Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

TABLE OF CONTENTS

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 1

C/C++ Refresher & Memory Model → Arrays Foundations

The first chapter of the course, and the foundation every later chapter builds on. Every line of code in this chapter was actually compiled with g++ -Wall -Wextra and run — the exact terminal transcript is reproduced in Appendix 1.A.

Learning Objectives — after this chapter you will be able to:
(1) Explain, in one paragraph, what "Data Structure" studies as a discipline and why it matters as programs and their data grow. (2) Distinguish a variable's value from its address, and read basic `&` / pointer-preview syntax in C/C++. (3) Explain how an array of a struct is laid out contiguously in memory, and derive the address of any element from the base address, the index, and `sizeof`. (4) Declare, populate, and traverse a 1D array of a struct type. (5) Declare a 2D array and derive the row-major address-arithmetic formula for any `arr[row][col]`.

One Toolchain, Three Operating Systems — Windows 11, macOS, Ubuntu

OS	One-time install	Compile & run (identical everywhere)
Windows 11	MinGW-w64: winget install BrechtSanders.WinLibs (or WSL2: wsl --install)	g++ -Wall -Wextra -std=c++17 -O2 -o prog prog.cpp then prog.exe / ./prog
macOS	xcode-select --install (Apple clang answers as g++)	g++ -Wall -Wextra -std=c++17 -O2 -o prog prog.cpp then ./prog
Ubuntu/Linux	sudo apt update && sudo apt install build-essential	g++ -Wall -Wextra -std=c++17 -O2 -o prog prog.cpp then ./prog

Every example in this book runs identically on Windows 11, macOS, and Ubuntu/Linux; commands differ only at install time. Pick your row once and follow along on your own laptop.

1.1 Welcome to Data Structure

"Data Structure" is the study of how to organize data in memory so that the operations a program actually needs — inserting, deleting, searching, sorting, and traversing — are as fast and as simple to reason about as possible. The same collection of data can be stored a dozen different ways, and the way you choose has real, measurable consequences: a search that takes microseconds in one layout can take seconds in another, once the collection is large enough. This course is not about learning ten unrelated tricks; it is about learning a small number of layout ideas — contiguous arrays, linked chains of nodes, hierarchical trees, networked graphs, and hashed buckets — and, for each one, understanding exactly what it makes fast, what it makes slow, and why.

Why does this matter in practice? Every non-trivial program keeps data around: a phonebook app keeps contacts, a game keeps players and their inventories, a compiler keeps the symbols it has seen so far. As the number of items grows from ten to ten thousand to ten million, an operation that scans every item one by one — fine at ten items — becomes the reason a real application feels slow. A big part of a working programmer's job is recognizing which structure the current problem calls for, and this course is built to give you that judgment, not just the syntax.

The term's roadmap

Sixteen chapters, one term. Chapters 1–3 build the array/searching/sorting foundation; Chapter 4 is Quiz 1. Chapters 5–7 cover stacks, queues, pointers/functions, and singly linked lists; Chapter 8 is the Midterm. Chapters 9–11 cover doubly linked lists, recursion, and trees; Chapter 12 is Quiz 2. Chapters 13–15 cover graphs, heaps, and hashing — the material every earlier chapter has been building toward — and Chapter 16 is the Final Exam. Assessment chapters (4, 8, 12, 16) test the three content chapters immediately before them and ship no code/ subfolder of their own.

1.1.1 Meet Pustaka Ceria

Rather than switching examples every chapter, this course follows one running case study from Chapter 1 all the way through Chapter 15: Pustaka Ceria (Indonesian for “Cheerful Library”), a small, friendly campus library. Pustaka Ceria has a real, ordinary problem: it needs to store its book catalog, look books up by code or title, keep the catalog sorted for browsing, let a librarian push and pop a stack of “recently returned, not yet reshelved” books, walk a queue of hold requests in the order they arrived, organize its whole collection as a browsable tree of categories, model which books are frequently borrowed together as a graph, and — once the catalog gets large — look a book up by its code in constant time with a hash table.

Every one of those needs maps directly onto a chapter later in this course. Chapter 1 does the most basic version of the very first need: give every book a fixed, predictable shape in memory (a `struct Book`), and store a handful of them contiguously in an array. Nothing about Pustaka Ceria is based on a real institution — it exists purely to give every chapter's code a shared, recognizable cast of data to work with, instead of a new invented example each time.

1.2 C/C++ Refresher: Variables, Memory, and a Pointer Preview

This section is a refresher, not new material — you have almost certainly written variables, primitive types, and simple functions before. What matters for this course is the mental model underneath them: every variable a C/C++ program declares occupies some real, addressable location in the computer's memory, and understanding that location is what the rest of this chapter (and, more directly, Chapter 6's full pointer coverage) builds on.

1.2.1 Variables and Primitive Types

A variable declaration such as `int age = 20;` does two things at once: it reserves a fixed-size block of memory (4 bytes, for a typical `int`), and it gives that block a name — `age` — that the rest of the program can use instead of tracking the raw memory location by hand. C/C++'s primitive types each reserve a different, fixed number of bytes: `char` (1 byte), `int` (commonly 4 bytes), `float` (4 bytes), `double` (8 bytes), and so on — the exact sizes are implementation-defined, but `sizeof(type)` always

tells you the true answer on the machine you are compiling for, which is exactly the tool this chapter leans on.

1.2.2 Stack vs. Heap, Conceptually

A running C/C++ program has (at least) two very different regions of memory it uses for variables. The stack is where ordinary local variables live — every variable you declare inside a function (like `catalog` in this chapter's demo programs) is placed on the stack automatically when the function starts, and is automatically reclaimed the instant the function returns. The stack is fast and requires no manual bookkeeping, but its size is fixed and modest (often a few megabytes), and a variable's lifetime is tied strictly to the function it was declared in. The heap, by contrast, is memory a program requests explicitly at runtime (in C++, with `new`; in C, with `malloc`) and must release explicitly when done (`delete` / `free`) — it survives past the function that created it, at the cost of the programmer now being responsible for its lifetime by hand.

> **Why only a preview here**
> Every `struct Book` this chapter creates lives on the stack — declared as an ordinary local array, with no `new`/`malloc` anywhere in sight. That is deliberate: this chapter's whole job is the memory model of contiguous arrays, and mixing in heap allocation and pointer ownership at the same time would blur two separate lessons together. Chapter 6 (Pointers & Functions) is where heap allocation, `new`/`delete`, and pass-by-reference get the full treatment they deserve.

1.2.3 Addresses: the `&` Operator and a Pointer Preview

Every variable, once it exists, has two things worth asking about: its value, and its address — where in memory it lives. C/C++ gives you the address-of operator, `&`, to ask that second question directly: given `int age = 20;`, the expression `&age` is the memory address where `age`'s 4 bytes live, not the value 20 itself. A pointer is simply a variable whose own value is an address rather than an ordinary number; `int *p = &age;` declares `p` as "a pointer to `int`" and initializes it to hold `age`'s address, and `*p` (dereferencing `p`) reads the `int` stored at that address — which is 20, the same value `age` itself holds.

```
int age = 20;
int *p = &age;      // p now holds age's address

printf("%d\n", age);    // 20  -- the value
printf("%p\n", (void*)&age);  // e.g. 0x7ffd1234abcd -- the address
printf("%d\n", *p);   // 20  -- dereferencing p reads the same value
```

> **This is a preview, not the full topic**
> This section introduces just enough of `&` and pointer syntax to make Section 1.3's address arithmetic readable — it deliberately stops well short of pointer arithmetic on arbitrary types, passing pointers to functions, or dynamic allocation. Chapter 6 is where pointers become a first-class topic in their own right; treat every pointer example between here and then as read-only background, not something you are expected to write from scratch yet.

1.2.4 A Foundational Record: struct Book

Pustaka Ceria's catalog is a collection of books, and every book needs the same four pieces of information recorded about it: a short identifying code, a title, an author, and a category. C/C++'s `struct` keyword lets us describe exactly that shape once, as a new type:

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

Every field here is a fixed-size `char` array rather than a dynamically-sized string — deliberately, for this chapter: a fixed-size struct with no pointers inside it is the simplest possible thing to lay out contiguously in memory, which is exactly what Section 1.3 needs to explain cleanly. `struct Book` is the one record type every later Pustaka Ceria chapter reuses — as a node's payload in a linked list (Chapter 7), as the value stored in a tree (Chapter 11), as the record a hash table maps a code to (Chapter 15) — so it is worth having exactly right from here on.

1.3 Memory Layout and Address Arithmetic

Declare an array of `struct Book` — `struct Book catalog[8];` — and C/C++ makes a very specific promise: all 8 `Book`s are laid out back-to-back in one contiguous block of memory, with no gaps and no reordering. `catalog[0]` occupies the first `sizeof(struct Book)` bytes of that block, `catalog[1]` occupies the next `sizeof(struct Book)` bytes, and so on. Figure 1.1 shows this for our 4-field `Book` struct.

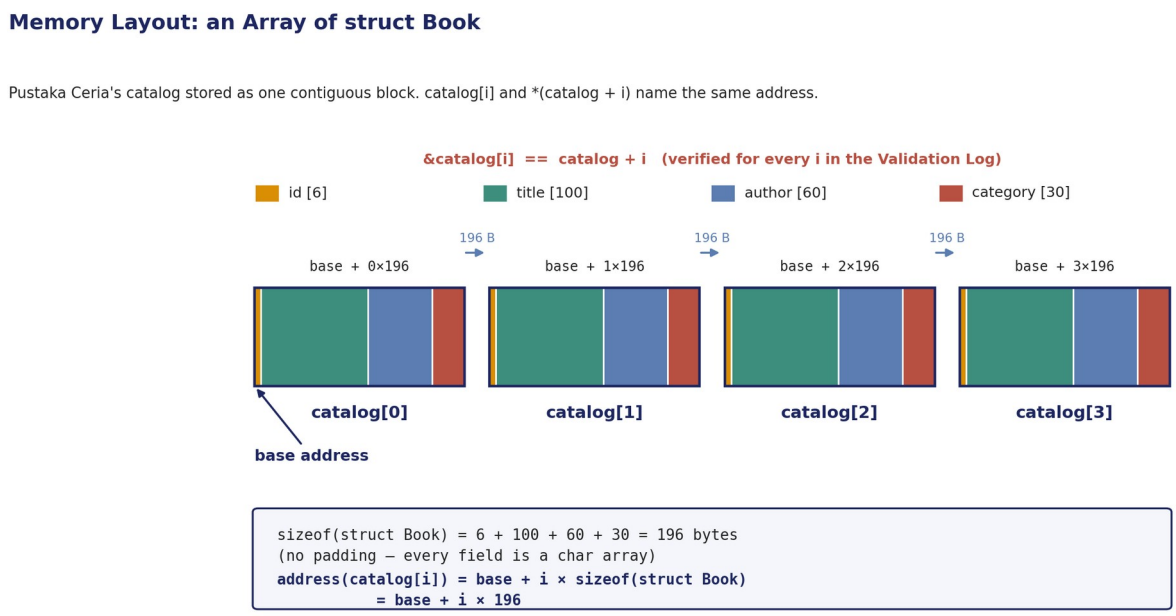


Figure 1.1 — An array of struct Book laid out contiguously in memory, with byte offsets for each element.

This layout is what makes array indexing, `catalog[i]`, equivalent to pointer arithmetic, `*(catalog + i)`: the array name `catalog` decays to a pointer to its first element, and adding `i` to that pointer advances

it by exactly `i * sizeof(struct Book)` bytes — not `i` bytes — because pointer arithmetic on a `T*` always moves in units of `sizeof(T)`. That is precisely why the formula in Figure 1.1 reads `address(catalog[i]) = base + i × sizeof(struct Book)`: the compiler is doing exactly that multiplication every time `catalog[i]` appears in your code.

> **sizeof(struct Book) has no hidden padding here**
> For most structs, the compiler inserts invisible padding bytes between fields so that each field starts at an address aligned to its own type's requirement (an `int` field, for instance, is usually placed at a 4-byte-aligned offset). `struct Book` sidesteps this entirely: every field is a `char` array, and `char` has an alignment requirement of exactly 1 byte, so the compiler has nothing to align and adds no padding at all. `sizeof(struct Book)` is therefore exactly 6 + 100 + 60 + 30 = 196 bytes — confirmed by the compiled program's own output in Appendix 1.A. Once a struct mixes `char` fields with `int`/`double` fields, `sizeof` can be noticeably larger than the sum of the field sizes; that is a topic for a later chapter, not this one.

`catalog[i]` and `*(catalog + i)` are not just equivalent in what they compute — they are the exact same address, every time, which is something you can verify directly rather than take on faith:

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

1.4 1D Arrays: Declaration, Population, and Traversal

With `struct Book` defined and its memory layout understood, declaring Pustaka Ceria's whole catalog as a 1D array is a single line:

```
struct Book catalog[N];   // N = 8 for this chapter's demo
```

Populating it means filling in each element's fields — this chapter's `populateBook()` helper copies four C-strings into a `Book`'s fixed-size fields, always leaving every field null-terminated even if a caller's string is longer than the field can hold (truncating safely rather than overflowing the 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);
}
```

Traversing the catalog — visiting every element exactly once, in order — is the ordinary `for` loop every C/C++ program uses over an array:

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

This one loop shape — declare, populate, traverse — is the pattern every later chapter's array-based code (Chapter 2's searching, Chapter 3's sorting) builds directly on top of.

1.5 2D Arrays: Shelves as a Grid

Pustaka Ceria's building has a natural physical structure a flat 1D catalog does not capture: books sit on physical shelves, and each shelf has a fixed number of slots. A 2D array models this directly — `struct Book catalog2D[SHELVES][SLOTS];` declares a grid where `catalog2D[s][p]` is the book on shelf `s`, slot `p`.

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

C/C++ lays a 2D array out row-major: the entire first row (shelf 0's `SLOTS` books) occupies the first `SLOTS × sizeof(struct Book)` bytes, immediately followed by the entire second row (shelf 1), with no gap in between — exactly as if the 2D grid had been “unrolled” into one long 1D array in row order. Figure 1.2 shows this directly: the same 8 books, first as a 2-shelf grid, then unrolled into the single contiguous block they actually occupy in memory.

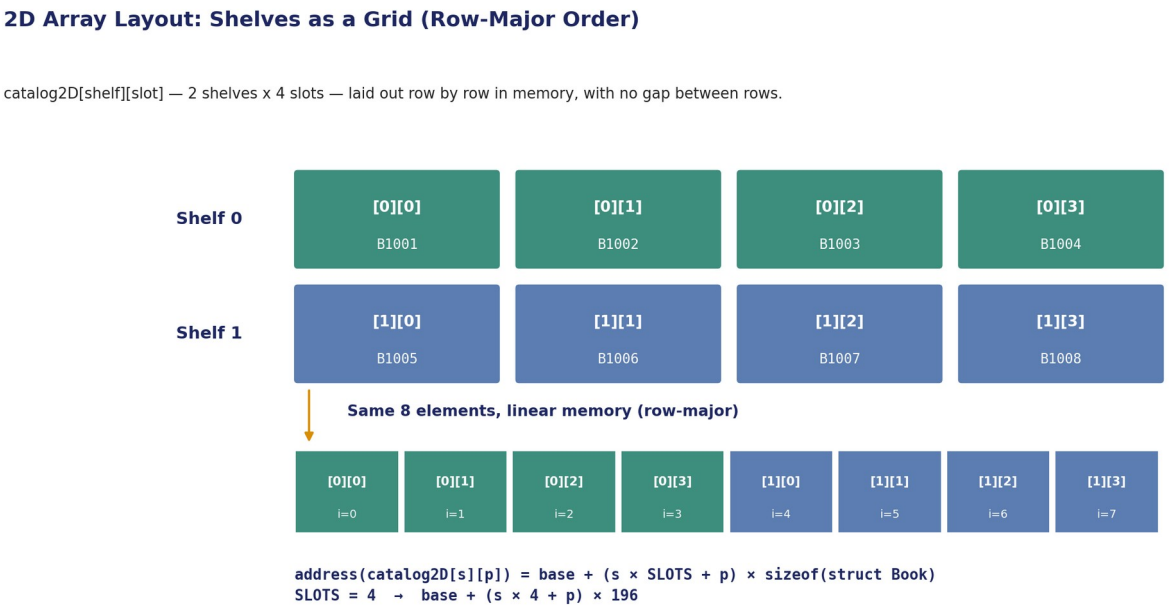


Figure 1.2 — A 2D array of struct Book (shelves × slots) and its row-major linear memory layout.

Generalizing Section 1.3's 1D address formula to two dimensions: element `catalog2D[s][p]` is the `(s × SLOTS + p)`-th element of that unrolled 1D layout, so its address follows the same base-plus-offset shape, just with a row-major linear index substituted in for `i`:

```
// address(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], for every s, p
```

Row-major is a choice, not a law of nature

C/C++ specifically guarantees row-major layout for multidimensional arrays — the rightmost index varies fastest as you walk through memory in address order. Some other languages (Fortran, for instance) default to column-major instead, where the leftmost index varies fastest. The formula in this section is specific to C/C++'s row-major guarantee; porting address-arithmetic code between languages with different array layouts is a genuine, easy-to-miss source of bugs.

1.6 Appendix 1.A — Validation Log

Both demo programs below were compiled with `g++ -Wall -Wextra -std=c++17` (zero warnings from either) and actually executed; the transcript below is the real, unedited terminal output — not hand-written “expected” output. Addresses will differ on every run (the OS randomizes stack addresses for security), but the relationships the chapter claims — `&catalog[i] == catalog + i`, and a constant 196-byte gap between elements — hold on every run, as shown.

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)...
```

Honesty note

Every value shown above (the 196-byte struct size, every “yes”, every matching byte offset) came directly from a real compile-and-run in this chapter's build environment — no discrepancy or bug was found in this chapter's own code, and none is being manufactured here for effect; this course's standing policy is to report validation results exactly as they occurred, whether or not that includes a bug.

