jalan DS: checkpoint terakhir mata kuliah ini, menutup Bab 14 (Heap) dan Bab 15 (Hashing).

Tidak ada yang baru di bawah ini — ujian ini menguji integrasi, bukan teori baru

Setiap satu dari enam subtugas di Bagian 16.3 dan 16.4 adalah penerapan langsung dan bernama dari sesuatu yang sudah diajarkan, didemonstrasikan bekerja, dan benar-benar diuji oleh Bab 14 atau Bab 15. Kelima fungsi Tugas 1 adalah persis lima fungsi yang ditulis dan diverifikasi Bab 14 — Anda diminta mereproduksi desain yang sama dan sudah terbukti itu sendiri, bukan menciptakan yang baru. Tugas 2 adalah contoh kerja yang sama persis yang diselesaikan Bab 15 dua kali (sekali dengan hash table buatan sendiri, sekali dengan std::unordered_map) — Anda diminta menyelesaikannya untuk ketiga kalinya, sendiri, lalu menjelaskan solusi Anda secara tertulis.

16.2 Cara Menggunakan Bab Latihan Ini

Apa bab ini — dan apa yang bukan

Ini adalah bab praktik, bukan bab materi baru — yang terakhir dalam mata kuliah ini. Ujian Akhir, seperti asesmen DS sesungguhnya yang menjadi modelnya, adalah ujian coding INDIVIDUAL, open, take-home — Anda bekerja sendiri, persis seperti Kuis 1 (Bab 4) dan UTS (Bab 8); pengumpulan tim hingga tiga mahasiswa hanyalah pengecualian pada Kuis 2 (Bab 12), bukan aturan umum mata kuliah ini. Bagian 16.3 dan 16.4 di bawah menjabarkan kedua tugas ujian ini, masing-masing disertai callout panduan (THINK, TRAP, INSIGHT), bukan solusi lengkap yang sudah dikerjakan. Seperti setiap bab latihan sebelumnya, bab ini TIDAK menyertakan subfolder code/ — program yang Anda tulis untuk Ujian Akhir, dan laporan yang Anda tulis tentangnya, adalah pekerjaan individual dan mandiri Anda sendiri.

Sebelum Anda mulai

Baca ulang heap.h milik Bab 14 dan demo itinerary milik Bab 15 (baik `demo_itinerary_handrolled.cpp` maupun `demo_itinerary_unordered_map.cpp`) sebelum membuka file kosong. Setiap fungsi yang perlu Anda tulis di bawah adalah kerabat dekat dari sesuatu di salah satu dari dua file itu — tugas Anda adalah mengenali kerabat yang mana, bukan menurunkan algoritma dari prinsip dasar.

16.3 Tugas 1 — Lengkapi Skeleton MaxHeap (50 poin, 5 × 10)

Anda diberikan skeleton di bawah — heap.h milik Bab 14 sendiri, dengan operasi dasarnya (insert, deleteRoot, getMax, heapifyArray, heapSort, printHeap, clear, dan heapifyUp/heapifyDown privat) sudah lengkap, persis seperti yang didemonstrasikan bekerja oleh Bab 14. Lima fungsi anggota dibiarkan sebagai stub TODO. Lengkapi masing-masing sehingga kelas hasilnya lulus pemeriksaan urutan-heap dan kebenaran sejenis yang dijalankan harness otomatis Bab 14 sendiri.

Listing 16.1 — Skeleton yang diberikan (heap.h)