1.7 Chapter Summary and Key Takeaways

- Data Structure studies how to lay data out in memory so that the operations a program needs (insert, delete, search, sort, traverse) stay fast as the data grows; this course's 16 chapters build up a toolbox of layouts — arrays, lists, stacks/queues, trees, graphs, and hash tables — for exactly that purpose.
- Pustaka Ceria, a small campus library needing to store, search, sort, and organize its book catalog, is this course's running case study from Chapter 1 through Chapter 15; `struct Book` (id/title/author/category) is its foundational record type.
- Every variable has both a value and an address; `&x` gives you `x`'s address, and a pointer is simply a variable whose value is an address. This chapter previews only enough pointer syntax to read address arithmetic — full pointer coverage is Chapter 6.
- An array of a struct is laid out contiguously in memory with no gaps: `address(catalog[i]) = base + i × sizeof(struct Book)`, and `catalog[i]` and `*(catalog + i)` are always the exact same address.
- A 2D array is laid out row-major: the entire first row is stored before the second row begins, so `address(catalog2D[s][p]) = base + (s × SLOTS + p) × sizeof(struct Book)`.

1.8 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Chapter 2 (Searching).

1. In your own words, explain why the choice of data structure can matter for a program's performance even when the data structure stores exactly the same information either way. Give one concrete example (it does not have to involve Pustaka Ceria).
2. `struct Book` uses fixed-size `char` arrays for every field instead of, say, a dynamically-sized string type. Explain what this buys the chapter's memory-layout explanation, and name one limitation of representing a title this way (hint: what happens to a title longer than 99 characters?).
3. Given `struct Book catalog[10];`, write the expression (using pointer arithmetic, not indexing) that refers to the same element as `catalog[6]`, and explain in one sentence why the two expressions are guaranteed to be the same address.
4. Suppose `struct Book` gained a fifth field, `int yearPublished;`, inserted between `id` and `title`. Would you expect `sizeof(struct Book)` to still equal the simple sum of the field sizes? Explain why or why not, referring to alignment.
5. For `struct Book catalog2D[3][5];` (3 shelves, 5 slots), derive the address-arithmetic formula for `catalog2D[2][3]` in terms of the base address and `sizeof(struct Book)`, following the same row-major reasoning as Section 1.5.
6. Pustaka Ceria's librarian wants to add a ninth book to the 8-book 1D catalog from Section 1.4 without changing anything about how the array was declared. Explain what practical

problem this runs into, and name (without implementing it) the C/C++ feature that Chapter 6 introduces to solve exactly this kind of problem.

References

[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 (accessed August 2026).

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

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

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 2

Searching

Four ways to answer the question every catalog eventually gets asked: "is this book here, and if so, where?" Every line of code in this chapter was actually compiled with g++ -Wall -Wextra and run — the exact terminal transcript is reproduced in Appendix 2.A.

Learning Objectives — after this chapter you will be able to:
(1) Explain what a search algorithm's "found / not found" return contract means, and why returning a position (an index), not just a yes/no answer, is almost always the more useful contract. (2) Implement and trace sequential (linear) search over Pustaka Ceria's catalog, by id and by title, and state its worst-case cost. (3) Explain sentinel search as a micro-optimization of sequential search, and precisely what it does and does not save. (4) Implement binary search over a catalog already sorted by id, trace its low/mid/high narrowing by hand, and explain why sortedness is a genuine prerequisite, not an implementation detail. (5) Implement interpolation search over a sorted numeric key, and explain why it is a poor fit for Pustaka Ceria's string-valued book ids. (6) State the formal definition of Big-O notation, and use it to compare all four algorithms' best-, average-, and worst-case behavior.

2.1 Why Searching Matters: the Found / Not Found Contract

Chapter 1 gave Pustaka Ceria's catalog a shape in memory — a `struct Book` and a contiguous array to hold several of them. The very first thing anyone actually does with a catalog, though, is not admire its memory layout; it is ask it a question: "is there a book with id `B1006` in here, and if so, which one is it?" That question, asked over and over, in every system that keeps any kind of collection, is what this chapter calls searching.

Every searching algorithm in this chapter answers exactly the same two-part question, in exactly the same shape, so that the algorithm underneath can be swapped out later without changing anything that calls it: given a collection and a key to look for, return the position (the index) of a matching element if one exists, or a clearly-distinguishable "not found" signal if none does. This chapter's code returns `-1` for "not found" — a value no valid array index can ever equal — rather than returning a plain `true`/`false`. A `bool` can only tell a caller that a book exists somewhere; an index tells the caller exactly which element it is, so the very next line of code can print its title, edit its category, or hand it to a different function — with no second search required to relocate it.

"Found or not" vs. "found where"
It is tempting to write a search function that returns `bool` — it reads cleanly at the call site (`if (bookExists(catalog, N, "B1006")) { ... }`). But almost every real caller's very next step is "...and now do something with that book," which needs its index or its address, not just the fact that it exists. Every function in this chapter's `search.h` returns an `int` index (or `-1`), for exactly this reason — it is a strictly more useful contract, at no extra cost to write.

This chapter covers four algorithms that all answer that same found/position contract, each trading a different assumption about the data for a different amount of speed: sequential search (Section 2.2) assumes nothing about the catalog's order and is always available as a fallback; sentinel search (Section 2.3) is a small, honest optimization of sequential search's inner loop; binary search (Section 2.4) assumes the catalog is already sorted and is dramatically faster in exchange; and interpolation search (Section 2.5) assumes the sort key is numeric and roughly evenly spread, and is faster again when that assumption actually holds. Section 2.6 introduces Big-O notation formally so all four can be compared on the same footing.

2.2 Sequential (Linear) Search

Sequential search is the algorithm every one of us reaches for without ever being taught it: to find something in an unordered pile, look at the first item, then the second, then the third, and so on, until you find it or you run out of items. It makes no assumption whatsoever about how the catalog is arranged — which is exactly why it is the first algorithm this chapter covers, and the one every other algorithm in this chapter is compared against.

```
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` is the identical shape, comparing `catalog[i].title` instead of `catalog[i].id` — the algorithm does not care which field it is comparing, only that some field can be compared for equality. Both functions accept an optional `comparisons` counter so this chapter's demo program can report a genuine, measured comparison count rather than an estimated one.

> **Best case, worst case, and why the worst case is the one that matters**
> If the target is `catalog[0]`, sequential search finds it in exactly 1 comparison — its best case. If the target is `catalog[n-1]`, or is not in the catalog at all, sequential search must make exactly `n` comparisons before it can conclude anything — its worst case. As Section 2.6 makes precise, it is the worst case that Big-O notation describes, because a caller cannot know in advance which case a given search will land in, and a catalog that grows from 8 books to 8,000 will make the worst case 1,000× more expensive whether or not any particular search happens to get lucky.

This chapter's `demo_sequential.cpp` runs exactly this comparison count for four real searches over Pustaka Ceria's 8-book catalog: the first element (`"B1001"`, 1 comparison), a middle element (`"B1005"`, 5 comparisons), the last element (`"B1008"`, 8 comparisons), and an id that is not in the catalog at all (`"B1099"`, 8 comparisons) — the real, compiled, and run output is reproduced in Appendix 2.A.

2.3 Sentinel Search: Removing the Bounds Check

Sentinel search is not a different algorithm from sequential search — it is the same “check elements from the front until you find a match” idea, with one small, deliberate change to the loop. Ordinary sequential search's loop condition, `i < n`, is tested on every single iteration purely to protect against reading past the end of the array. Sentinel search removes that test entirely, by first copying the target itself into one extra slot at the very end of the array (index `n`, the sentinel), which guarantees some element will match — so the loop's only remaining job is to check for a match, never to check whether it has run out of room to look.

```
int sentinelSearchById(Book catalogWithSentinel[], int n, const char *id, long
*comparisons) {
    // catalogWithSentinel[n] already holds a copy of the target (the
    // sentinel), planted there by the caller before this runs.
    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 means only the sentinel matched -> not a
real hit
}
```

What sentinel search actually saves — and what it does not

Running the exact same searches through both `sequentialSearchById` and `sentinelSearchById` (this chapter's `demo_sentinel.cpp` does exactly this, side by side) produces IDENTICAL comparison counts for every id tried — sentinel search does not look at fewer elements, and does not change the algorithm's O(n) worst case. What it removes is a second, separate test (`i < n`) that ordinary sequential search's loop re-evaluates on every iteration in addition to the match test. That is a real, honest micro-optimization — one fewer machine test per iteration, which can matter in a tight loop over a very large array — but it is a constant-factor improvement, not a change in the algorithm's asymptotic behavior. This chapter reports it exactly that way, rather than manufacturing a timing benchmark to make the difference look bigger than it is.

The trade-off sentinel search accepts for this saving is a real one worth naming: it needs a writable array one element larger than the data itself (to hold the sentinel), and it destroys whatever was at index `n` before the search starts. For Pustaka Ceria's read-only catalog lookups, that trade-off is easy to make (this chapter's demo simply works on a throwaway copy); for a catalog you are simultaneously mutating, it would need more care.

2.4 Binary Search: Halving the Problem Each Step

Binary search is the first algorithm in this chapter that is not a variation on “check every element” — instead, it exploits order. If the catalog is sorted by id, then comparing the target id against the id sitting in the exact middle of the array tells you, in one comparison, which HALF of the array the target

could possibly be in — the other half can be discarded entirely, without ever looking at a single element inside it. Repeating this — look at the middle of what is left, discard half again — narrows the search window down to nothing (or to a match) in a number of steps that grows only logarithmically with the catalog's size, not linearly.

> **Binary search's real prerequisite: the array must already be sorted**
> This is not a minor implementation detail — it is the whole reason binary search works at all. Sortedness is what makes "the target's id is smaller than the middle element's id" a reliable, one-comparison signal that the entire right half of the array can be ignored; over an unsorted array, that inference would simply be false, and binary search would silently return wrong answers. HOW to sort an array from scratch (Bubble, Selection, Insertion, Quick, Merge, Heap sort) is Chapter 3's own topic, not this chapter's — here, sorting is treated honestly as a one-line prerequisite step, using the C++ standard library's own `std::sort`, exactly the way a working programmer would reach for it before hand-rolling a sort just to satisfy this one function.

`demo_binary.cpp` makes this dependency explicit rather than hiding it: it starts from Pustaka Ceria's catalog in the order the books actually arrived (which has nothing to do with their id numbers), sorts a copy of it by id with `std::sort`, and only then runs binary search.

```
std::sort(sorted, sorted + N, [](const Book &a, const Book &b) {
    return std::strcmp(a.id, b.id) < 0;
});
// NOTE: std::sort is a library call, not a hand-rolled sorting
// algorithm -- Chapter 3 covers Bubble/Selection/Insertion/Quick/
// Merge/Heap sort from scratch. Here it is only the honest
// prerequisite step binary search genuinely needs.
```

With the array genuinely sorted, binary search itself is a short loop that maintains a shrinking `[low, high]` window and always probes its midpoint:

```
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;             // match!
        else if (cmp < 0) low = mid + 1;       // target is to the right
        else high = mid - 1;                   // target is to the left
    }
    return -1;   // low > high -- window is empty, target is not present
}
```

Figure 2.1 traces this loop over Pustaka Ceria's sorted catalog for a real search, `id = "B1001"`, using the actual output `demo_binary.cpp` prints — not a hand-simulated example.

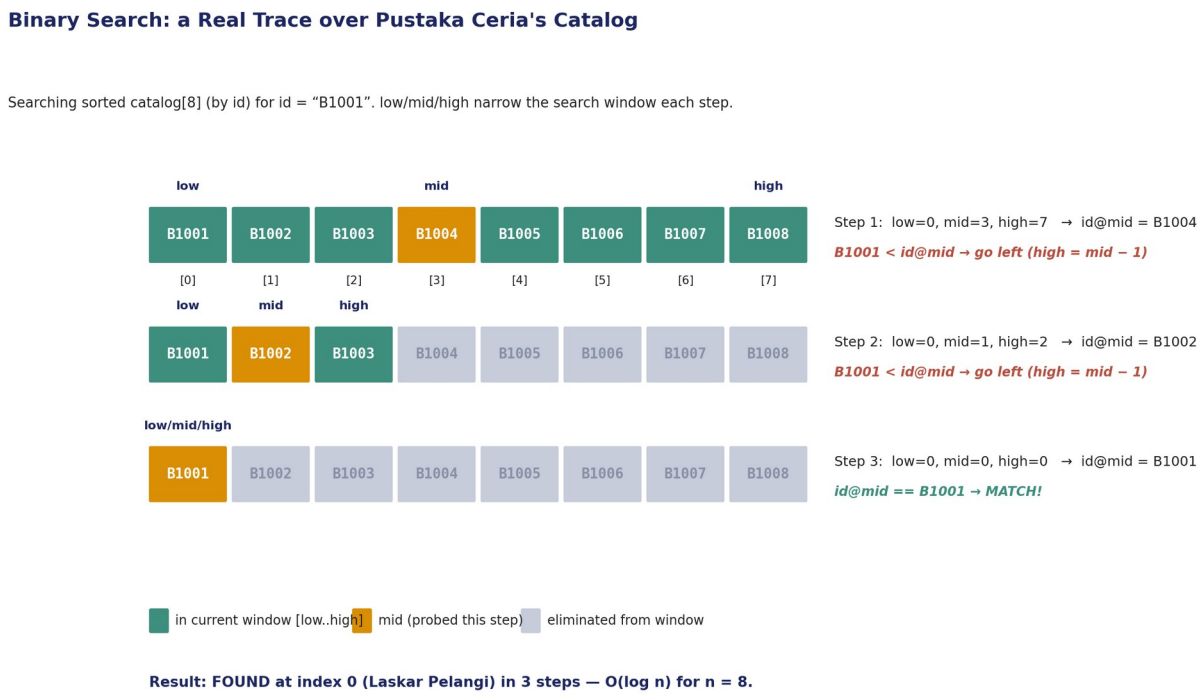


Figure 2.1 — Binary search's low/mid/high window narrowing over Pustaka Ceria's sorted catalog, tracing a real search for id = “B1001”.

> **mid = low + (high - low) / 2, not (low + high) / 2**
> Both formulas compute the same midpoint mathematically, but `low + (high - low) / 2` is the one worth writing out of habit: for very large `low` and `high` values (near the limit of what an `int` can hold), `low + high` can silently overflow before the division even happens, while `low + (high - low)` cannot, because `high - low` is always small and non-negative in a valid search window. It is a small, defensive habit, not something this chapter's 8-element catalog will ever trigger — but it is the version worth learning correctly the first time.

Notice, in Figure 2.1, exactly how few steps binary search needed: 3 steps to find `id = "B1001"` among 8 books, where sequential search's worst case over the same 8 books needs up to 8 comparisons. That gap only widens as the catalog grows — Section 2.6 makes precise exactly how much.

2.5 Interpolation Search: Guessing Smarter Than the Midpoint

Binary search always probes the exact middle of whatever window remains, regardless of what the values actually look like. Interpolation search refines that single decision: instead of always guessing the middle, it estimates WHERE in the window the target is likely to be, based on the target's value relative to the values at the window's two ends — much like how a person looking up a name in a phone book flips toward the front of the book for a name starting with “B” and toward the back for one starting with “T”, instead of always opening to the exact middle first.

```
// Estimate a probe position assuming values are roughly uniformly
// spread between arr[low] and arr[high] -- instead of binary
// search's fixed mid = low + (high - low) / 2:
int pos = low + (int)(((double)(high - low) * (key - arr[low]))
                      / (arr[high] - arr[low]));
```

Why Pustaka Ceria's book ids are the wrong key for this algorithm

Interpolation search's whole estimate rests on being able to ask “roughly how far is `key` from `arr[low]`, as a fraction of the distance from `arr[low]` to `arr[high]`?” — a question that only makes sense for a NUMERIC key, where subtraction and division are meaningful. Book ids like `"B1001"` and `"B1008"` are strings; `strcmp` can tell you their ORDER, but “how far apart” two strings are has no natural numeric answer the way it does for two integers. Applying the interpolation formula to string ids would not even compile without an ad-hoc numeric encoding, and any such encoding would be arbitrary rather than meaningful — so this chapter's interpolation search demo deliberately switches to a genuinely numeric key: each book's PUBLICATION YEAR, held in a separate `int` array parallel to (but not stored inside) `struct Book`. `struct Book` itself is unchanged from Chapter 1.

`demo_interpolation.cpp` searches a sorted array of the same 8 Pustaka Ceria books' publication years (1975 through 2011) for a real key, `2005`, which happens to sit near the high end of that range. Because the values are roughly evenly spread, interpolation search's very first probe lands close to the correct position, in contrast to binary search's fixed midpoint — the algorithm's real, printed trace (reproduced in Appendix 2.A) finds it in 2 steps.

Interpolation search also has to handle a case binary search never needs to consider: what if the current window's low and high values are equal? Dividing by `arr[high] - arr[low]` would then divide by zero. This chapter's implementation checks for that case explicitly and falls back to probing the window's start — always safe, if not always “smart”:

```
if (arr[high] == arr[low]) {
    pos = low;   // avoid a divide-by-zero; still correct, just not "smart"
} else {
    pos = low + (int)(((double)(high - low) * (key - arr[low])) / (arr[high] -
arr[low]));
}
```

2.6 Comparing the Four Algorithms: Introducing Big-O Notation

Every section so far has described an algorithm's cost informally — “n comparisons in the worst case,” “a number of steps that grows only logarithmically.” Big-O notation is the formal language computer science uses to make statements like that precise and comparable, and this course introduces it here, in its first chapter about algorithms rather than data layout, because searching is the first place a difference in growth rate becomes a genuinely practical concern.

> **Big-O notation, formally**
> Big-O notation describes how an algorithm's worst-case running time (or number of basic operations, such as comparisons) grows as the input size `n` grows, IGNORING constant factors and lower-order terms. An algorithm is O(f(n)) if, for large enough `n`, its running time is bounded above by some constant multiple of f(n). O(1) ("constant") means the cost does not grow with `n` at all. O(log n) ("logarithmic") means the cost grows very slowly — doubling `n` adds only one more step. O(n) ("linear") means the cost grows in direct proportion to `n`. Big-O answers the question “what happens as `n` gets large?” — it is deliberately silent on the exact number of operations any one run takes, or on which algorithm is faster for some small, particular `n`; those questions matter too, but Big-O is not the tool for them.

With that definition in hand, here is what this chapter's four algorithms actually cost, in the worst case, as Pustaka Ceria's catalog grows from 8 books to 8,000 to 8 million:

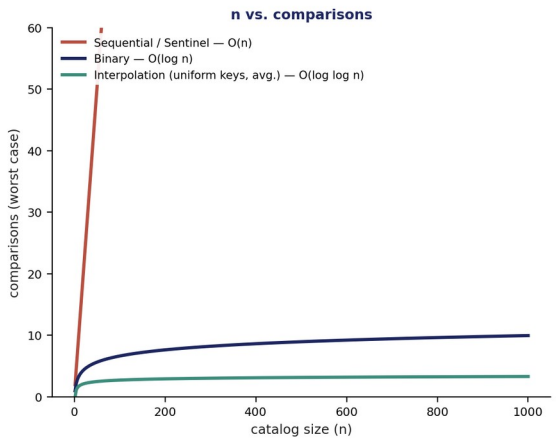
Algorithm	Best case	Average case	Worst case	Needs sorted input?
Sequential search	O(1)	O(n)	O(n)	No
Sentinel search	O(1)	O(n)	O(n)	No
Binary search	O(1)	O(log n)	O(log n)	Yes
Interpolation search	O(1)	O(log log n)*	O(n)*	Yes

**Interpolation search's average-case O(log log n) assumes the numeric keys are roughly uniformly distributed, as Section 2.5's publication-year example deliberately was; on a badly skewed distribution of keys, its worst case degrades all the way to O(n) — no better than sequential search, in the worst possible arrangement of data.*

Figure 2.2 puts the two growth rates that matter most here — O(n) and O(log n) — on the same axes, alongside interpolation search's O(log log n) curve, so the gap between them is visible directly rather than only asserted in a table.

Big-O Comparison: the Four Search Algorithms

How each algorithm's worst-case comparison count grows as the catalog size n grows.



Algorithm	Best	Average	Worst	Sorted?
Sequential	O(1)	O(n)	O(n)	No
Sentinel	O(1)	O(n)	O(n)	No
Binary	O(1)	O(log n)	O(log n)	Yes
Interpolation	O(1)	O(log log n)*	O(n)*	Yes

**Interpolation search's O(log log n) average case assumes roughly uniformly-distributed numeric keys; on a badly skewed distribution it degrades toward O(n).*

Big-O notation, defined

Big-O describes how an algorithm's worst-case work grows as input size n grows — O(1) constant, O(log n) logarithmic, O(n) linear — ignoring constant factors and lower-order terms. It answers “what happens as n gets large?”, not “which exact number of steps does this run take?”.

Figure 2.2 — Big-O growth-rate comparison of the four search algorithms, and a summary of their best/average/worst-case behavior.

> **Why this table's "average case" column is not the whole story**
>
> A catalog of 8 books, like this chapter's demos use throughout, is far too small for any of these growth-rate differences to matter in wall-clock time — the whole point of Big-O is that the differences only become dramatic once `n` is large. Sequential search over 8 elements and binary search over 8 elements both finish, for a human, instantly; the same two algorithms over 8 million elements do not. Reading Figure 2.2's curves is the whole exercise — do not expect this chapter's tiny demo catalog to show a measurable speed difference by itself.

2.7 Appendix 2.A — Validation Log

All four demo programs below were compiled with `g++ -Wall -Wextra -std=c++17` (zero warnings from all four) and actually executed; the transcripts below are the real, unedited terminal output — not hand-written “expected” output.

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)
```

Note the honest asymmetry in the last row: for an id that is genuinely absent, sentinel search's own count (9) is one MORE than sequential search's (8) — it still has to walk past all 8 real elements and then hit the sentinel itself at index 8 to terminate. Sentinel search's real saving was never "fewer comparisons"; Section 2.3 explains exactly what it does save.

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)
```

Honesty note

Every count and every step shown above (all 8 comparison counts, the sentinel's genuine +1 in the not-found case, every low/mid/high and low/pos/high row) came directly from a real compile-and-run in this chapter's build environment — no discrepancy or bug was found in this chapter's own code, and none is being manufactured here for effect; this course's standing policy is to report validation results exactly as they occurred, whether or not that includes a bug.

2.8 Chapter Summary and Key Takeaways

- Every search algorithm in this chapter answers the same found/position contract: given a key, return the matching element's index, or a clearly-distinguishable “not found” signal (this chapter uses -1) — not just a bool.
- Sequential (linear) search checks elements from the front, one at a time; it needs no assumption about order, and costs O(n) in the worst case (target absent, or at the very end).
- Sentinel search is a micro-optimization of sequential search: planting a copy of the target at index n removes the loop's separate bounds check. Its comparison count matches plain sequential search exactly — the saving is one fewer machine test per iteration, not fewer elements examined.
- Binary search requires the array to already be sorted by the search key; given that, it halves its search window every step, costing O(log n) in the worst case. Hand-rolling a sorting algorithm is Chapter 3's own topic — this chapter used std::sort as an honest one-line prerequisite.
- Interpolation search refines binary search's fixed midpoint into an estimated position, assuming a roughly uniformly-distributed NUMERIC key; it is a poor fit for string-valued keys like Pustaka Ceria's book ids, which is why this chapter's demo used publication years instead.
- Big-O notation describes how an algorithm's worst-case cost grows as input size n grows, ignoring constant factors: O(1) constant, O(log n) logarithmic, O(n) linear. The four algorithms in this chapter span O(1) best case up through O(n) worst case, and the gap between O(n) and O(log n) only widens as n grows large.

2.9 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Chapter 3 (Sorting).

1. Explain, in your own words, why returning an index (or -1) is a more useful contract for a search function than returning a plain bool. Describe one concrete situation where a bool would force a caller to do extra, avoidable work.
2. Sequential search's best case is O(1) and its worst case is O(n). Give the specific position of the target that produces each case, and explain why Big-O notation is defined in terms of the worst case rather than the best case.
3. Sentinel search's demo in this chapter shows identical comparison counts to plain sequential search for every id searched. If sentinel search does not reduce the number of comparisons, explain precisely what it does save, and under what circumstances that saving would actually matter in a real program.
4. Why can binary search not simply be run directly on Pustaka Ceria's catalog in its arrival order? Referring to Figure 2.1, explain what would go wrong (concretely, which comparison would give a misleading answer) if the low/mid/high logic were applied to an unsorted array.

5. Explain, using arr[low] and arr[high], why interpolation search's probe-position formula is not meaningful for Pustaka Ceria's book ids (“B1001”, “B1002”, …) the way it is for the publication-year array this chapter actually used.
6. A catalog grows from 8 books to 8,000 books (1,000x larger). Using the Big-O costs from Section 2.6's table, estimate how many times MORE comparisons sequential search's worst case will need, and how many MORE steps (not times more — how many additional steps) binary search's worst case will need. What does this contrast tell you about why the choice of search algorithm matters more as data grows?

References

[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. (Big-O notation, Chapter 3.)

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

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

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 3

Sorting

Seven ways to put a catalog in order — the very step Chapter 2's binary and interpolation search quietly leaned on `std::sort` for. Every line of code in this chapter was actually compiled with g++ -Wall -Wextra and run — the exact terminal transcript is reproduced in Appendix 3.A.

Learning Objectives — after this chapter you will be able to:
(1) Explain why sortedness is a genuine prerequisite for binary and interpolation search (Chapter 2), not an incidental detail, and recap Big-O notation (introduced formally in Chapter 2) well enough to use it without re-deriving it. (2) Implement and trace Bubble, Exchange, and Selection sort over a Pustaka Ceria book subset, and explain precisely how their comparison PATTERNS differ even when their comparison COUNTS are identical. (3) Implement and trace Insertion sort, and demonstrate — with a real measured run, not just an assertion — its genuinely adaptive O(n) best case on nearly-sorted input. (4) Implement Quicksort with a consistent Lomuto partition scheme, trace its recursive partitioning, and explain honestly why its worst case is O(n^2) despite an O(n log n) average case. (5) Implement Merge sort, trace its recursive split/merge structure, and explain the O(n) space it trades for a worst case that is exactly as good as its average case. (6) Implement a self-contained array-based Heap sort (build-heap + repeated root extraction) and state its O(n log n) complexity, without yet needing the full heap ADT that Chapter 14 covers. (7) Compare all seven algorithms on time complexity (best/average/worst), auxiliary space, and stability, and use that comparison to justify why more than one sorting algorithm is worth knowing.

3.1 Recap: Why Sorting Matters, and What We Already Know

Chapter 2 asked a simple question of Pustaka Ceria's catalog — “is this book here, and where?” — and answered it four different ways. Two of those four answers, binary search and interpolation search, were dramatically faster than the other two, but both came with a catch: they only work on a catalog that is ALREADY SORTED. Chapter 2's own demo code did not pretend otherwise. `demo_binary.cpp` called `std::sort` before it called `binarySearchById`, with a code comment promising that “Chapter 3 covers Bubble/Selection/Insertion/Quick/Merge/Heap sort from scratch” — a promise this chapter now keeps.

So this chapter is, in one sense, the most practical one so far: it is entirely about how to actually produce the sorted order that Chapter 2 assumed. It covers seven algorithms in total — four that compare and swap elements directly within the array (Bubble, Exchange, Selection, Insertion), and three that divide the problem into smaller pieces first (Quicksort, Merge sort, Heap sort). All seven solve exactly the same problem — rearrange an array so its elements appear in ascending order by some key — and this chapter sorts Pustaka Ceria's catalog by TITLE throughout, the single most natural “put the shelf in order” task a librarian actually has.

Big-O notation — recap, not re-introduction

Chapter 2, Section 2.6 already defined Big-O notation formally: it describes how an algorithm's worst-case cost grows as input size n grows, ignoring constant factors and lower-order terms — O(1) constant, O(log n) logarithmic, O(n) linear, and so on. This chapter does not re-derive that definition; it simply uses it, adding two new growth rates to the vocabulary Chapter 2 already built: O(n^2) ("quadratic" — cost grows with the SQUARE of input size; doubling n roughly quadruples the cost) and O(n log n) (a genuinely different, much better, shape that four of this chapter's seven algorithms are built specifically to achieve instead of O(n^2)).

One more piece of vocabulary this chapter needs and Chapter 2 did not: STABILITY. A sorting algorithm is stable if two elements with EQUAL keys always keep their original relative order after sorting. For Pustaka Ceria this matters concretely: two books can share a title-adjacent first word (this chapter's own worked example has both “Filosofi Kopi” and “Fisika Dasar” — different titles, not equal keys, so stability does not apply to them specifically — but imagine instead two copies of the exact same title by different printings) — a stable sort guarantees that if copy A was listed before copy B before sorting, it still is afterward. Section 3.9's comparison table states which of the seven algorithms this chapter covers are stable and which are not, and why.

3.2 Bubble Sort

Bubble sort is the most direct possible reading of “keep swapping until things are in order”: repeatedly scan the array, comparing every pair of ADJACENT elements, and swap them if they are out of order. After one full pass over n elements, the single largest element is guaranteed to have “bubbled” all the way to the end of the array — so the next pass only needs to consider the first n-1 elements, and so on.

```
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)++;
            }
        }
    }
}
```

This chapter's Bubble sort has no early-exit flag — on purpose

A common textbook optimization adds a `bool anySwap` flag that breaks out of the outer loop the moment a full pass makes zero swaps, since that means the array is already sorted. This chapter's implementation deliberately omits it, so that Bubble sort's BEST case is exactly as expensive as its worst case: O(n^2) either way, always running every one of its n-1 passes. This is a genuine, honest design choice, not an oversight — it exists specifically to contrast with Insertion sort (Section 3.5), whose adaptive O(n) best case is real and measured, not merely possible with an added flag.

Figure 3.1, Panel A, traces pass 1 over a real 6-book subset of Pustaka Ceria's catalog (in its normal arrival order), showing exactly which of the 5 adjacent comparisons trigger a swap and which do not — and the single largest title of the six, “Negeri 5 Menara”, correctly ends up at the far right after just that one pass.

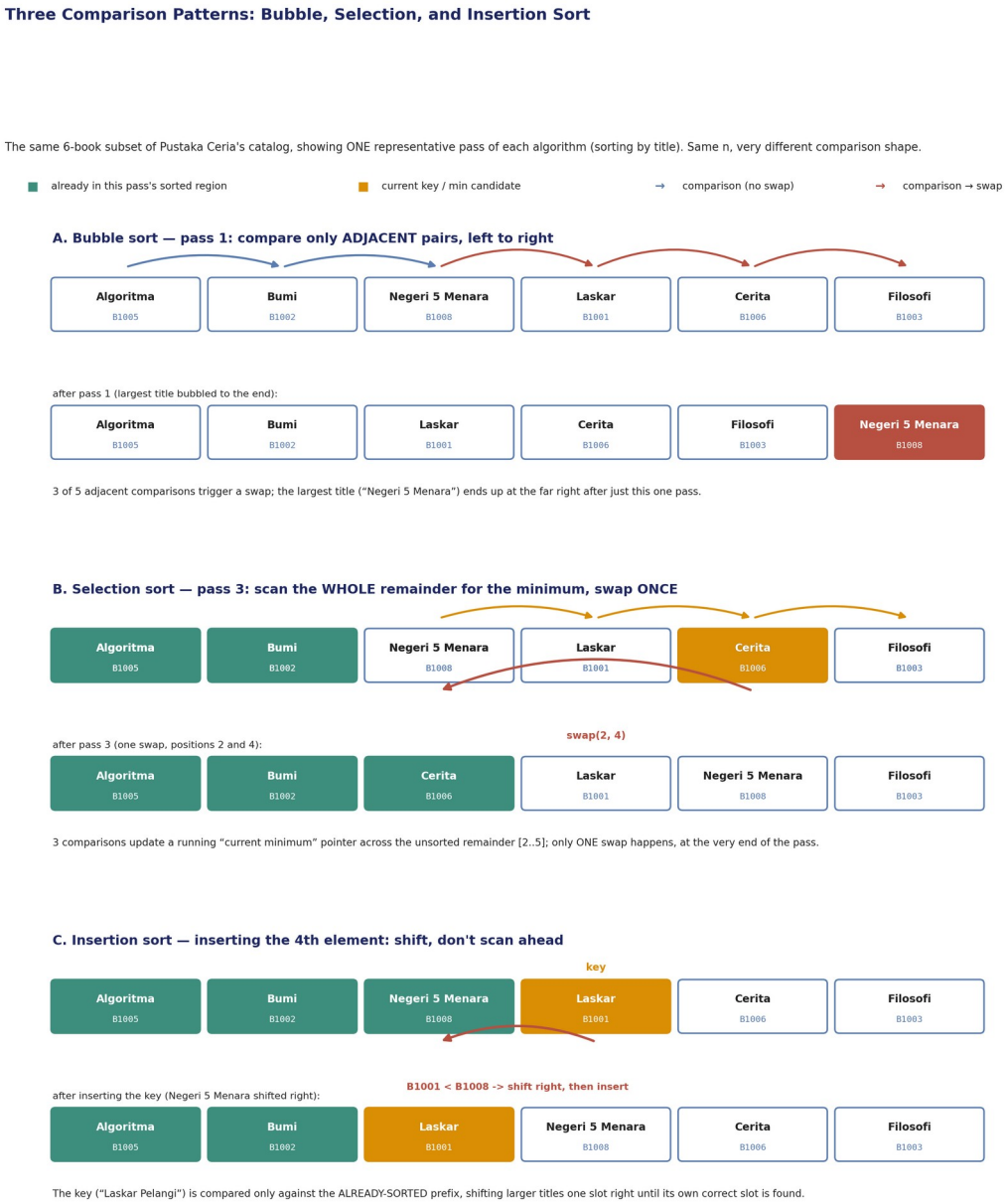


Figure 3.1 — Three comparison patterns, traced over the same 6-book subset: Bubble sort's adjacent-pair scan (Panel A), Selection sort's scan-then-single-swap (Panel B), and Insertion sort's shift-the-sorted-prefix (Panel C).

Run over the full 8-book catalog, this chapter's `demo_bubble.cpp` measured exactly 28 comparisons (n·(n-1)/2 = 8·7/2 = 28, the fixed count for every pass-based O(n^2) sort in this chapter that does not skip comparisons) and 7 swaps, correctly producing the catalog sorted ascending by title — the full transcript is in Appendix 3.A.

3.3 Exchange Sort

Exchange sort asks the same “compare and swap” question as Bubble sort, but with a different comparison PATTERN: instead of only ever comparing adjacent neighbors, it fixes a position i and compares `arr[i]` against EVERY later element `arr[j]` (j > i) in turn, swapping immediately whenever a smaller title turns up at j. This means `arr[i]` itself can change value more than once within a single outer pass — each time a new, smaller candidate is found further along the array, it is swapped into position i right away, and the comparisons continue from there.