Semua yang ditandai DIBERIKAN di bawah sudah lengkap dan tidak berubah dari Bab 14 — tidak perlu dimodifikasi. Semua yang berada di bawah spanduk UJIAN AKHIR yang ditandai // TODO -- tulis kode Anda di sini. adalah bagian yang harus Anda tulis; SIGNATURE fungsi itu sendiri (nama, tipe parameter, tipe kembalian) tidak boleh berubah, karena file Anda akan dikompilasi dan diuji terhadap persis kelima signature ini. Listing dipecah menjadi empat blok di bawah semata-mata untuk tata letak halaman — ini tetap satu file heap.h yang menyatu.

```
// heap.h -- SKELETON YANG DIBERIKAN untuk Ujian Akhir, Tugas 1.
// Ini adalah kelas MaxHeap milik Bab 14. Setiap fungsi yang sudah pernah
// Anda lihat bekerja (insert, deleteRoot, getMax, heapifyArray, heapSort,
// printHeap, clear, plus heapifyUp/heapifyDown privat) SUDAH DIBERIKAN,
// tidak berubah. Tugas Anda adalah menulis badan lima fungsi di bawah
// yang ditandai TODO -- tidak ada bagian lain file ini yang perlu diubah.
#ifndef HEAP_H
#define HEAP_H

#include <iostream>
#include <stdexcept>
#include <vector>

class MaxHeap {
public:
    MaxHeap() = default;

    // --- DIBERIKAN: insert -- tambah + heapifyUp ("percolate up") ------
    void insert(int key) {
        heap.push_back(key);
        size_t index = heap.size() - 1;
        heapifyUp(index);
    }

    // --- DIBERIKAN: deleteRoot -- hapus max, cetak-lalu-kembali jika kosong
    void deleteRoot() {
        if (heap.empty()) {
            std::cout << "Heap is empty" << std::endl;
            return;
        }
        heap[0] = heap.back();
        heap.pop_back();
        if (!heap.empty()) heapifyDown(0);
    }
```

```
    // --- DIBERIKAN: getMax -- lihat root, O(1) ---------------------------
    int getMax() const {
        if (heap.empty()) {
            std::cout << "Heap is empty" << std::endl;
            return -1;
        }
        return heap[0];
    }

    // --- DIBERIKAN: heapifyArray -- bangun O(n) bottom-up, GANTI heap ----
    void heapifyArray(const std::vector<int> &arr) {
        heap = arr;
        for (int i = static_cast<int>(heap.size()) / 2 - 1; i >= 0; --i) {
            heapifyDown(static_cast<size_t>(i));
        }
    }

    // --- DIBERIKAN: heapSort -- ambil max berulang kali, O(n log n) ------
    std::vector<int> heapSort() {
        std::vector<int> sorted;
        while (!heap.empty()) {
            sorted.push_back(getMax());
            deleteRoot();
        }
        return sorted;
    }

    void printHeap() const {
        for (int v : heap) std::cout << v << " ";
        std::cout << std::endl;
    }

    // --- DIBERIKAN: clear -- kosongkan heap sepenuhnya --------------------
    void clear() { heap.clear(); }
```

Lima fungsi yang harus Anda tulis

Semua yang mulai dari sini hingga spanduk UJIAN AKHIR yang sesuai di bawah adalah yang Anda lengkapi — kelas dasar di atas sudah memberi Anda setiap helper yang dibutuhkan kelima fungsi ini (heapifyUp, heapifyDown, dan vektor heap privat).

```
// ============================== UJIAN AKHIR ==============================

    // 1. ExtractMax: hapus DAN kembalikan elemen maksimum. [10 poin]
    int extractMax() {
        // TODO -- tulis kode Anda di sini.
    }

    // 2. IncreaseKey: naikkan key pada `index` menjadi `newValue`, lalu
    //    pulihkan properti urutan heap. [10 poin]
    void increaseKey(int index, int newValue) {
        // TODO -- tulis kode Anda di sini.
    }

    // 3. GetMin: cari elemen minimum TANPA menghapusnya. [10 poin]
    int getMin() {
        // TODO -- tulis kode Anda di sini.
    }

    // 4. BuildMaxHeap: bangun Max-Heap dari array tak terurut. [10 poin]
    void buildMaxHeap(const std::vector<int> &arr) {
        // TODO -- tulis kode Anda di sini.
    }

    // 5. IsMaxHeap: periksa apakah array sembarang sudah memenuhi properti
    //    Max-Heap, tanpa menyentuh atau membangunnya ulang. [10 poin]
    bool isMaxHeap(const std::vector<int> &arr) {
        // TODO -- tulis kode Anda di sini.
    }

// ============================== UJIAN AKHIR ==============================
```

```
private:
    std::vector<int> heap;

    void heapifyDown(size_t index) { /* DIBERIKAN -- tak berubah dari Bab 14
*/ }
    void heapifyUp(size_t index)   { /* DIBERIKAN -- tak berubah dari Bab 14
*/ }
};

#endif // HEAP_H
```

Asal Setiap Subtugas Ujian Akhir

Tugas 1 memberi Anda skeleton MaxHeap Bab 14 dengan lima fungsi dihapus; Tugas 2 memakai ulang contoh kerja itinerary Bab 15 -- setiap baris di bawah dapat ditelusuri kembali ke bab sebelumnya yang spesifik.

#	Subtugas Ujian Akhir	Berasal dari	Poin
1	extractMax() -- hapus dan kembalikan nilai maksimum	Bab 14 -- bentuk penghapusan deleteRoot() yang sama persis, ditingkatkan agar throw saat kosong, bukan cetak-lalu-kembali	10
2	increaseKey(index, newValue)	Bab 14 -- heapifyUp(): naikkan sebuah key, lalu pulihkan urutan heap ke atas mulai dari indeks itu	10
3	getMin() -- cari nilai minimum tanpa menghapusnya	Bab 14 -- pembuktian Bagian 14.6 bahwa nilai minimum max-heap selalu berada di daun (leaf)	10
4	buildMaxHeap(arr) -- bangun dari array tak terurut	Bab 14 -- pembangunan bottom-up O(n) Bagian 14.4, bentuknya identik dengan heapifyArray()	10
5	isMaxHeap(arr) -- periksa validitas pada array sembarang	Bab 14 -- properti urutan heap, diperiksa tanpa menyentuh atau membangun ulang apa pun	10
6	Hashing: rekonstruksi itinerary + laporan tertulis	Bab 15 -- HashMap<from,to> + reverse map untuk menemukan titik awal, lalu rangkai from->to; unordered_map, bukan std::map	50

Total: 100 poin

Gambar 16.2 — Setiap subtugas dalam ujian ini dapat ditelusuri kembali ke ide spesifik Bab 14 atau Bab 15; tidak satu pun merupakan materi baru.