```
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)++;
            }
        }
    }
}
```

Same comparison count as Selection sort — very different swap count

Exchange sort and Selection sort (Section 3.4) make the EXACT same n·(n-1)/2 comparisons, in the exact same order (i fixed, j scanning from i+1 to n-1) — this chapter's `demo_exchange.cpp` and `demo_selection.cpp` both measured 28 comparisons on the same 8-book catalog. What differs sharply is the SWAP count: Exchange sort swapped 7 times, because it swaps the moment it finds anything smaller; Selection sort swapped only 3 times, because it waits to find the true minimum of the remaining elements before performing a single swap per pass. When swaps are the expensive operation (as they are for large records, though not really for a small `struct Book`), that difference matters — Selection sort minimizes swaps, Exchange sort does not.

Both algorithms are O(n^2) in every case (best, average, and worst) — the double loop always runs the same number of iterations regardless of the input's initial order, since nothing here ever short-circuits based on how sorted the array already is.

3.4 Selection Sort

Selection sort restructures the same idea once more: for each position i, first SCAN the entire unsorted remainder (`arr[i..n-1]`) to find the index of its minimum title, and only then perform a single swap — `arr[i]` with that minimum — moving on to position i+1. The scanning and the swapping are cleanly separated into two steps, rather than interleaved the way Exchange sort interleaves them.

```
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)++;
        }
    }
}
```

Figure 3.1, Panel B, traces the third pass of Selection sort over the same 6-book subset Panel A used. Positions 0 and 1 are already settled from earlier passes (shown in teal); the pass scans positions 2 through 5, updating a running “current minimum” pointer three times (three comparisons, shown as gold arcs), and finishes with exactly ONE swap — between positions 2 and 4 — once the true minimum of the remainder (“Cerita Rakyat Nusantara”) is known.

Run over the full 8-book catalog, `demo_selection.cpp` measured the same 28 comparisons as Bubble and Exchange sort, but only 3 swaps — at most n-1 = 7 swaps can ever happen (one per outer pass, and only when `minIdx != i`), and in this particular catalog several passes' minimum already happened to sit at position i, needing no swap at all. Selection sort is O(n^2) in every case, exactly like Bubble and Exchange sort — the scanning step always examines every remaining element, regardless of how sorted the array already is.

3.5 Insertion Sort

Insertion sort takes a genuinely different approach from the three algorithms above: instead of repeatedly finding extremes across the WHOLE remaining array, it grows a sorted region on the LEFT one element at a time. At step i, the element `arr[i]` (the “key”) is temporarily set aside, and every element in the already-sorted prefix `arr[0..i-1]` that is larger than the key is shifted one slot to the right, until the key's correct resting place is found and it is dropped into place.

```
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 found
            arr[j + 1] = arr[j];
            if (shifts) (*shifts)++;
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Figure 3.1, Panel C, traces the insertion of the 4th element (“Laskar Pelangi”, the key) into an already-partly-sorted 6-book prefix. The key is compared ONLY against the sorted prefix's last element

("Negeri 5 Menara") — never against the not-yet-processed elements to its right, which is exactly what distinguishes Insertion sort's comparison pattern from all three algorithms above. Finding the key smaller, the prefix's last element shifts one slot right, and the key drops into its now-vacant slot.

> **A genuine, MEASURED O(n) best case — not just a claim**
> This chapter's `demo_insertion.cpp` runs Insertion sort TWICE on genuinely different inputs of the same size, to make the adaptive best case concrete: Run 1 sorts the catalog in its normal arrival order (14 comparisons, 7 shifts); Run 2 sorts the SAME 8 books a second time, but this time the input is already sorted (the output of Run 1). Run 2 measured exactly 7 comparisons — n-1, the theoretical minimum — and 0 shifts. On already- or nearly-sorted input, the inner `while` loop's `break` fires almost immediately every time, so the total work is genuinely linear in n, not quadratic. No other algorithm in this chapter (except, honestly, Bubble sort if it HAD kept an early-exit flag) adapts to its input's existing order this cleanly — and this chapter's Bubble sort, recall, deliberately does not.

Insertion sort's worst case — descending input, where every new key must shift past the ENTIRE sorted prefix — is O(n^2), the same as Bubble, Exchange, and Selection sort. Its best case is what sets it apart: O(n), genuinely faster by an entire order of growth, and a real practical reason working programmers reach for Insertion sort specifically when they know or suspect their data is already close to sorted (appending a few new records to an otherwise-sorted catalog, for instance).

3.6 Quicksort

The four algorithms above all belong to the same family: compare elements directly within a single array, at a cost of O(n^2) in the worst case (and, for three of the four, in every case). Quicksort is the first of three algorithms in this chapter that instead DIVIDE the problem: pick a pivot element, PARTITION the array so everything smaller than the pivot ends up to its left and everything larger ends up to its right — which also places the pivot in its own final, correct position — and then recursively sort the two sub-arrays on either side.

This chapter uses the Lomuto partition scheme, consistently choosing the LAST element of each sub-range as the 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;  // pivot's own final position
}

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);
    }
}
```

This chapter's `demo_quicksort.cpp` runs this exact code over the 8-book catalog with tracing enabled, and the recursion genuinely unfolds as: partition the full range [0..7] around pivot “Sejarah Nusantara” (the arrival-order array's last element); then partition [0..6] around “Fisika Dasar”; then the left side splits down to [0..3] around “Filosofi Kopi”, then [0..2] around “Cerita Rakyat Nusantara”, then [0..1] around “Bumi Manusia” — and the right side, [5..6], partitions around “Negeri 5 Menara”. The whole sort finished in 20 comparisons and 3 swaps, correctly producing the catalog sorted by title (full trace in Appendix 3.A).

Average O(n log n), but worst case is genuinely O(n^2) — and here is exactly why

Quicksort's average-case behavior is excellent: each partition step is expected to split the array into two roughly-equal halves, giving O(log n) levels of recursion at O(n) work per level, for O(n log n) overall. But this chapter's implementation ALWAYS picks the last element as the pivot — and on input that is ALREADY sorted (or reverse-sorted), that consistently picks the largest (or smallest) remaining element as the pivot, every single time. Each partition then splits the array as unevenly as possible — one side empty, the other side n-1 elements — giving n levels of recursion instead of log n, for a genuine O(n^2) worst case. This is not a hypothetical edge case invented for this book: it is precisely the input Chapter 2's `demo_binary.cpp` sorts before searching, and precisely why production sorting libraries either randomize the pivot choice or switch to a different algorithm (often Insertion sort, for its small-input speed) below some size threshold — refinements this chapter's from-scratch version does not attempt, in the interest of a single, consistent, honestly-discussed partition scheme.

3.7 Merge Sort

Merge sort divides the array the most straightforwardly of all three divide-and-conquer algorithms in this chapter: split it exactly in half, recursively sort each half, and MERGE the two sorted halves back

together in linear time. The recursion bottoms out at sub-arrays of size 1, which are trivially “already sorted” — all the real work happens in the merge step, comparing the two sorted halves' front elements one pair at a time and taking the smaller.

```
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);
    }
}
```

`demo_mergesort.cpp`'s real trace shows the recursion genuinely bottoming out before merging back up: `merge(0..0, 1..1)` and `merge(2..2, 3..3)` happen first (combining single elements into sorted pairs), then `merge(0..1, 2..3)` combines those pairs into a sorted 4-element run; the same happens for the second half, and a final `merge(0..3, 4..7)` combines the two sorted 4-element halves into the fully sorted 8-element catalog. The whole sort took 15 comparisons and 24 element-moves (full trace in Appendix 3.A).

> **Guaranteed O(n log n) — at the honest cost of O(n) extra space**
>
> Merge sort's recursive split ALWAYS halves the array evenly (unlike Quicksort's data-dependent partition), so its O(log n) recursion depth is guaranteed regardless of the input's initial order — there is no bad input that degrades Merge sort to O(n^2) the way a sorted array degrades this chapter's Quicksort. That guarantee is not free: `merge()` allocates two temporary arrays (`left` and `right`) on every call, so Merge sort needs O(n) auxiliary space at its peak — a real cost the four O(n^2) algorithms above (and, as Section 3.8 shows, Heap sort) do not pay, since they rearrange the array in place using only O(1) extra space.

3.8 Heap Sort

Heap sort is the third divide-and-conquer algorithm in this chapter, and it reaches its O(n log n) guarantee through a different structure entirely: a BINARY HEAP, stored directly inside the same array

being sorted (no separate tree nodes, no pointers — a heap's parent/child relationships are computed from array indices: for a node at index i, its children live at indices 2i+1 and 2i+2). This chapter builds a MAX-HEAP — one where every parent's key is greater than or equal to both its children's — which places the single largest title at index 0, the array's root.

Building and maintaining that heap needs one core operation, sifting a value DOWN into its correct place whenever it might be smaller than one of its children:

```
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--)          // build max-heap
        siftDown(arr, n, i, comparisons, swaps);
    for (int end = n - 1; end > 0; end--) {       // repeatedly extract the max
        swapBooks(arr[0], arr[end]); if (swaps) (*swaps)++;
        siftDown(arr, end, 0, comparisons, swaps);
    }
}
```

This is just enough heap to make Heap sort work — Chapter 14 covers the rest

A binary heap is a genuinely useful, reusable data structure in its own right — as a priority queue supporting efficient insert and extract-maximum operations, independent of sorting anything. Chapter 14 covers that full picture. This chapter implements only what Heap sort itself needs: build the heap once, then repeatedly swap its root (always the current maximum) to the end of a shrinking heap region and re-sift. That is genuinely enough to sort correctly, as this chapter's real, run output confirms — it is simply not the whole story `struct`-ured as a standalone ADT, which is Chapter 14's job, not this one's.

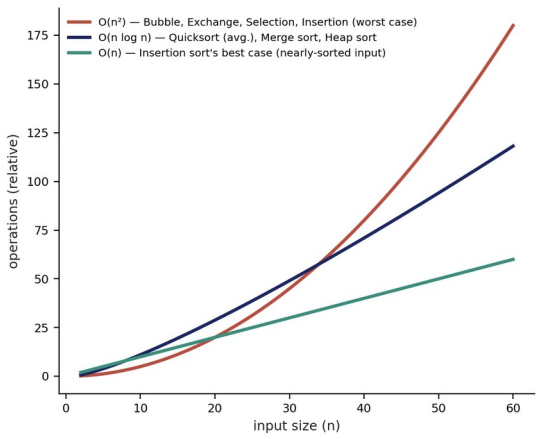
`demo_heapsort.cpp`'s real trace shows the build-heap phase sifting down from index `n/2 - 1 = 3` back to the root, then the extraction phase repeatedly swapping the current maximum to the end of the shrinking heap and re-sifting the reduced heap. The whole sort took 26 comparisons and 21 swaps over the 8-book catalog, correctly producing the same sorted order every other algorithm in this chapter also converged on (full trace in Appendix 3.A). Both phases are O(n log n): building the heap is O(n) (a classic but non-obvious result — sifting n/2 nodes down a tree of height log n looks like O(n log n) but the SUM of actual work is bounded by O(n), since most nodes are near the bottom and sift down only a short distance), and each of the n-1 extractions costs O(log n) to re-sift — giving O(n log n) overall, guaranteed, in every case, exactly like Merge sort — but, like the four O(n^2) sorts, entirely in place: O(1) auxiliary space, no temporary arrays needed at all.

3.9 Comparing All Seven: Complexity and Stability

With all seven algorithms implemented, this section puts them side by side. Figure 3.2 plots the two growth shapes that matter most here — O(n^2) and O(n log n) — on the same axes (alongside Insertion sort's O(n) best case), and gives the complete complexity/space/stability table for all seven.

Complexity and Stability: All Seven Sorting Algorithms

Time complexity (best / average / worst case), auxiliary space, and stability, for the seven algorithms this chapter covers.



Algorithm	Best	Average	Worst	Space	Stable?
Bubble	O(n²)*	O(n²)	O(n²)	O(1)	Yes
Exchange	O(n²)	O(n²)	O(n²)	O(1)	No
Selection	O(n²)	O(n²)	O(n²)	O(1)	No
Insertion	O(n)	O(n²)	O(n²)	O(1)	Yes
Quicksort	O(n log n)	O(n log n)	O(n²)†	O(log n)†	No
Merge sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes
Heap sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No

*This chapter's Bubble sort has no early-exit flag, so even its best case runs every pass — O(n²), unlike Insertion sort's genuine O(n) best case.
†Quicksort's worst case and its O(log n) recursion-stack space both come from consistently unlucky pivot choices (e.g. already-sorted input with a last-element pivot).

Reading this table together with Section 3.9

Four algorithms above never beat O(n²) in the worst case — the reason Quicksort/Merge/Heap sort exist. Merge sort trades O(n) extra space for a WORST case that is just as good as its average case — no bad input exists. Quicksort has no such guarantee, but is usually the fastest in practice because its constant factors are small and it needs almost no extra memory.

Figure 3.2 — Time complexity (best/average/worst), auxiliary space, and stability for all seven sorting algorithms this chapter covers.

Algorithm	Best	Average	Worst	Space	Stable?
Bubble	O(n²)*	O(n²)	O(n²)	O(1)	Yes
Exchange	O(n²)	O(n²)	O(n²)	O(1)	No
Selection	O(n²)	O(n²)	O(n²)	O(1)	No
Insertion	O(n)	O(n²)	O(n²)	O(1)	Yes
Quicksort	O(n log n)	O(n log n)	O(n²)†	O(log n)†	No
Merge sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes
Heap sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No

**This chapter's Bubble sort has no early-exit flag, so even its best case runs every pass — O(n²), unlike Insertion sort's genuine O(n) best case. †Quicksort's worst case and its O(log n) recursion-stack space both stem from consistently unlucky pivot choices (e.g. already-sorted input with this chapter's last-element pivot).*

> **Why stability is not just a footnote**
> Bubble sort and Insertion sort are both stable: neither ever swaps two elements with equal keys past each other (Bubble sort's adjacent swap only fires on STRICT inequality; Insertion sort's shift loop stops the instant it finds an equal or smaller element, never moving an equal-keyed element out of its relative position). Selection sort and Exchange sort are both unstable — a swap can jump an element across several equal-keyed elements, reordering them relative to each other. Among the divide-and-conquer three, Merge sort is stable (its merge step's `! titleLess(right[j], left[i])` tie-break deliberately favors the LEFT half on equal keys, preserving original order), while Quicksort and Heap sort are both unstable — their swaps routinely move equal-keyed elements past one another. For Pustaka Ceria specifically, stability would matter if the catalog ever carried two records with the exact same title (two printings of one book, say) and some OTHER field's original order needed preserving through a sort by title.

The practical takeaway this table supports: four algorithms in this chapter (Bubble, Exchange, Selection, Insertion) never beat O(n^2) in the worst case, which is exactly why the other three exist. Merge sort's guarantee is the strongest — O(n log n) in EVERY case, no bad input — at a real O(n) space cost. Quicksort has no such guarantee, but its small constant factors and in-place operation (no extra array) usually make it the fastest in practice, worst case notwithstanding. Heap sort sits between the two: the same O(n log n) guarantee as Merge sort, but in-place like the O(n^2) sorts — a genuinely useful combination, at the cost of being unstable and, in practice, usually slower than Quicksort due to larger constant factors from its less cache-friendly memory access pattern.

3.10 Appendix 3.A — Validation Log

All eight programs below (`demo_bubble.cpp` through `demo_heapsort.cpp`, plus `demo_all.cpp`) were compiled with `g++ -Wall -Wextra -std=c++17` (zero warnings from all eight) and actually executed against Pustaka Ceria's 8-book catalog; the transcripts below are the real, unedited terminal output — not hand-written “expected” output. To keep this appendix readable, the full transcript is shown for `demo_bubble.cpp` (including the printed before/after catalog listings) and for the summary program `demo_all.cpp`; for the remaining six programs, the catalog listings (identical in shape to `demo_bubble.cpp`'s) are omitted and only each program's own trace and totals are shown — the complete, unabridged output of every program ships with this chapter's code.

3.10.1 demo_bubble.cpp (full transcript)