16.3.1 Subtugas 1 — extractMax() (10 poin)

Hapus DAN kembalikan elemen maksimum (root) dari heap, memulihkan properti urutan heap sesudahnya. Berbeda dari deleteRoot() (yang sudah diberikan skeleton, mencetak pesan dan kembali pada heap kosong), extractMax() harus mengomunikasikan kondisi heap-kosong melalui exception C++ yang tepat, bukan pesan cetak dan pengembalian diam-diam.

Petunjuk

BENTUK penghapusannya identik dengan deleteRoot(): pindahkan elemen terakhir ke slot root, kecilkan vektor satu, lalu heapifyDown(0) untuk memulihkan urutan — hanya dua perbedaannya adalah (a) Anda harus menangkap dan mengembalikan nilai root ASLI sebelum menimpanya, dan (b) heap kosong harus melempar exception, bukan cetak-lalu-kembali.

Simpan nilainya sebelum Anda menimpa indeks 0

heap[0] = heap.back() menimpa persis nilai yang perlu Anda kembalikan — baca dan simpan heap[0] ke variabel lokal TERLEBIH DAHULU, baru lakukan penghapusan, baru kembalikan nilai yang disimpan. Ingat juga untuk hanya memanggil heapifyDown ketika heap tidak kosong SESUDAH pop_back(): heap yang tadinya memiliki tepat satu elemen sebelum ekstraksi menjadi kosong sesudahnya, dan heapifyDown(0) pada vektor kosong membaca di luar batas.

Mengapa exception dilempar di sini, bukan pesan cetak

Perilaku cetak-lalu-kembali milik deleteRoot() adalah pilihan desain skeleton ASLI, mendahului ujian ini. extractMax() adalah peningkatan yang disengaja dari ujian ini: pemanggil yang menulis `int m = h.extractMax();` pada heap kosong dan menerima kembali pesan cetak plus int yang tidak jelas tidak punya cara andal untuk mendeteksi kegagalan itu dalam kode — pemanggil yang membungkus panggilan yang sama dalam try/catch untuk std::runtime_error punya cara itu. Ini adalah perbaikan desain API yang sesungguhnya dan terungkap, bukan pekerjaan sia-sia.

16.3.2 Subtugas 2 — increaseKey(int index, int newValue) (10 poin)

Naikkan nilai yang tersimpan pada indeks tertentu menjadi newValue, lalu pulihkan properti urutan heap dengan memindahkan nilai (yang kini lebih besar) ke atas sejauh yang diperlukan.

Petunjuk

Ini adalah heapifyUp(), diterapkan mulai dari indeks yang Anda terima, bukan dari indeks terakhir yang baru disisipkan — fungsi yang Anda tulis adalah pembungkus tipis: validasi, tetapkan, lalu panggil heapifyUp(index) privat yang SAMA yang sudah dipanggil insert() yang DIBERIKAN.

Dua mode kegagalan BERBEDA butuh dua exception BERBEDA

Indeks yang negatif atau berada di/melewati heap.size() adalah akses di luar jangkauan — jenis kesalahan pemanggil yang berbeda dari newValue yang justru LEBIH KECIL dari nilai saat ini pada indeks itu, yang dijanjikan nama fungsinya sendiri ("increase") tidak pernah terjadi. Bedakan keduanya dengan dua tipe exception yang berbeda dan bernama jelas, bukan satu error generik, dan periksa batas indeks SEBELUM Anda mengindeks ke vektor dengannya — membaca heap[index] untuk dibandingkan dengan newValue ketika index sudah di luar jangkauan itu sendiri adalah undefined behavior.

Hanya perlu bergerak ke ATAS, tidak pernah ke bawah

Karena nilai baru hanya bisa LEBIH BESAR dari yang sebelumnya ada, ia hanya bisa melanggar properti heap dengan PARENT-nya (parent terlalu kecil), tidak pernah dengan anak-anaknya (anak-anak sudah tidak lebih besar dari nilai lama yang lebih kecil, sehingga tetap tidak lebih besar dari nilai baru yang lebih besar). Inilah persis mengapa heapifyUp — bukan heapifyDown — adalah langkah pemulihan yang benar di sini.

16.3.3 Subtugas 3 — getMin() (10 poin)

Ambil elemen minimum yang sedang tersimpan dalam max-heap, TANPA menghapusnya dan tanpa mengganggu struktur heap dengan cara apa pun.

Petunjuk — di mana minimum harus berada?

Max-heap hanya menjamin bahwa parent tidak lebih kecil dari anak-anaknya — ia tidak mengatakan apa pun tentang urutan ANTAR saudara atau lintas subtree berbeda, sehingga minimum secara prinsip bisa berada di mana saja. Tapi ia tidak pernah bisa menjadi node INTERNAL (node mana pun dengan setidaknya satu anak harus >= anak itu, sehingga tidak bisa menjadi nilai terkecil yang ada) — minimum karenanya harus berada di sebuah DAUN (leaf). Dalam representasi array ini, setiap indeks dari heap.size()/2 hingga heap.size()-1 adalah daun (tidak ada anak untuk indeks-indeks itu) — memindai hanya rentang itu, kira-kira separuh array, sudah cukup.

Ini memindai DAUN, bukan seluruh array — dan tetap O(n), bukan O(log n)

Jangan keliru menganggap ini operasi O(log n) seperti getMax(): meskipun Anda hanya memindai kira-kira separuh array, itu tetap linear terhadap ukuran heap, karena kira-kira separuh dari array mana pun tetaplah Theta(n). Kesalahan umum adalah memindai SEMUA n elemen alih-alih hanya rentang daun — ini tetap menghasilkan jawaban benar, hanya melakukan persis dua kali kerja yang diperlukan; kesalahan yang lebih halus adalah salah satu (off-by-one) pada indeks awal rentang daun (yaitu heap.size()/2, bukan heap.size()/2 - 1 atau heap.size()/2 + 1) sehingga diam-diam mengecualikan atau menyertakan satu node internal ekstra.

Heap kosong tidak punya minimum untuk dilaporkan
Sama seperti extractMax(), getMin() sebaiknya melempar exception, bukan mengembalikan nilai sentinel buatan, pada heap kosong — tipe kembalian int tidak punya cara merepresentasikan "tidak ada nilai seperti itu" selain nilai yang, membingungkan, juga bisa menjadi entri heap yang sah.

16.3.4 Subtugas 4 — buildMaxHeap(const std::vector<int>& arr) (10 poin)

Bangun max-heap dari array masukan sembarang yang tak terurut, menggantikan apa pun yang sedang dipegang objek heap ini.

Petunjuk — ini adalah heapifyArray(), dengan nama lain
Bandingkan perilaku yang dibutuhkan fungsi ini dengan heapifyArray() yang DIBERIKAN pada skeleton di atas: keduanya menerima array sembarang, keduanya menggantikan heap internal objek dengannya, dan keduanya memulihkan properti heap melalui pembangunan bottom-up O(n) standar — mulai dari indeks non-daun terakhir (heap.size()/2 - 1) dan memanggil heapifyDown di setiap indeks turun ke 0. Jika buildMaxHeap Anda dan heapifyArray yang diberikan berakhir terlihat hampir identik, itu memang diharapkan, bukan tanda Anda salah membaca tugas.

Pengurangan signed/unsigned pada array kosong
heap.size() adalah size_t yang TIDAK BERTANDA; menghitung heap.size()/2 - 1 ketika heap.size() adalah 0 atau 1 bisa menghasilkan 0 (aman, loop tidak jalan sama sekali jika Anda memakai penghitung loop bertanda dimulai dari n/2-1 yang di-cast dengan tepat) atau, jika dilakukan sembarangan dengan variabel loop tak bertanda, underflow menjadi bilangan sangat besar dan loop berjalan jauh melewati batas array. heap.h milik Bab 14 sendiri menangani ini dengan menghitung batas loop sebagai int bertanda dan hanya meng-cast ke size_t di dalam badan loop, setelah memastikan indeksnya >= 0 — ikuti pola yang sama.

Mengapa ujian meminta ini PADAHAL heapifyArray sudah ada
buildMaxHeap dan heapifyArray secara fungsional adalah algoritma yang sama, secara sengaja — Bab 14 sudah menunjukkan ini secara langsung (catatan berdampingan Bagian 14.4) sehingga subtugas ini seharusnya terasa seperti mengenali, bukan menciptakan. Ujian menamainya buildMaxHeap secara khusus karena itulah kosakata yang paling sering dipakai literatur heap dan mata kuliah lain untuk persis langkah konstruksi O(n) ini, berbeda dari memanggil insert() sebanyak n kali (yang akan berbiaya O(n log n)).

16.3.5 Subtugas 5 — isMaxHeap(const std::vector<int>& arr) (10 poin)

Periksa apakah array SEMBARANG (bukan heap internal milik objek ini) sudah memenuhi properti max-heap — setiap parent pada indeks i harus >= kedua anaknya pada 2i+1 dan 2i+2 — tanpa memodifikasi atau membangun ulang apa pun.