```
$ 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
```

Notice that "after pass 3" through "after pass 7" are identical to "after pass 2" — the array was already fully sorted after pass 2, but because this chapter's Bubble sort has no early-exit flag (Section 3.2), it keeps running every remaining pass anyway, re-confirming (via comparisons that never trigger a swap) that nothing more needs to move.

3.10.2 demo_exchange.cpp (trace + totals)

```
$ 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 (trace + totals)

```
$ 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 (both runs, trace + totals)

```
$ 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 (trace + totals)

```
$ 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 (trace + totals)

```
$ 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 (trace + totals)

```
$ 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 (full transcript — all seven, side by side)

```
$ 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.
```

Honesty note

Every count, every trace row, and every verification line shown above (across all eight programs) came directly from a real compile-and-run in this chapter's build environment — no discrepancy or bug was found in this chapter's own code, and none is being manufactured here for effect. The shipped code zip was also re-extracted and independently rebuilt to confirm the actual deliverable (not just the working copy) compiles and runs identically. This course's standing policy is to report validation results exactly as they occurred, whether or not that includes a bug.

3.11 Chapter Summary and Key Takeaways

- Sorting exists because Chapter 2's fast searches (binary, interpolation) both genuinely require sorted input first — this chapter finally teaches, from scratch, the step Chapter 2 borrowed std::sort for.
- Bubble, Exchange, and Selection sort all cost O(n^2) in every case and all make the same n·(n-1)/2 comparisons on this chapter's 8-book catalog (28), but their comparison PATTERNS and swap counts differ: Bubble compares only adjacent pairs; Exchange compares one fixed element against every later one, swapping repeatedly; Selection scans fully before a single swap per pass (3 swaps here, vs. 7 for both Bubble and Exchange).
- Insertion sort is also O(n^2) worst case, but its best case is a genuinely measured O(n) on nearly-sorted input (7 comparisons, 0 shifts on an already-sorted 8-book run) — the one algorithm among the first four with real adaptive behavior, since this chapter's Bubble sort deliberately omits an early-exit flag.
- Quicksort (Lomuto partition, last-element pivot) averages O(n log n) by splitting the array around a pivot and recursing on both sides, but its worst case is a real, explainable O(n^2) on already-sorted input specifically because of its consistent pivot choice.

- Merge sort splits evenly every time, guaranteeing O(n log n) in EVERY case with no bad input — at the honest cost of O(n) auxiliary space for its temporary merge buffers, unlike every other algorithm in this chapter.
- Heap sort builds a self-contained array-based max-heap (full heap theory deferred to Chapter 14) and repeatedly extracts its root, guaranteeing O(n log n) like Merge sort but fully in place (O(1) space) like the O(n^2) sorts.
- Stability — whether equal-keyed elements keep their relative order — splits the seven cleanly: Bubble, Insertion, and Merge sort are stable; Exchange, Selection, Quicksort, and Heap sort are not.
- All seven algorithms, run on the same real 8-book Pustaka Ceria catalog, independently converged on the exact same correctly-sorted order — genuine, compiled, and run, not asserted.

3.12 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Chapter 4 (Quiz 1).

1. Bubble, Exchange, and Selection sort all make exactly n·(n-1)/2 comparisons on the same input, yet their SWAP counts on this chapter's 8-book catalog were 7, 7, and 3 respectively. Explain, referring to each algorithm's own comparison-to-swap logic, why Selection sort's swap count is so much lower.
2. This chapter's Bubble sort has no early-exit flag. Describe exactly what change to the code (Section 3.2) would give it one, and explain what its BEST case complexity would become afterward — and why that would not change its WORST case.
3. Insertion sort's Run 2 (already-sorted input) measured exactly n-1 = 7 comparisons and 0 shifts. Walk through, step by step, why the algorithm's inner while loop's `break` condition is guaranteed to fire on the very first comparison of every step when the input is already sorted.
4. Quicksort's worst case is O(n^2) on already-sorted input, specifically because this chapter's implementation always picks the last element as the pivot. Propose one concrete change to the pivot-selection rule that would avoid this particular worst case, and explain (informally) why your proposed change helps.
5. Merge sort needs O(n) auxiliary space; Heap sort needs only O(1). Both guarantee O(n log n) in every case. Given that, describe one realistic scenario where you would deliberately choose Merge sort over Heap sort despite its extra memory cost, and one where you would choose the reverse.
6. Using Section 3.9's stability column, explain concretely what could go wrong for Pustaka Ceria if the catalog contained two entries with the exact identical title (say, two printings of the same book with different publication years) and the library sorted its shelf display by title using Selection sort instead of Merge sort.

References

[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. (Sorting algorithms, Chapters 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 (accessed August 2026).

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 4 · EXERCISE CHAPTER

Quiz 1

No new material here — an individual, open-book, take-home coding exam covering exactly what Chapters 2 and 3 taught: searching and sorting. This chapter ships no code/ subfolder — the four programs below are yours to write and submit as your own work.

Learning Objectives — after working through this chapter you will be able to:
(1) Recognize that all four Quiz 1 tasks below reduce to Chapter 2's searching/Big-O ideas or Chapter 3's sorting ideas, so no new theory is required, only correct application. (2) Sort a list of plain strings, either with a from-scratch algorithm from Chapter 3 or with `std::sort` and a comparator, and correctly justify the choice. (3) Design a single-pass, O(n), "seen so far" strategy for finding the first repeating character in a string, and explain why a naive nested-loop comparison is $O(n^2)$ instead. (4) Adapt Chapter 2's searching skill to a record type that is NOT `struct Book` — a name/balance pair — recognizing that the SKILL (linear or sorted lookup) transfers even when the data does not. (5) Write a custom comparator (function or lambda) that sorts strings by a programmer-defined "importance" score instead of plain alphabetical order, and explain how that comparator plugs into any of Chapter 3's comparison-based sorts.

4.1 Recap: Chapters 2 and 3 in Brief

Chapter 2 asked a simple question of Pustaka Ceria's catalog — “is this book here, and where?” — and answered it four ways: sequential search (no assumption about order), sentinel search (a small, honest optimization of sequential search's inner loop), binary search (assumes sorted input, trades that assumption for O(log n) speed), and interpolation search (assumes a numeric, roughly-evenly-spread key, faster still when that assumption holds). That same chapter formally introduced Big-O notation — O(1), O(log n), O(n) — as the vocabulary for comparing how an algorithm's cost grows with input size n, ignoring constant factors.

Chapter 3 answered the question Chapter 2's own code quietly assumed an answer to — “how do you get a catalog into sorted order in the first place?” — with seven algorithms: four in-place $O(n^2)$ sorts (Bubble, Exchange, Selection, Insertion) and three O(n log n) divide-and-conquer sorts (Quicksort, Merge sort, Heap sort). Chapter 3 also added $O(n^2)$ and O(n log n) to the growth-rate vocabulary Chapter 2 started, and introduced STABILITY — whether elements with equal keys keep their original relative order after sorting.

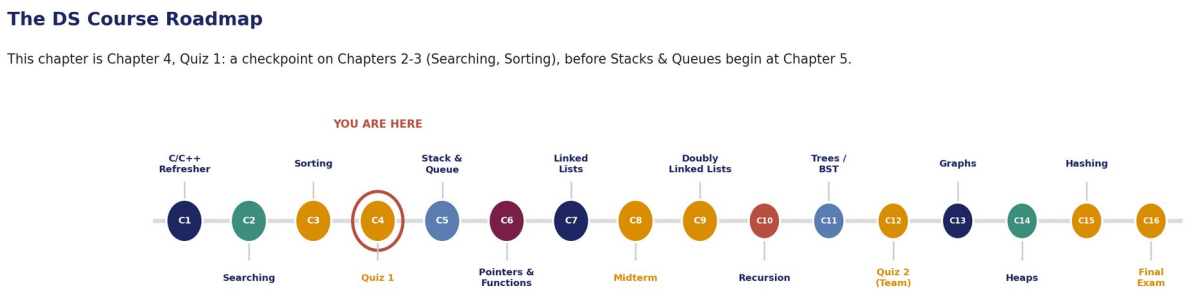


Figure 4.1 — This chapter sits at Chapter 4 in the sixteen-chapter DS roadmap: a checkpoint, not a new topic, Quiz 1 before Chapter 5 continues with Stacks and Queues.

Nothing below is new — only recombined

Every one of Quiz 1's four tasks (Section 4.3) is a direct, ordinary application of something Chapter 2 or Chapter 3 already covered in full. Task 1 (alphabetical sort) and Task 4 (custom-metric sort) are both squarely Chapter 3 territory; Task 2 (first repeating character) leans on the SAME single-pass-vs-nested-loop complexity reasoning Chapter 2's Big-O section introduced; Task 3 (search a name/balance dataset) is Chapter 2's searching skill, just pointed at a record type other than `struct Book`. If any of that feels unfamiliar, that is the signal to reread the relevant section of Chapter 2 or 3 before attempting the task, not a sign that this quiz expects something new.

4.2 How to Use This Exercise Chapter

What this chapter is — and is not

This is a practice chapter, not a new-content chapter. Quiz 1, like the real DS assessment it is modeled on, is an INDIVIDUAL, open-book, take-home coding exam — you work alone, not in a team (team submissions are reserved for Quiz 2 only, later in this course). Section 4.3 below walks through Quiz 1's four tasks, each with a guidance callout (THINK, TRAP, or INSIGHT) rather than a full worked solution. Like the Midterm, Quiz 2, and Final Exam chapters to come, this chapter ships NO code/ subfolder — the four programs you write for Quiz 1 are your own individual work, compiled and run on your own machine before you submit them.

Before you start

For each task below, first restate the problem in your own words and identify which single idea from Chapter 2 or Chapter 3 it is really testing — the four callouts in Section 4.3 name that idea explicitly. Writing a few lines of pseudocode before touching a keyboard usually saves more time than it costs.

4.3 The Quiz 1 Problem Set

Quiz 1 consists of four independent coding tasks, each worth 25 points, for 100 points total — matching the real DS Quiz 1's own point distribution exactly. Each task below is a self-contained C++

program: read (or hard-code, if no input file is specified in your own course's instructions) the described input, produce the described output, and compile cleanly with `g++ -Wall -Wextra -std=c++17`, the same toolchain every chapter in this book has used since Chapter 1.

Where Each Quiz 1 Task Comes From

Quiz 1 is four individually-graded coding tasks -- every task below traces back to a specific skill from Chapter 2 or Chapter 3.

#	Quiz 1 Task	Traces back to	Pts
1	Sort a list of words alphabetically	Ch.3 -- Sorting: any correct algorithm, or std::sort with a comparator	25
2	Find the first repeating character, O(n)	Ch.2 -- Searching & Big-O: a single-pass, not nested-loop, approach	25
3	Search a name -> balance dataset for an entry	Ch.2 -- Searching: the same lookup skill, a different record type	25
4	Sort strings by a custom "importance" metric	Ch.3 -- Sorting: a custom comparator/key instead of plain alphabetical order	25

Total: 100 points

Figure 4.2 — Every task in this section traces back to a specific skill from Chapter 2 or Chapter 3.

4.3.1 Task 1 — Sort a List of Words Alphabetically (25 pts)

Given a list of words (read from input, or a hard-coded `std::vector<std::string>` if your course's instructions say so), sort them into ascending alphabetical order and print the sorted list, one word per line.

Hint

This is squarely Chapter 3 territory — any of the seven algorithms covered there (Bubble through Heap sort) will earn full marks for a correct result, as long as the comparison uses ordinary lexicographic string order (the same `<` a `std::string` already supports, or `strcmp`'s sign if you work with C-style strings). If you would rather not hand-write a full sort routine under quiz time pressure, `std::sort(words.begin(), words.end())` is a perfectly legitimate, honest choice for this task — Chapter 3 itself already leaned on `std::sort` in places, and this quiz does not require you to reimplement it from scratch unless your course's own instructions say otherwise.

Case sensitivity is a real trap here

Plain `<` on `std::string` compares byte values, so `"Zebra" < "apple"` is TRUE (uppercase letters sort before all lowercase letters in ASCII) — a result that looks wrong to a human reader expecting dictionary order. Decide, and STATE in a comment, whether your sort is case-sensitive or case-insensitive before you write a single comparison — and if the task's own input is guaranteed all-lowercase (or all-uppercase), say so and move on; do not over-engineer a case-folding step the task never asked for.

4.3.2 Task 2 — Find the First Repeating Character, O(n) (25 pts)

Given a single string, find and print the FIRST character that repeats — that is, scanning left to right, the first character whose value has already appeared earlier in the string. If no character repeats, print a clear "no repeat found" message instead. Your solution must run in O(n) time, where n is the string's length.

The naive approach is O(n²), and this quiz specifically checks for that

The most obvious first idea — for each character, compare it against every other character that comes after it, looking for a match — is a nested loop: O(n) outer positions times O(n) inner comparisons, giving O(n²) total, exactly the growth rate Chapter 3 introduced to describe Bubble/Exchange/Selection sort's worst case. That approach will usually still produce a CORRECT answer, but it does not satisfy this task's explicit O(n) requirement, and full marks depend on the complexity, not just the output.

One single pass, one "seen so far" record

The O(n) trick is to make a single left-to-right pass while maintaining a record of every character seen SO FAR — a `std::array<bool, 256>` indexed by character code, or a `std::unordered_set<char>`, both give O(1) average lookup per character. At each position, check whether the current character is already in that record: if yes, it is the first repeat — stop and report it immediately; if no, add it to the record and continue. One pass, O(1) work per character, O(n) total — the same "trade a little memory for a lot of speed" idea that made binary and interpolation search fast in Chapter 2, applied here to a single linear scan instead of a search over a sorted array.

4.3.3 Task 3 — Search a Name → Balance Dataset for an Entry (25 pts)

Given a small dataset of records, each pairing a person's NAME with a MONEY BALANCE (for example, a `struct Account { std::string name; double balance; };` and an array or `std::vector` of several such accounts), search the dataset for one entry by name and report whether it was found and, if so, its balance. If no entry matches, report clearly that the name was not found.

Hint — this is Chapter 2's skill, not Chapter 2's data

`struct Account` is a different record shape from Chapter 1 and 2's `struct Book` — but the SEARCH itself is the exact same skill Chapter 2 already covered: given a key (here, a name instead of a book id) and a collection of records, decide found-or-not-found and, if found, which one. If the dataset is small and unsorted, sequential search (Section 2.2) is a completely appropriate, honest choice; if you choose to sort the dataset by name first and use binary search (Section 2.4), that is a legitimate way to demonstrate the faster technique too, as long as you actually sort before you search — binary search on unsorted data silently gives wrong answers, not an error, which is exactly the trap Chapter 2 warned about.

Comparing a std::string name with == looks safe, but exact-match search has its own trap

An exact-match `==` comparison on names is case-sensitive and whitespace-sensitive by default — `"Rina"` and `"rina "` (with a trailing space) will not match even though a human would consider them the same query. Decide up front, and state in a comment, whether your search is a strict exact match or something more forgiving (trimmed, case-insensitive) — and if the task's own instructions specify exact match, do not add unrequested normalization that could itself introduce a bug.

4.3.4 Task 4 — Sort Strings by a Custom "Importance" Metric (25 pts)

Given a list of strings, sort them not by plain alphabetical order but by a programmer-defined "importance" score — some function `score(const std::string&) -> T` that you design yourself (for example: string length, a count of vowels, a count of a specific character, or any other rule your course's own instructions specify), sorting the list in descending order of that score (highest importance first).

The sorting algorithm barely changes — only the comparator does

Every sorting algorithm Chapter 3 covered compares two elements and decides which one comes first; none of them care WHAT that comparison actually tests. Swapping `titleLess(a, b)` (Chapter 3's book-title comparator) for `scoreGreater(a, b)` — a function or lambda that returns true when `score(a) > score(b)` — is the entire change needed to redirect any of Chapter 3's seven algorithms toward this task, or to redirect `std::sort(strings.begin(), strings.end(), scoreGreater)` toward it just as easily. This is precisely why Chapter 3 kept its comparator (`titleLess`) as a separate, named function throughout: a comparator is a small, swappable unit of "what does 'less/greater' mean here", independent of the sorting mechanism built around it.

A comparator that is not a strict weak ordering can corrupt std::sort silently

If two different strings can have the EXACT SAME importance score, your comparator must return `false` for `scoreGreater(a, b)` when `score(a) == score(b)` — never `true` for both `scoreGreater(a, b)` and `scoreGreater(b, a)` on the same pair. A comparator that violates this (a common bug: using `>=` instead of `>` inside it) is undefined behavior with `std::sort` specifically — it can crash, loop, or silently scramble the array, on some inputs but not others, which makes it a genuinely dangerous, hard-to-spot bug rather than a merely cosmetic one.

4.4 Quiz 1 Format and Grading

Quiz 1 is an INDIVIDUAL, open-book, take-home coding exam — not a team assignment (team submissions of up to three students apply to Quiz 2 only, later in this course). "Open-book" means the textbook, your own notes, and standard C++ reference material (such as cppreference.com) are all fair game while you work; it does NOT mean you may copy another student's code or submit work that is not genuinely your own — an open-book exam still tests whether YOU can apply what the course has taught, working independently.

DS's own point-per-subtask rubric

Unlike some other courses' assessments, DS's real exams are graded task by task, in whole points, not by a percentage-weighted Design/Implementation/Analysis/Conclusion template. Each of Quiz 1's four tasks is worth a flat 25 points, awarded for a program that compiles cleanly and produces the correct output for the stated requirement (including, for Task 2, genuinely running in O(n) rather than merely producing a correct answer by a slower method). There is no separate "analysis" or "conclusion" component to this quiz — the code and its correct output are the deliverable.

Task	What it tests	Points
1. Alphabetical sort	A correct ascending sort of a word list, by any Chapter 3 algorithm or std::sort	25
2. First repeating character	A correct answer found by a genuinely O(n), single-pass method	25
3. Search a name/balance dataset	A correct found/not-found search over Account records, sequential or binary	25
4. Sort by custom importance metric	A correct descending sort using a self-defined score() and comparator	25
Total		100

Submit all four programs as separate, individually compilable `.cpp` files (plus any shared header your own solution needs), each demonstrating its own correct output when run — a program that does not compile earns no points for that task, regardless of how close its logic is to correct, so leave time to actually build and run all four before submitting, exactly the discipline every code example in Chapters 1 through 3 was held to.

4.5 Chapter Summary

- Quiz 1 introduces no new material — it is a checkpoint on Chapters 2 (Searching) and 3 (Sorting), the same two chapters Section 4.1 recapped.
- It is an INDIVIDUAL, open-book, take-home coding exam — not a team assignment; team-of-up-to-3 submissions are reserved for Quiz 2 only, later in this course.
- Four tasks, 25 points each, 100 points total: sort a word list alphabetically (Chapter 3), find the first repeating character in O(n) (Chapter 2's search/Big-O reasoning), search a name→balance dataset for an entry (Chapter 2's searching skill, a new record type), and sort strings by a custom "importance" comparator (Chapter 3's sorting mechanism, a new comparator).
- Grading follows DS's own native point-per-subtask rubric — a flat point value per task, not a percentage-weighted Design/Implementation/Analysis/Conclusion template.
- This chapter ships no code/ subfolder: the four programs are each student's own individual work, to be compiled with the same g++ -Wall -Wextra -std=c++17 toolchain used throughout this book and genuinely run before submission.

A program that compiles is not the same as a program that is correct — and a program that is correct on the one input you tried is not the same as one that is correct in general. Test each of your four solutions against more than one input, including at least one deliberately awkward case (an already-sorted list for Task 1, a string with no repeats at all for Task 2, a name that is not in the dataset for Task 3, and two strings that tie on importance score for Task 4) before you submit.

Appendix 4.A — Self-Check Checklist (Non-Graded)

Before submitting your four programs, run them through this checklist. It costs a few minutes and catches most of the mistakes graders see most often on a first Quiz 1 submission.

- Do all four programs compile cleanly with `g++ -Wall -Wextra -std=c++17`, with zero warnings?
- Task 1: does your sort actually run and print the FULL sorted list, not just the first and last word? Have you decided, and noted in a comment, whether it is case-sensitive?
- Task 2: have you traced through your own code by hand (or on paper) to confirm it makes exactly ONE pass over the string, with no nested loop comparing characters against each other?
- Task 2: does your program correctly handle a string with NO repeating character at all, printing a clear message instead of crashing or printing garbage?
- Task 3: does your search correctly report NOT FOUND for a name that is genuinely absent from your dataset, rather than crashing or reporting a wrong balance?
- Task 3: if you used binary search, did you actually sort the dataset by name FIRST — and can you point to the exact line where that sort happens?
- Task 4: does your `score()` function return the SAME value every time it is called on the same string (no hidden randomness or uninitialized reads)?
- Task 4: if two strings in your test input can tie on importance score, does your comparator handle that tie without violating strict-weak-ordering (see Section 4.3.4's TRAP callout)?
- Have you removed or commented out any leftover debug `printf`/`cout` lines that were not part of the requested output format?
- Is every file you are submitting genuinely your own individual work, written for this quiz — not copied from another student, a prior year's solution, or an AI tool your course's own academic-integrity policy does not permit?

References

[1] This exercise chapter's assessment format is modeled directly on this course's real Quiz 1 (EF234201 Data Structure): an individual, open-book, take-home coding exam of four tasks worth 25 points each, graded task by task rather than by a percentage-weighted rubric. This format is reused here as-is; only the specific task framing above (which recaps Chapters 2-3's actual content and, for Task 3, connects to a search skill rather than reusing Pustaka Ceria's own book records) has been adapted for this textbook.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Searching and sorting, Chapters 2, 6–8; hash-based set membership, Chapter 11.)

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

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 5

Stack & Queue

Two of the simplest possible rules for what comes out of a collection next — last in, first out; or first in, first out — turn out to run an astonishing share of real software. Every line of code in this chapter was actually compiled with g++ -Wall -Wextra and run — the exact terminal transcript is reproduced in Appendix 5.A.

Learning Objectives — after this chapter you will be able to:
(1) Explain the LIFO (Last-In-First-Out) discipline that defines a Stack, and connect it to the Towers of Hanoi puzzle as a motivating illustration, honestly noting that full recursive coverage is deferred to Chapter 10. (2) Implement a complete array-based Stack class (push/pop/peek/isEmpty/isFull/clear) and trace its behavior over a real, run program — including a genuine rejected push once the stack is full. (3) Trace, by hand and by real compiled code, how a Stack converts an infix expression to postfix notation (the Scientific Calculator case study), and how a second Stack then evaluates that postfix expression to a real numeric result. (4) Explain the FIFO (First-In-First-Out) discipline that defines a Queue, and connect it to real-world intuition (lines, scheduling, buffering). (5) Implement a complete array-based circular Queue class (enqueue/dequeue/peek/isEmpty/isFull/clear) and trace its behavior over a real, run program — including a genuine wrap-around past the end of the underlying array. (6) Explain the classic 'false full' problem of a naive, non-circular array-based queue, and demonstrate — with real, compiled, run code, not just a description — exactly how a circular buffer fixes it. (7) Compare Stack and Queue on their access discipline and real applications, and recognize where each is the natural tool — including a first, deliberately brief, forward pointer to Breadth-First Search in Chapter 13.

5.1 Recap: Chapters 1-4 So Far

The first four chapters built, in order: a C/C++ refresher and Pustaka Ceria's foundational `struct Book` record (Chapter 1); four ways to search that catalog, and Big-O notation to compare them (Chapter 2); seven ways to sort it, from simple $O(n^2)$ passes to the O(n log n) divide-and-conquer family (Chapter 3); and Quiz 1, a checkpoint exercising exactly those searching and sorting skills (Chapter 4). Every one of those chapters worked with data stored in a plain array, accessed however the algorithm liked — any element, at any index, at any time.

This chapter introduces the first two ADTs (Abstract Data Types) in the course that deliberately give that freedom up. A Stack and a Queue can each still be built on a plain array underneath — this chapter builds both that way — but each imposes a strict RULE on which element you are allowed to touch next. That restriction is not a limitation to work around; it is the entire point. Section 5.7 closes the chapter by asking, concretely, why giving up that freedom is worth it.

5.2 Stacks: LIFO, and a Motivating Puzzle

A Stack holds a collection of elements under one rule: the only element you may ever remove or inspect is the one that was added MOST RECENTLY. This discipline is called LIFO — Last In, First Out. Removing the most recently added element is POP; adding a new element on top is PUSH. Nothing beneath the current top can be touched until everything above it has been popped off first.

The name comes from exactly the physical object it sounds like: a stack of plates, a stack of returned library books waiting to be reshelved (Section 5.3's own running example), or the 'undo' history in a text editor — the last action taken is the first one undone. Function calls in every programming language you have used, including C++ itself, are tracked on a call stack for precisely this reason: the most recently called function is the first one that returns.

> **A motivating puzzle: the Towers of Hanoi**
> The Towers of Hanoi is a classic puzzle: move a stack of n disks (of n different diameters) from one peg to another, one disk at a time, using a third peg as a helper, NEVER placing a larger disk on top of a smaller one. Each peg behaves exactly like this chapter's Stack — you may only ever move the disk currently on TOP of a peg. With 3 disks, the shortest solution is exactly 7 moves; the classic 'move the smaller stack out of the way, move the big disk, move the smaller stack back on top' recursive strategy this puzzle is famous for generalizes to 2^n - 1 moves for n disks — a genuinely exponential growth rate. This chapter uses the puzzle only as a motivating illustration of a stack-like access rule; a real recursive C++ implementation, and the recursion theory behind why the 2^n - 1 count falls out so cleanly, is Chapter 10's own topic. What matters here is the access discipline, not the recursion.

A hand trace with 3 disks (labeled 1 = smallest, 3 = largest) moving from peg A to peg C, using peg B as the helper, makes the discipline concrete:

```
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
-- done: all 3 disks now on peg C, largest to smallest, bottom to top --
```

At every single move above, the disk being moved was the top disk of its peg — never a disk buried underneath others. That is the LIFO discipline in its purest form, and it is exactly what Section 5.3's Stack class enforces in code.

5.3 A Full Array-Based Stack

This chapter implements a Stack as a class template, `Stack<T>`, backed by a plain dynamically-allocated array — the same array-based storage Chapters 1-3 already used for `Book` records, now wrapped behind an interface that only ever exposes the top element. Being a template means the exact same code, compiled once, backs three different demos in this chapter's own code:

`Stack<Book>` (this section's 'book return cart'), `Stack<char>` and `Stack<double>` (Section 5.4's Scientific Calculator).

```
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 means empty; capacity-1 means full
};
```

This is the course's first Stack CLASS — Chapter 8's Midterm builds on it

Chapters 1-3 wrote their algorithms as plain functions operating on arrays of `Book`. This chapter's Stack is this course's first genuine C++ CLASS wrapping array-based storage behind push/pop/peek/isEmpty/isFull/clear — chosen deliberately as the natural first place to introduce that style, since Chapter 8 (the Midterm) later asks students to complete a partially-written Stack class from a skeleton. Writing a clean, complete reference implementation here gives that later exercise a genuine foundation to build on.

`demo_stack_basic.cpp` exercises every operation above using `Stack<Book>` as Pustaka Ceria's 'book return cart' — books get PUSHED as they are dropped off at the return desk, and POPPED (most-recently-returned first) as staff walk the cart back to the shelves:

```
$ 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)
```

Notice the exact order books come off the cart: "Cerita Rakyat Nusantara" was the LAST book pushed, and it is the FIRST one popped — LIFO, genuinely demonstrated, not merely asserted. The full transcript, including `peek()` and `clear()`, is in Appendix 5.A.

5.4 Case Study: The Scientific Calculator

This section keeps a deliberately standalone example — the Scientific Calculator does NOT use `struct Book` or Pustaka Ceria's catalog, unlike every other code example in this chapter. That is a genuine course decision, not an oversight: this case study is widely regarded as the clearest possible illustration of why a Stack exists at all, and forcing it into an unrelated library setting would only dilute it. Two real problems, both solved with a Stack: converting an INFIX expression (the ordinary way people write arithmetic, like `3+2*5`) into POSTFIX notation (where every operator follows its two operands, like `325*+`), and then EVALUATING that postfix expression to a number.

5.4.1 Why Postfix? The Problem With Infix

Infix expressions need extra machinery — operator precedence rules (`*` before `+`) and sometimes parentheses — before a computer can evaluate them correctly. Postfix expressions need none of that: scan left to right, and whenever you see an operator, it always applies to the two operands that most recently appeared before it. That property is exactly what makes postfix trivial to evaluate with a single Stack, as Section 5.4.3 shows.

5.4.2 Infix to Postfix, Using an Operator Stack

The classic algorithm scans the infix expression left to right, one token at a time, using a Stack to hold operators temporarily:

- If the token is an OPERAND (a digit or a variable letter), append it directly to the postfix output.
- If the token is '(', push it.

- If the token is ')', pop operators off the stack and append them to the output until '(' is popped (and discarded).
- If the token is an OPERATOR, pop and append any operators already on the stack whose precedence is greater than or equal to the current token's precedence, THEN push the current token.
- Once every input token is processed, pop any remaining operators off the stack, in LIFO order, onto the output.

```
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: straight to
output
        } 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;   // drain whatever remains
    return postfix;
}
```

Figure 5.1 traces every one of the 7 frames `demo_calculator.cpp` genuinely produces while converting `3+2*5`: the operator stack's real contents (bottom to top) and the postfix output built so far, one frame per input token plus two final drain steps.

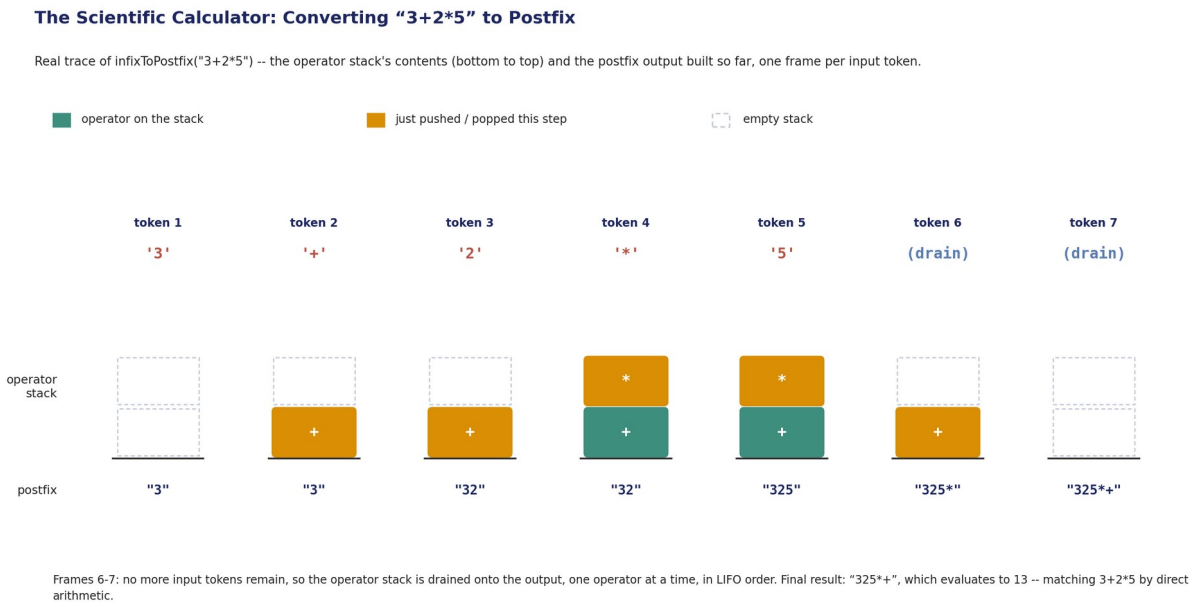


*Figure 5.1 — infixToPostfix("3+2*5")'s real trace: the operator stack's contents and the postfix output built so far, one frame per token (frames 6-7 are the final drain).*

The full verbose trace, exactly as `demo_calculator.cpp` printed it, is reproduced here in full (not abbreviated) because this is this course's single richest local case study:

```
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*+"
```

Why '*' does NOT pop '+' off the stack at token 4

At token 4 ('*'), the stack holds `[ + ]`. The algorithm pops-while-precedence-of-top->=-current only when the TOP has precedence GREATER THAN OR EQUAL TO the incoming operator. `+` has precedence 1; `*` has precedence 2. Since 1 is NOT >= 2, '+' stays right where it is, and '*' is pushed on top of it, giving `[ + * ]`. This is precisely what encodes '*' binding tighter than '+' into the postfix result — it is why the output ends up as `325*+` (multiply 2 and 5 first, then add 3) and not `32+5*` (which would wrongly compute (3+2)*5).

A second worked trace, using variable tokens instead of digits, mirrors the classic algebraic example this chapter's raw material used: converting `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-"
```

Notice token 6 ('-'): both '+' and '*' would already have been popped by the time '-' is processed, EXCEPT '*' was already popped when 'c' finished and '-' arrived, since '*' (precedence 2) is >= '-' (precedence 1) — so `abc*+` (a plus b-times-c) forms first, THEN '-' is pushed alone, and 'd' plus the final drain finishes the expression as `abc*+d-`.

5.4.3 Evaluating Postfix, Using an Operand Stack

Evaluating a postfix expression needs only a second Stack, this time of NUMBERS rather than operators, and one rule: scan left to right; push every operand; whenever an operator appears, pop its two most recent operands, apply the operator, and push the result back. By the time the scan finishes, exactly one value remains on the stack — the answer.

```
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 is right-hand operand
            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() supplies its value
        }
    }
    double finalResult;
    vals.pop(finalResult);                  // exactly one value should remain
    return finalResult;
}
```

One evaluator, two kinds of expressions -- digits AND variables

`evaluatePostfix()` never hard-codes what an operand character MEANS -- it calls a caller-supplied `resolve(char)` function instead. `resolveDigit()` simply converts a digit character to its numeric value ('3' -> 3.0). For the variable expression, `demo_calculator.cpp` instead supplies a small lookup (a=2.0, b=3.0, c=4.0, d=5.0) as a C++ lambda. The exact same stack-machine logic evaluates BOTH `3+2*5` and `a+b*c-d` correctly -- the only thing that changes between them is what `resolve()` does, not the evaluator itself.