Petunjuk — periksa setiap node INTERNAL, dan hanya node internal

Daun tidak punya anak untuk dilanggar propertinya, sehingga hanya indeks dengan setidaknya satu anak yang perlu diperiksa: i berkisar dari 0 hingga (dan termasuk) (arr.size()-2)/2. Untuk setiap i seperti itu, verifikasi arr[i] >= arr[2*i+1], dan, jika anak kanan ada (2*i+2 < arr.size()), juga arr[i] >= arr[2*i+2]. Begitu satu saja pemeriksaan gagal, array itu bukan max-heap — kembalikan false segera, jangan lanjutkan memindai.

arr.size() - 2 pada array 0 atau 1 elemen mengalami underflow

arr.size() tidak bertanda; untuk array dengan kurang dari 2 elemen, arr.size() - 2 melingkar menjadi nilai sangat besar SEBELUM batas loop bahkan dipakai, dan akses array berikutnya membaca jauh di luar batas — ini adalah bug sesungguhnya yang ditemukan proses QA Bab 14 sendiri pada jawaban model ujian yang dipulihkan, diungkapkan secara jujur alih-alih diam-diam direproduksi (Lampiran 14.A). Jaga secara eksplisit: array dengan 0 atau 1 elemen secara sepele memenuhi properti max-heap (tidak ada pasangan parent-anak untuk dilanggar), jadi kembalikan true segera sebelum mengevaluasi aritmetika indeks bergantung-ukuran apa pun.

Ini memeriksa array yang BISA dihasilkan buildMaxHeap milik Bab 14, tapi tidak harus

isMaxHeap menerima vector<int> sembarang sebagai argumennya — ia tidak perlu, dan umumnya bukan, heap milik objek ini SENDIRI. Inilah persis mengapa ia ditulis sebagai pemeriksaan pada parameter arr yang diberikan sepanjang fungsi, tidak pernah merujuk ke anggota heap privat kelas sama sekali; implementasi yang benar bisa ditulis sebagai fungsi bebas, atau (seperti pilihan Bab 14, diungkapkan sebagai keputusan desain mata kuliah) fungsi anggota static, tanpa mengubah perilakunya.

16.4 Tugas 2 — Hashing: Rekonstruksi Itinerary + Laporan Tertulis (50 poin)

Anda diberikan daftar leg penerbangan/perjalanan yang tidak berurutan, masing-masing dicatat sebagai pasangan (from, to). Menggunakan pendekatan berbasis hash (hash table sesungguhnya — std::unordered_map, atau hash table buatan sendiri seperti milik Bab 15 — bukan std::map), rekonstruksi dan cetak seluruh itinerary dalam urutan perjalanan yang benar, dimulai dari titik awal perjalanan yang unik.

Algoritma persisnya, dalam tiga langkah

[1] Bangun HashMap bernama dataset, di mana setiap entri berbentuk from -> to (satu entri per leg). [2A] Bangun HashMap KEDUA, reverseMap, di mana setiap entri berbentuk to -> from — kebalikan dari setiap entri di dataset. [2B] Temukan titik awal itinerary: telusuri key-key dataset, dan satu key yang TIDAK muncul sebagai key di reverseMap adalah titik awal perjalanan (ia tidak pernah menjadi tujuan siapa pun, sehingga tidak pernah muncul sebagai "to"). [3] Mulai dari titik itu, cari berulang kali dataset[current] untuk mendapatkan kota berikutnya, mencetak setiap hop from -> to, hingga tidak ada entri lagi untuk diikuti.

Contoh kerja — dataset persis yang dipakai ujian ini (juga contoh kerja milik Bab 15 sendiri, demi kesinambungan), diberikan dalam urutan penyisipan teracak (shuffled) berikut:

```
Cianjur  -> Bandung
Badung   -> Denpasar
Gianyar  -> Cianjur
Denpasar -> Gianyar
```

reverseMap yang dibangun dari dataset di atas:

```
Bandung  -> Cianjur
Denpasar -> Badung
Cianjur  -> Gianyar
Gianyar  -> Denpasar
```

Badung tidak pernah muncul sebagai KEY reverseMap (tidak ada rute yang pernah tiba DI Badung) — Badung adalah titik awal. Merangkai dari sana melalui dataset menghasilkan keluaran yang diharapkan:

```
Badung -> Denpasar, Denpasar -> Gianyar, Gianyar -> Cianjur, Cianjur -> Bandung
```

Petunjuk — pakai ulang bentuk persis Bab 15, secara mandiri

Ini persis algoritma yang dijabarkan Bagian 15.6 milik Bab 15 dua kali — sekali dengan hash table chaining buatan sendiri berkunci hashString(), sekali dengan std::unordered_map<std::string,std::string>. Anda tidak diminta menciptakan algoritma baru; Anda diminta mengimplementasikan bentuk tiga-langkah yang sama ini sendiri, dengan benar, dan menjelaskannya dengan kata-kata Anda sendiri dalam laporan Anda.

Gunakan hash table SESUNGGUHNYA — ini persis kesalahan penamaan yang dikoreksi Bab 15

Jawaban model historis ujian yang asli untuk tugas ini menggunakan std::map<std::string,std::string> — balanced binary search tree berkompleksitas O(log n), bukan hash table, meskipun tugas ini bernama "Hashing" dan secara eksplisit mensyaratkan "metode hashing". Pengumpulan yang memakai std::map di sini mengulangi kesalahan penamaan yang sama yang sudah diungkapkan itu; gunakan std::unordered_map (pencarian rata-rata O(1) lewat fungsi hash sesungguhnya) atau hash table buatan sendiri seperti milik Bab 15, dan siaplah menjelaskan dalam laporan Anda MENGAPA pilihan container Anda benar-benar hashing dan bukan sekadar pengganti bernama sama.

Inisialisasi setiap variabel loop/flag yang Anda pakai untuk memisahkan hop dengan koma

Versi historis dari jawaban model program persis ini mendeklarasikan `int a;` polos dan memakainya, tanpa inisialisasi, sebagai flag penentu apakah harus mencetak koma di depan setiap hop — membaca nilai variabel lokal yang tidak diinisialisasi adalah undefined behavior dalam C++, dan keluaran yang dihasilkan tidak dapat diandalkan. Deklarasikan dan INISIALISASI flag semacam itu secara eksplisit (misalnya `bool firstHop = true;`) sebelum loop traversal Anda dimulai, persis seperti yang dilakukan demo terkoreksi milik Bab 15 sendiri.

Mengapa reverse map adalah bagian yang cerdas, bukan traversalnya

Begitu titik awal diketahui, mencetak rute adalah pencarian hash-map berulang yang sederhana — langkah yang sesungguhnya cerdas adalah MENEMUKAN titik awal itu sejak awal tanpa memindai seluruh dataset secara brute force untuk "kota-from mana yang tidak pernah menjadi kota-to": membangun reverseMap dan memeriksa keanggotaan mengubah pencarian yang jika tidak begitu akan berbiaya O(n^2) (bandingkan setiap from terhadap setiap to) menjadi dua pass ber-O(n) atas data, masing-masing pencarian berjalan dalam waktu rata-rata O(1) berkat hashing sesungguhnya.

16.4.1 Laporan Tertulis (wajib, bagian dari 50 poin ini)

Isi laporan — wajib, dan menonjol dalam instruksi ujian ini sendiri

Laporan Anda harus mencakup: (1) METODE yang Anda gunakan untuk menyelesaikan masalah — pendekatan hashing, trik reverse-map, dan mengapa keduanya benar; (2) KODE SUMBER Anda, disertakan atau dirujuk dengan jelas; (3) HASIL EKSEKUSI Anda dan ANALISIS atasnya (apakah keluarannya cocok dengan itinerary yang diharapkan? apa yang terjadi pada masukan cacat tanpa titik awal unik, atau dengan siklus?); dan (4) KESIMPULAN yang merangkum apa yang didemonstrasikan latihan ini. Ini adalah bentuk empat-bagian yang sama — metode, kode, hasil-dan-analisis, kesimpulan — yang diharapkan dimiliki setiap laporan teknis dalam lingkungan profesional atau riset.

Catatan: berbeda dari komponen laporan asesmen-asesmen sebelumnya, laporan Ujian Akhir memiliki bobot proporsional yang lebih besar

Kuis 1, UTS, dan Kuis 2 meminta kode yang bekerja dan benar sebagai satu-satunya deliverable, tanpa komponen laporan tertulis yang dinilai terpisah. Ujian Akhir berbeda secara sengaja by design: ini adalah dokumentasi CAPSTONE Anda atas seluruh perangkat struktur data mata kuliah ini, dan laporan tertulisnya dinilai sebagai bagian substansial yang berbobot terpisah dari 50 poin Tugas 2 (rubrik Bagian 16.5 memberikan pembagian persisnya) — bukan sekadar formalitas. Perlakukan bagian analisis dan kesimpulan sama seriusnya dengan kode itu sendiri: program yang benar dengan laporan tipis satu paragraf tidak akan mendapat nilai penuh pada tugas ini, sebagaimana program Kuis 1 atau UTS yang bekerja saja dulu bisa.

Apa yang sesungguhnya terkandung dalam bagian analisis yang kuat

Selain mengonfirmasi keluaran cocok dengan contoh kerja Bagian 16.4, analisis yang kuat membahas: mengapa std::unordered_map (atau tabel buatan sendiri Anda) adalah container yang TEPAT di sini dan apa yang akan salah (atau sekadar menjadi lebih lambat) jika memakai std::map; apa yang dilakukan program Anda pada masukan TANPA titik awal unik (setiap kota adalah tujuan seseorang — siklus tanpa awal) atau pada masukan dengan lebih dari satu awal yang masuk akal; dan kompleksitas waktu solusi Anda sebagai fungsi jumlah leg, dalam notasi Big-O, menghubungkan kembali ke pengenalan formal notasi itu di Bab 2.

16.5 Format dan Penilaian Ujian Akhir

Ujian Akhir adalah ujian coding INDIVIDUAL, open, take-home — bukan tugas tim, persis seperti Kuis 1 (Bab 4) dan UTS (Bab 8). "Open" berarti buku teks, catatan Anda sendiri, dan materi referensi C++ standar (seperti cppreference.com) semuanya boleh digunakan selama Anda mengerjakan — ini TIDAK berarti Anda boleh menyalin kode mahasiswa lain, laporan mahasiswa lain, atau mengumpulkan pekerjaan yang bukan sesungguhnya milik Anda sendiri — ujian open tetap menguji apakah ANDA bisa menerapkan apa yang sudah diajarkan mata kuliah ini, bekerja secara mandiri, dan sebagian besar versi nyata ujian ini mensyaratkan janji integritas akademik individual yang ditandatangani dan dikumpulkan bersama pekerjaan Anda.

Rubrik poin-per-subtugas khas DS, sekali lagi

Seperti setiap asesmen sebelumnya dalam mata kuliah ini, Ujian Akhir dinilai tugas demi tugas, dalam poin utuh, bukan dengan template Design/Implementation/Analysis/Conclusion berbobot persentase. Kelima subtugas Tugas 1 masing-masing bernilai 10 poin (total 50); 50 poin Tugas 2 terbagi menjadi 35 untuk implementasi yang benar dan benar-benar hashing, serta 15 untuk laporan tertulis — bobot laporan yang sengaja lebih berat dibanding asesmen mana pun sebelumnya dalam mata kuliah ini, mencerminkan catatan Bagian 16.4.1 di atas.

Subtugas	Apa yang diuji	Poin
1a. extractMax()	Hapus-dan-kembalikan max dengan benar, dengan exception dilempar saat kosong (bukan pesan cetak)	10
1b. increaseKey(index, newValue)	Validasi batas/arah yang benar, lalu heapifyUp memulihkan urutan	10
1c. getMin()	Pemindaian rentang-daun yang benar (bukan pemindaian O(n) penuh yang salah jenis, dan bukan node internal), melempar exception saat kosong	10
1d. buildMaxHeap(arr)	Pembangunan bottom-up O(n) yang benar dari array tak terurut sembarang	10
1e. isMaxHeap(arr)	Pemeriksaan validitas yang benar pada array sembarang, dengan underflow 0/1-elemen dijaga	10
2a. Implementasi Hashing	Algoritma HashMap + reverseMap yang benar memakai hashing SESUNGGUHNYA (unordered_map atau hash table buatan sendiri, bukan std::map); menemukan titik awal unik dengan benar dan mencetak seluruh rute terurut	35
2b. Laporan tertulis	Metode, kode sumber, hasil eksekusi & analisis, dan kesimpulan — keempat bagian hadir dan substantif	15
Total		100

Kumpulkan Tugas 1 sebagai satu heap.h/.cpp yang dapat dikompilasi sendiri (atau satu .cpp dengan kelas inline) yang mendemonstrasikan kelima fungsi baru bekerja dengan benar terhadap contoh kerja di Bagian 16.3, dan Tugas 2 sebagai satu file .cpp ditambah laporan tertulis Anda — program yang tidak dapat dikompilasi tidak mendapat poin untuk subtugas itu, tidak peduli seberapa dekat logikanya

dengan benar, jadi sisakan waktu untuk benar-benar membangun dan menjalankan kedua pengumpulan sebelum menyelesaikan laporan Anda.

16.6 Ringkasan Bab — dan Akhir Mata Kuliah Ini

- Ujian Akhir tidak memperkenalkan materi baru — ini adalah checkpoint kumulatif atas Bab 14 (Heap & Priority Queue) dan Bab 15 (Hashing & Hash Table), dua bab tepat sebelumnya.
- Ini adalah ujian coding INDIVIDUAL, open, take-home — bukan tugas tim; pengumpulan tim hingga 3 mahasiswa hanyalah pengecualian Kuis 2.
- Tugas 1 (50 poin, 5 x 10): lengkapi skeleton MaxHeap milik Bab 14 dengan menulis extractMax, increaseKey, getMin, buildMaxHeap, dan isMaxHeap — persis lima fungsi yang sudah diimplementasikan dan diverifikasi Bab 14.
- Tugas 2 (50 poin: 35 implementasi + 15 laporan): rekonstruksi contoh kerja itinerary Bab 15 memakai hash table SESUNGGUHNYA, ditambah laporan tertulis yang mencakup metode, kode, hasil-dan-analisis, dan kesimpulan — laporan ini berbobot lebih berat dibanding komponen laporan asesmen mana pun sebelumnya dalam mata kuliah ini.
- Penilaian mengikuti rubrik poin-per-subtugas khas DS sendiri — bentuk yang sama dipakai di Bab 4, 8, dan 12, diterapkan untuk yang terakhir kalinya.
- Bab ini tidak menyertakan subfolder code/: kedua pengumpulan, dan laporannya, adalah pekerjaan individual dan mandiri masing-masing mahasiswa.

Ini adalah bab keenam belas dan terakhir dari Struktur Data (EF234201). Dari array dan pencarian di Bab 1 hingga heap dan hashing di Bab 14-15, setiap struktur data dan algoritma yang dibahas buku ini telah membawa Anda ke sini — ujian ini meminta Anda menunjukkan, sekali lagi dan sepenuhnya sendiri, bahwa perangkat itu sesungguhnya milik Anda. Selamat telah mencapai akhir mata kuliah ini.

Lampiran 16.A — Daftar Periksa Mandiri (Tidak Dinilai)

Sebelum mengumpulkan Ujian Akhir Anda, jalankan melalui daftar periksa ini. Ini memakan waktu beberapa menit dan menangkap sebagian besar kesalahan yang paling sering dilihat penilai pada pengumpulan pertama ujian ini.

- Apakah kedua program dikompilasi dengan bersih menggunakan `g++ -Wall -Wextra -std=c++17`, tanpa peringatan?
- Tugas 1: apakah extractMax() menyimpan heap[0] ke variabel lokal SEBELUM menimpanya, dan melempar exception (bukan mencetak) pada heap kosong?
- Tugas 1: apakah increaseKey() memeriksa batas indeks SEBELUM mengindeks ke vektor dengannya, dan melempar dua tipe exception BERBEDA untuk indeks di luar jangkauan versus nilai yang menurun?
- Tugas 1: apakah getMin() memindai hanya rentang daun (heap.size()/2 hingga heap.size()-1), bukan seluruh array dan bukan hanya node internal?

- Tugas 1: apakah isMaxHeap() mengembalikan true segera untuk array 0 atau 1 elemen, SEBELUM mengevaluasi arr.size()-2 di mana pun?
- Tugas 2: sudahkah Anda memakai std::unordered_map (atau hash table buatan sendiri) alih-alih std::map untuk dataset maupun reverseMap?
- Tugas 2: apakah setiap variabel flag yang dipakai dalam loop traversal Anda diinisialisasi secara eksplisit sebelum loop dimulai — tidak ada int tak terinisialisasi dipakai sebagai flag pemisah koma?
- Tugas 2: apakah program Anda dengan benar mengidentifikasi Badung sebagai titik awal pada contoh kerja, dan mencetak rute persis yang diharapkan?
- Laporan: apakah berisi keempat bagian wajib — metode, kode sumber, hasil eksekusi & analisis, dan kesimpulan — masing-masing substantif, bukan satu baris saja?
- Apakah setiap file yang Anda kumpulkan, dan setiap kata laporan Anda, sungguh-sungguh pekerjaan individual Anda sendiri, ditulis untuk Ujian Akhir ini — bukan disalin dari mahasiswa lain, solusi tahun sebelumnya, atau alat AI yang tidak diizinkan kebijakan integritas akademik mata kuliah Anda sendiri?

Referensi

[1] Format asesmen bab latihan ini dimodelkan langsung dari Ujian Akhir sesungguhnya mata kuliah ini (EF234201 Struktur Data): ujian coding individual, open, take-home dengan dua tugas — Heap (50 poin, 5 subtugas x 10) dan Hashing (50 poin, termasuk laporan tertulis wajib) — dinilai tugas demi tugas, bukan dengan rubrik berbobot persentase. Format ini dipakai ulang di sini apa adanya; kerangka tugas di atas (yang mengulas kembali konten sesungguhnya Bab 14-15 dan mengganti masalah kualitas-kode yang diketahui pada jawaban model ujian asli — kesalahan penamaan std::map-sebagai-"Hashing" dan bug variabel tak terinisialisasi — dengan panduan generik yang terkoreksi alih-alih diam-diam mereproduksi keduanya) telah diadaptasi untuk buku teks ini.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” Edisi ke-3, MIT Press, 2009. (Heap dan heapsort, Bab 6; hash table, Bab 11.)

[3] cppreference.com. “std::unordered_map,” “std::vector,” “Exceptions.” https://en.cppreference.com/w/cpp (diakses Agustus 2026).