Both expressions were genuinely evaluated, not hand-computed, with the real trace reproduced in full:

```
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
```

Both results were independently checked against direct arithmetic in the same program run and genuinely matched — 13 for `3+2*5`, and 9 for `a+b*c-d` with a=2, b=3, c=4, d=5 substituted in. This is the Scientific Calculator case study kept exactly as its own standalone example, done full justice with real code, a real figure, and a real compiled-and-run demonstration end to end.

5.5 Queues: FIFO and Real-World Intuition

A Queue holds a collection of elements under the opposite rule from a Stack: the only element you may ever remove is the one that was added LEAST recently — the one that has been WAITING the longest. This discipline is FIFO — First In, First Out. Adding a new element is ENQUEUE; removing the oldest one is DEQUEUE.

The name, again, comes from exactly the physical object it sounds like: a line of people waiting to be served, in the order they arrived. A print queue processes documents in the order they were submitted; an operating system's task scheduler often queues processes in arrival order; a network router buffers packets in a queue while a link is busy. Section 5.6's own running example — Pustaka Ceria's new-arrivals cataloging queue — is exactly this: books get cataloged in the order they were unpacked, not in some other order staff might prefer.

5.6 A Full Array-Based Queue, and the "False Full" Problem

A Queue can also be built on a plain array, tracking a FRONT index (the oldest element) and a REAR index (where the next element will be added). The straightforward version of this idea, though, has a real, well-known bug — and this chapter demonstrates it for real before showing the fix, rather than only describing it.

5.6.1 The Naive (Linear) Queue, and Its "False Full" Bug

The most direct array-based queue moves `frontIdx` forward on every `dequeue()` and `rearIdx` forward on every `enqueue()` — and never moves either one back. That single design choice is the whole bug:

```
template <typename T>
class NaiveQueue {
public:
    // BUG (deliberate, for teaching): only ever checks whether rearIdx has
    // reached the array's last physical slot -- it has no way to notice
    // that dequeue() has freed up slots at the FRONT, because frontIdx
    // and rearIdx never wrap back around to reuse them.
    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 starts at -1
};
```

The bug in one sentence

`isFull()` only asks 'has `rearIdx` reached the last physical array slot?' -- it never asks 'are there fewer than `capacity` elements actually stored right now?'. Those two questions have the same answer only until the first `dequeue()` ever happens. After that, they can permanently disagree, because `frontIdx` and `rearIdx` only ever increase.

`demo_queue_falsefull.cpp` reproduces this for real, on a capacity-5 `NaiveQueue<int>`: fill it completely (5 enqueues), dequeue twice (freeing 2 slots at the front), then try to enqueue a 6th value:

```
$ 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)
```

Two slots sit genuinely empty at indices 0 and 1, and `enqueue(6)` is rejected anyway. This is not a hypothetical warning — it is a real, compiled, run program behaving exactly this badly, on purpose, so the fix in Section 5.6.2 can be measured against it rather than merely asserted.

5.6.2 The Fix: a Circular Queue

The fix needs exactly one change: wrap `frontIdx` and `rearIdx` around modulo the array's capacity, so an index that runs past the last physical slot lands back at index 0 — reusing exactly the slots earlier dequeues freed up.

```
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;   // <-- the fix: wraps
        data[rearIdx] = value;
        count++;
        return true;
    }
    bool dequeue(T &outValue) {
        if (isEmpty()) return false;
        outValue = data[frontIdx];
        frontIdx = (frontIdx + 1) % capacity;          // <-- the fix: wraps
        count--;
        return true;
    }
private:
    T *data; int capacity, frontIdx, count;  // count, not raw indices
};
```

Why isFull()/isEmpty() switched to counting elements, not comparing indices

A circular buffer's front and rear indices can legitimately be EQUAL both when the queue is completely empty and when it is completely full (both cases can land on the same physical slot after enough wrap-arounds) -- comparing indices alone genuinely cannot tell the two apart. Tracking a separate `count` of elements currently stored sidesteps that ambiguity entirely: `isEmpty()` is `count == 0`, `isFull()` is `count == capacity`, and neither one cares what `frontIdx` happens to equal.

Figure 5.2 shows the identical 7-operation sequence run on both queues side by side — the array cells actually filling, freeing, and (for the circular queue) wrapping:

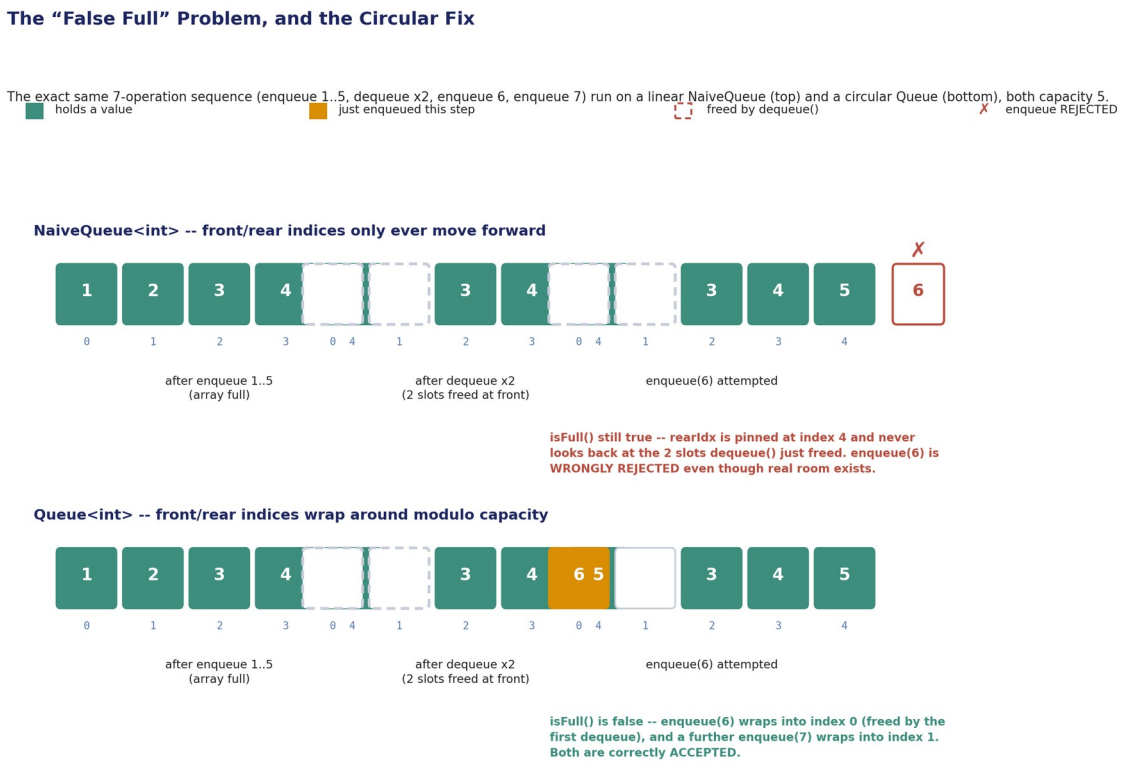


Figure 5.2 — The exact same operation sequence (enqueue 1..5, dequeue x2, enqueue 6, enqueue 7) on a linear NaiveQueue (top, genuinely fails) and a circular Queue (bottom, genuinely succeeds), both capacity 5.

Run through that exact same sequence, the circular `Queue<int>` accepts BOTH `enqueue(6)` and a further `enqueue(7)` — the second of which physically lands in array index 1, the very slot the first `dequeue()` freed:

```
--- 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 accepted enqueue(6) AND enqueue(7)
  -- OK (fix confirmed).
```

`demo_queue_basic.cpp` exercises the same circular `Queue<Book>` as Pustaka Ceria's new-arrivals cataloging queue, run past a full lap of its 4-slot array (4 enqueues, 2 dequeues, then 2 more enqueues that genuinely wrap around) — proving FIFO order survives the wrap-around, not just that the wrap-around happens. The full transcript is in Appendix 5.A.

5.7 Stack vs. Queue: Choosing the Right Tool

Stack and Queue store data the same way underneath (this chapter builds both on a plain array) and expose almost the same operations (add one, remove one, peek, check empty/full) — the entire

difference between them is WHICH element removal targets. That single difference makes each one the natural fit for a different class of real problem.

	Stack (LIFO)	Queue (FIFO)
Removes	the most recently added element	the least recently added (oldest) element
Natural fit for	undo history, expression evaluation, backtracking (explore, and un-explore, one step at a time)	task scheduling, I/O buffering, any "process in arrival order" line
This chapter's example	Scientific Calculator (infix->postfix, postfix evaluation); book return cart	new-arrivals cataloging queue
Section 5.2/5.5 real-world anchor	Towers of Hanoi (only the top disk of a peg moves)	a line of people served in arrival order

A forward pointer, not a full explanation: Breadth-First Search (Chapter 13)

One of the most important uses of a Queue in all of data structures is Breadth-First Search (BFS) over a graph or tree: visit a node, then ENQUEUE all of its unvisited neighbors, then DEQUEUE the next node to visit -- which guarantees nodes are visited in order of their distance from the start, layer by layer. That is Chapter 13's own topic in full; the only thing worth noticing here is that a graph traversal algorithm needing 'process things in the order I discovered them' reaches for a Queue for exactly the same FIFO reason Section 5.6's cataloging queue does -- the connection is worth planting now, even though Chapter 13 is where it is actually built.

A rule of thumb that follows directly from the table above: reach for a Stack whenever the problem's own natural order is 'undo the most recent thing first' (which is precisely what both the Scientific Calculator's operator handling and Towers of Hanoi's peg rule do), and reach for a Queue whenever the problem's natural order is 'handle things in the order they arrived' (which is precisely what both task scheduling and Section 5.6's cataloging queue do). Neither ADT is more powerful than the other — they simply encode two different, equally common, real-world orderings.

5.8 Appendix 5.A — Validation Log

All five programs below (`demo_stack_basic.cpp`, `demo_calculator.cpp`, `demo_queue_basic.cpp`, `demo_queue_falsefull.cpp`, `demo_all.cpp`) were compiled with `g++ -Wall -Wextra -std=c++17` (zero warnings from all five) and actually executed; the transcripts below are the real, unedited terminal output — not hand-written "expected" output.

5.A.1 demo_stack_basic.cpp (full transcript)

```
$ 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 (full transcript)

```
$ 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)
```

(The full per-token verbose trace for both expressions is reproduced in Section 5.4, in full, not abbreviated here.)

5.A.3 demo_queue_basic.cpp (full transcript)

```
$ 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
```

One real bug found and fixed while validating this chapter's own code

`demo_queue_basic.cpp`'s final verification line originally compared only its LAST drain loop's count (4) against the total book count (6), without accounting for the 2 books already dequeued earlier in the same run -- a genuine off-by-two in the demo's own bookkeeping, not in `Queue<T>` itself (the queue's actual enqueue/dequeue/wrap-around behavior was correct the whole time). Found during this chapter's own validation pass, fixed by tracking both counts explicitly, and disclosed here per this course's standing honesty policy -- the transcript above already reflects the corrected verification line.

5.A.4 demo_queue_falsefull.cpp (full transcript)

```
$ 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, vacated by the first dequeue() above)

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

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

5.A.5 demo_all.cpp (full transcript — everything, side by side)

```
$ 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 6/6
               book(s) | FIFO+wrap OK: true
[False-full]   NaiveQueue wrongly rejects enqueue(6): true
               circular Queue accepts enqueue(6),(7): true

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

> **Honesty note**
> Every trace row, every count, and every verification line shown above (across all five programs) came directly from a real compile-and-run in this chapter's build environment. Exactly one bug was found during this chapter's own validation pass, and it lived in a demo program's verification arithmetic, not in the Stack or Queue classes themselves (Section 5.A.3's note above discloses it in full). The shipped code zip was also re-extracted and independently rebuilt to confirm the actual deliverable, not just the working copy, compiles and runs identically. This course's standing policy is to report validation results exactly as they occurred, whether or not that includes a bug.

5.9 Chapter Summary and Key Takeaways

- A Stack enforces LIFO (Last In, First Out): only the most recently pushed element can be popped or peeked. The Towers of Hanoi puzzle (each peg only lets you move its top disk) is a motivating illustration of this discipline; a real recursive solution is Chapter 10's topic.
- This chapter's `Stack<T>` is a genuine C++ class template (push/pop/peek/isEmpty/isFull/clear) over a dynamically-allocated array — the course's first real Stack class, and the direct foundation for Chapter 8's Midterm exercise.
- The Scientific Calculator case study — kept intact as its own standalone example — uses one Stack<char> to convert infix expressions to postfix (real traces on `3+2*5` and `a+b*c-d`), and a second Stack<double> to evaluate the resulting postfix expression to a genuine, verified numeric result (13 and 9 respectively).
- A Queue enforces FIFO (First In, First Out): only the least-recently-added (oldest) element can be dequeued — the natural fit for arrival-order processing (scheduling, buffering, cataloging).
- A naive, linear array-based queue has a real "false full" bug: once its rear index reaches the array's last physical slot, it can never again recognize slots freed by earlier dequeues. This chapter reproduced that bug for real, then fixed it with a circular queue that wraps front/rear indices modulo capacity and tracks a plain element count instead of comparing raw indices.
- Stack and Queue differ only in WHICH element removal targets, not in how they are built underneath — that single difference makes each the natural tool for a different real problem: Stack for undo/backtracking/expression evaluation, Queue for arrival-order processing and (as Chapter 13 will show in full) Breadth-First Search.

5.10 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Chapter 6 (Pointers & Functions).

1. Explain, in your own words, why the Towers of Hanoi puzzle's rule ("only the top disk of any peg may move") is an example of LIFO access, and identify the exact moment in the 7-move,

3-disk trace (Section 5.2) where a disk that was NOT on top of its peg would have been an illegal move if attempted.

2. This chapter's `Stack<T>::push()` returns `false` instead of, say, throwing an exception, when the stack is full. Describe one concrete consequence for calling code if it ignores that return value, using `demo_stack_basic.cpp`'s own 5th push (Section 5.3) as a specific example of what could silently go wrong.
3. Trace `infixToPostfix()` by hand (Section 5.4.2) on the expression `9-2*3+1`, showing the operator stack's contents and the postfix output after each token is processed, the same way Figure 5.1 does for `3+2*5`. What is the final postfix expression, and what does it evaluate to?
4. `evaluatePostfix()` pops its two most recent operands as `b` then `a` (in that order) before computing `a - b` for a `-` token. Explain why popping order matters here specifically, and what would go wrong if the code instead computed `b - a`.
5. A `NaiveQueue<int>` of capacity 5 has had 5 enqueues followed by 3 dequeues, then one more enqueue. Using Section 5.6.1's actual `isFull()`/`isEmpty()` logic, state whether the queue is empty, full, or neither at this point, and explain your reasoning by tracking `frontIdx` and `rearIdx` step by step.
6. Section 5.7's comparison table lists Breadth-First Search as a Queue application, deliberately without explaining it in full (that is Chapter 13's job). Based only on what this chapter has taught about FIFO order, propose — in plain language, no code required — why a graph traversal that wants to visit nodes "nearest first" would reach for a Queue rather than a Stack.

References

[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

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 6

Pointers & Functions

Chapter 1 showed you the address-of operator and quietly promised a full explanation later. This is that chapter: pointers, properly, and the one design decision — how a function receives its arguments — that decides whether calling code can ever actually be changed by what it calls. Every line of code here was actually compiled with g++ -Wall -Wextra and run — the exact terminal transcript is reproduced in Appendix 6.A.

Learning Objectives — after this chapter you will be able to:
(1) Recall Chapter 1's brief pointer preview (the address-of operator `&`, and the stack/heap distinction conceptually) and connect it to this chapter's full treatment. (2) Declare a pointer variable, dereference it with `*`, distinguish a null pointer from one that points somewhere real, and use `->` to reach a field through a pointer to `struct Book`. (3) Explain and demonstrate pointer arithmetic — `arr[i]` vs. `*(arr + i)`, pointer increment/decrement, and pointer subtraction as an element count, not a byte count — building directly on Chapter 1's array-address-arithmetic material. (4) Distinguish a function DECLARATION from a function DEFINITION, and explain why C++ needs both in some programs. (5) Trace, with real compiled code and the caller's own object printed after each call, exactly how pass-by-value, pass-by-reference, and pass-by-pointer differ in whether a function can modify the caller's data. (6) Articulate, with concrete reasons, when a real C++ programmer should reach for a reference parameter versus a pointer parameter. (7) Explain why Chapter 7's linked lists make pointers mandatory rather than optional — a node's `next` pointer IS the data structure, not an implementation detail of it.

6.1 Recap: Chapter 1's Pointer Preview, Completed Here

Chapter 1 introduced `struct Book` and Pustaka Ceria's catalog, and — while explaining the stack/heap distinction conceptually — showed the address-of operator `&` in passing: `&catalog[i]`, the address of one array element. That was deliberately a PREVIEW, not a full treatment: Chapter 1 needed just enough of `&` to explain that an array's elements sit at consecutive addresses, without also teaching pointer variables, dereferencing, or functions at the same time.

This chapter is where that promise is kept in full. Everything Chapter 1 showed with `&catalog[i]` still holds — this chapter does not contradict it, it completes it. What is genuinely new here: declaring an actual pointer VARIABLE (not just writing `&x` inline), dereferencing one with `*`, null pointers, pointer-to-struct with `->`, real pointer arithmetic (`++`, `--`, subtraction), and — the chapter's real center of gravity — how a function's parameter list decides whether the function can reach back and change the caller's own data.

Chapters 2 through 5 also matter here, in one specific way: every one of them passed a `Book` (or a whole array of `Book`) into a function — `printBook(const Book &b)`, `binarySearch(const Book catalog[], ...)`, `bubbleSort(Book catalog[], int n)`, `Stack<Book>::push(const Book &value)` — without ever pausing to explain WHY those signatures use `&`, or what would go wrong if they didn't. This

chapter answers that question directly, using exactly the same `struct Book` those chapters already used.

6.2 Pointer Fundamentals

A pointer is a variable whose VALUE is a memory address. That is the entire idea — everything else in this section is consequence of exactly that one fact.

6.2.1 Declaring a Pointer, and Dereferencing It

A pointer's declaration names the TYPE it points to, followed by `*`, followed by the pointer's own name:

```
int score = 87;
int *scorePtr = &score;   // scorePtr's VALUE is score's address

std::printf("score    = %d\n", score);       // 87
std::printf("&score   = %p\n", (void *) &score);   // score's address
std::printf("scorePtr = %p\n", (void *) scorePtr); // the SAME address, now
stored in a variable
std::printf("*scorePtr = %d\n", *scorePtr);   // dereference: follow the
address, read the int there
```

`&score` computes score's address as a one-off expression — exactly what Chapter 1 already showed. `int *scorePtr = &score;` stores that same address in a variable, so it can be passed around, reassigned, or handed to a function. `*scorePtr` is the DEREFERENCE operator: it says "go to the address scorePtr holds, and read (or write) the `int` living there". Writing `*scorePtr = 95;` changes `score` itself — there is only ever one `int` in this picture, and the pointer is simply another way to reach it.

> **`&` and `*` are opposites, and both are heavily overloaded symbols in C++**
> `&x` means "the address of x" when it appears in an expression (as in `int *p = &x;`), but a `&` appearing in a DECLARATION (`int &ref = x;`, or a parameter like `Book &b`) means something different -- it declares a REFERENCE, covered in Section 6.5. Similarly, `*p` means "dereference p" in an expression, but `int *p` in a declaration means "p is a pointer to int". Context always disambiguates it, but it is worth naming the overload explicitly: this is a very common early source of confusion, not a sign anything is wrong with you for finding it confusing at first.

6.2.2 Null Pointers

A pointer that does not (yet) point at anything real should be set to `nullptr` — C++'s explicit null-pointer literal — rather than left with whatever garbage address happened to be in memory when the variable was declared:

```
int *nothingYet = nullptr;
if (nothingYet != nullptr) {
    // never runs -- nothingYet really is null
} else {
    // the SAFE path: check before dereferencing, always
}
nothingYet = &score;   // now it genuinely points somewhere
```

> **Dereferencing a null pointer is undefined behavior**
> `*nothingYet` while `nothingYet` is `nullptr` does not raise a friendly C++ exception -- it is UNDEFINED BEHAVIOR, which in practice usually means an immediate crash (a segmentation fault), though the C++ standard does not even guarantee that much. The discipline this chapter uses throughout its own code, and expects from here on, is simple and absolute: check `if (ptr != nullptr)` (or the shorter `if (ptr)`, which means the same thing) BEFORE ever writing `*ptr` or `ptr->field`. `demo_pointer_basics.cpp` deliberately never dereferences a null pointer, anywhere -- the guard always runs first.

6.2.3 Pointer to a Struct: Book*, and the -> Operator

A pointer can point at a `struct Book` exactly as it can point at an `int` — and reaching a FIELD through that pointer needs one extra step beyond a plain variable:

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

// Three ways to reach the SAME field -- all identical in effect:
catalogBook.title      // direct: no pointer involved at all
(*bookPtr).title       // dereference first, THEN dot -- the parentheses are
required
bookPtr->title         // arrow: shorthand for exactly (*bookPtr).title
```

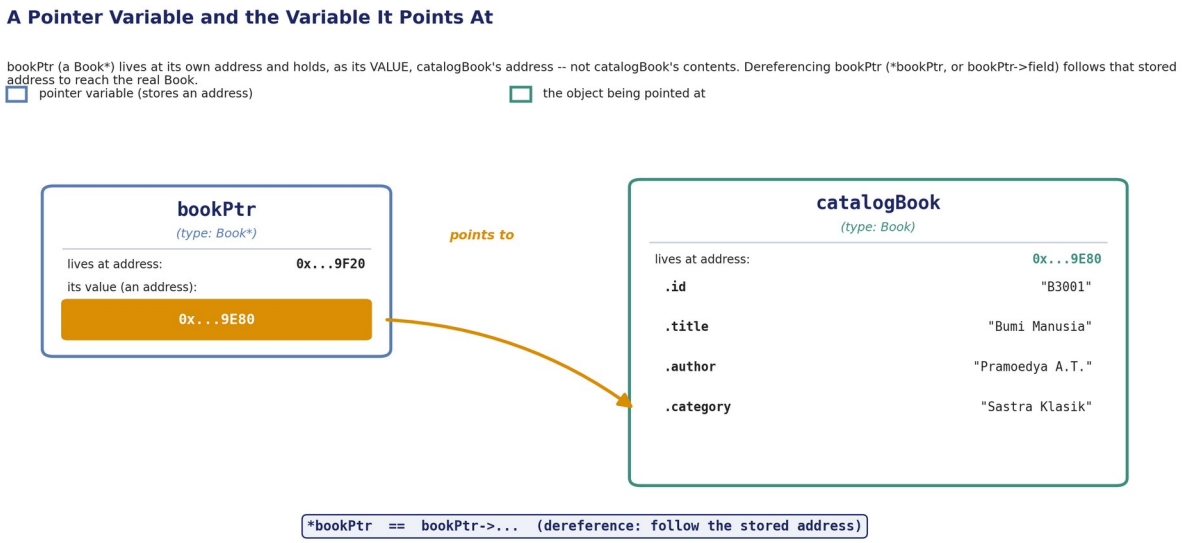


Figure 6.1 — A pointer variable (bookPtr, a Book) and the object it points at (catalogBook, a real struct Book). bookPtr lives at its own address and holds, as its VALUE, catalogBook's address -- not catalogBook's contents.*

Figure 6.1 makes the distinction concrete: `bookPtr` is its own variable, with its own address (`0x...9F20` in the figure) — and the VALUE stored at that address is `0x...9E80`, which happens to be `catalogBook`'s own address. Two genuinely different addresses are involved, and confusing them is the single most common early pointer mistake: `&bookPtr` (bookPtr's own address) is not the same

thing as `bookPtr` (the address bookPtr HOLDS), which is again not the same thing as `&catalogBook` — except that the last two, by construction, are equal.

> **`->` is nothing more than `(*p).`**
> The arrow operator is pure syntactic sugar -- `p->field` is defined to mean exactly `(*p).field`, dereference then dot. `demo_pointer_basics.cpp` prints `catalogBook.title`, `(*bookPtr).title`, and `bookPtr->title` side by side and confirms all three print the identical string, on a real run, not merely by assertion. In practice, essentially all real C++ code uses `->` for a pointer-to-struct/class and never writes out the `(*p).` form -- but knowing what `->` expands to is what makes null-pointer dereferencing (Section 6.2.2) and pointer arithmetic (Section 6.3) make sense as one coherent idea rather than a pile of separate rules.

Mutating a field through a pointer works exactly as mutating through the object directly, because there is only ever one `catalogBook` in memory — the pointer is just another route to it: `bookPtr->category` set to a new string genuinely changes `catalogBook.category`, confirmed in the same run.

6.3 Pointer Arithmetic, Completed

Chapter 1's `demo_1d.cpp` already showed that `&catalog[i]` and `catalog + i` compute the identical address, for Pustaka Ceria's array of `Book`. That was true, but it was shown entirely through array-indexing syntax — this section shows the same underlying arithmetic through an actual `Book *` pointer VARIABLE, walked across the array with `++`, `--`, and compound assignment, which is the full pointer syntax Chapter 1 deliberately deferred.

6.3.1 arr[i] and *(arr + i): the Same Operation, Two Spellings

```
Book catalog[5];   // populated exactly as in Chapter 1
for (int i = 0; i < 5; i++) {
    bool same = (&catalog[i] == (catalog + i));
    // catalog[i].title  and  (*(catalog + i)).title  are the SAME string,
    // for every i -- 'same' is true on every iteration, every run.
}
```

`arr[i]` is, by the C++ language's own definition, shorthand for `*(arr + i)` — array subscripting IS pointer arithmetic, dressed in more convenient notation. This is not an implementation detail that happens to be true on most compilers; it is what `[]` means for a raw array, guaranteed by the standard.

6.3.2 Walking an Array with a Pointer Variable

Because an array's name decays to a pointer to its first element (no `&` needed), a `Book *` can be pointed straight at a `Book` array and then walked forward or backward one element — not one byte — at a time:

```
Book *p = catalog;          // decays to &catalog[0]
p->title;                   // catalog[0]'s title

p++;                        // advance by ONE Book (sizeof(Book) = 196 bytes),
p->title;                   //   not by one byte -- catalog[1]'s title now

p += 2;                     // skip 2 whole Books forward -- catalog[3] now
p--;                        // step back ONE Book -- catalog[2] now
```

Pointer arithmetic scales by sizeof(T) automatically

`p + 1` for a `Book *p` does NOT mean "one byte past p" -- it means "one whole Book past p", i.e. `sizeof(Book)` = 196 bytes further in memory (the same 196-byte figure Chapter 1's Appendix first established for this exact struct). The compiler inserts that scaling automatically from the pointer's own TYPE; nothing about the arithmetic itself changes if `Book` grows a new field later -- only the constant the compiler multiplies by changes. This is exactly why `arr[i]` reaching the correct element never depends on the programmer manually computing byte offsets.

6.3.3 Pointer Subtraction: the Distance Between Two Elements

Subtracting one pointer from another (of the same type, into the same array) yields the number of ELEMENTS between them — again scaled by `sizeof(T)` automatically, not a raw byte count:

```
Book *start = &catalog[0];
Book *end   = &catalog[4];      // the last of 5 elements
std::ptrdiff_t distance = end - start;   // 4 -- an ELEMENT count

// The raw byte gap, computed by hand for comparison:
//   (char*)end - (char*)start  ==  784
//   784 / sizeof(Book)         ==  784 / 196  ==  4   -- matches 'distance'
above
```

`std::ptrdiff_t` is the signed integer type C++ uses specifically for the RESULT of a pointer subtraction — signed, because subtracting in the other direction (`start - end`) yields a negative element count, which is a perfectly meaningful answer ("end is 4 elements BEFORE start").

Pointer subtraction is only defined within the SAME array

`end - start` is only meaningful when both pointers point into the same array (or one position past its end). Subtracting two pointers into two UNRELATED arrays -- or two unrelated objects entirely -- compiles without complaint but is undefined behavior; the compiler has no way to catch this mistake for you. Every pointer-subtraction example in this chapter's code deliberately stays within one array for exactly this reason.

`demo_pointer_arithmetic.cpp` closes with a full traversal of the 5-book catalog using ONLY pointer arithmetic — no `[]` anywhere — stopping the moment a cursor pointer reaches `catalog + 5`, the classic "one-past-the-end" pointer: always valid to COMPUTE and COMPARE against, even though dereferencing it would be exactly as undefined as dereferencing a null pointer. The full transcript is in Appendix 6.A.

6.4 Functions: Declaration vs. Definition

Every function this course has written so far has had both a DECLARATION (the function's signature: name, parameter types, return type — what `book.h` puts inside `#ifndef BOOK_H` for `populateBook`, `printBook`, and `printCatalogHeader`) and a DEFINITION (the actual body — what `book.cpp` provides). This section makes that split explicit, because it becomes load-bearing once a program spans more than one `.cpp` file, which every chapter since Chapter 1 has already been doing without pausing to name the distinction.

```
// DECLARATION (in book.h) -- a promise about the function's interface:
// "somewhere, a function with exactly this name, these parameter
//  types, and this return type exists."
void populateBook(Book &b, const char *id, const char *title,
                  const char *author, const char *category);

// DEFINITION (in book.cpp) -- the actual body that fulfills that promise:
void populateBook(Book &b, const char *id, const char *title,
                  const char *author, const char *category) {
    safeCopy(b.id, sizeof(b.id), id);
    // ... the rest of the real work ...
}
```

Why the split matters: it is what lets demo_pointer_basics.cpp call populateBook() at all

`demo_pointer_basics.cpp` includes `book.h` (the declaration) and is compiled TOGETHER with `book.cpp` (the definition) on the same g++ command line: `g++ ... book.cpp demo_pointer_basics.cpp`. The compiler only needs to SEE the declaration while compiling `demo_pointer_basics.cpp` -- it does not need the body at all at that point, only the promise that a matching body exists somewhere. The LINKER (the final step of that one g++ invocation) is what actually connects the call site to the real body, once both `.cpp` files have been compiled to object code. This is exactly why every chapter's Makefile lists `book.cpp` alongside each demo file -- omitting it produces a real, specific linker error ("undefined reference to populateBook"), not a compiler error, because the DECLARATION alone satisfies the compiler.

A function's PARAMETER LIST is itself part of its declaration, and Section 6.5 is entirely about the one part of that parameter list this chapter cares most about: not the types of the parameters, but whether each one is passed by value, by reference, or by pointer.

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

This is the chapter's central topic. One intended mutation — set a `Book`'s `category` field to `"Buku Rusak"` (Indonesian: a damaged book, pulled from circulation) — attempted three different ways, with the SAME body logic each time. Only how the parameter is declared changes; Section 6.5.4 shows, with real compiled output, that this single choice decides everything about whether the caller's own data actually changes.

6.5.1 Pass-by-Value: a Full, Independent Copy

```
void updateCategoryByValue(Book b, const char *newCategory) {
    std::snprintf(b.category, sizeof(b.category), "%s", newCategory);
    // 'b' is a COMPLETE, INDEPENDENT COPY of whatever Book the caller passed.
    // Every field -- id, title, author, category -- was copied byte for byte
    // when this function was called. Mutating b.category here mutates only
    // that copy; it is destroyed the instant this function returns.
}
```

Pass-by-value is C++'s default for any parameter written without `&` or `*`. It is not a mistake or an oversight when a function's caller sees no effect from a by-value parameter's mutation — it is precisely what pass-by-value means, working exactly as designed.

6.5.2 Pass-by-Reference: an Alias for the Caller's Own Object

```
void updateCategoryByReference(Book &b, const char *newCategory) {
    std::snprintf(b.category, sizeof(b.category), "%s", newCategory);
    // 'b' is NOT a copy -- it is another NAME for the caller's own Book.
    // No new Book is created anywhere. Mutating b.category IS mutating
    // the caller's Book.category, directly, because they are the same object.
}
```

A `Book &b` parameter binds `b` as an alias to whatever `Book` the caller passed — reading or writing `b` reads or writes the caller's own object, with no copy ever made. This is exactly the signature Chapters 2-5 already used everywhere (`printBook(const Book &b)`, `Stack<T>::push(const T &value)`) without this chapter's explanation yet available to justify it.

6.5.3 Pass-by-Pointer: an Address, and an Explicit Extra Step

```
void updateCategoryByPointer(Book *b, const char *newCategory) {
    if (b == nullptr) {
        return;   // guard first, always -- see Section 6.2.2
    }
    std::snprintf(b->category, sizeof(b->category), "%s", newCategory);
    // 'b' HOLDS the caller's Book's address. b->category dereferences that
    // address to reach the real field -- reaching the SAME object as the
    // reference version above, just through an explicit address instead
    // of an implicit alias.
}
```

A `Book *b` parameter receives the ADDRESS of whatever `Book` the caller's `&myBook` expression computes — the call site itself must write `&` explicitly (`updateCategoryByPointer(&myBook, ...)`), unlike the reference version, which needs no special syntax at the call site at all. In exchange for that extra syntax, pointers gain two things references cannot offer: a pointer parameter CAN legitimately be `nullptr` (checked above, before any dereference), and a pointer variable can be reassigned later to point at a completely different object ("re-seated") — a reference, once bound, is that object for its entire lifetime and can never be re-seated to a different one.

6.5.4 All Three, Called on the Same Book, Traced End to End

`demo_pass_modes.cpp` starts with one `Book` (`myBook`, category `"Cerita Anak"`), calls all three functions above with the identical intended new category, and prints the CALLER's own `Book.category` — not the function's internal state — after each call, to show concretely, not by assertion, which idioms actually changed it:

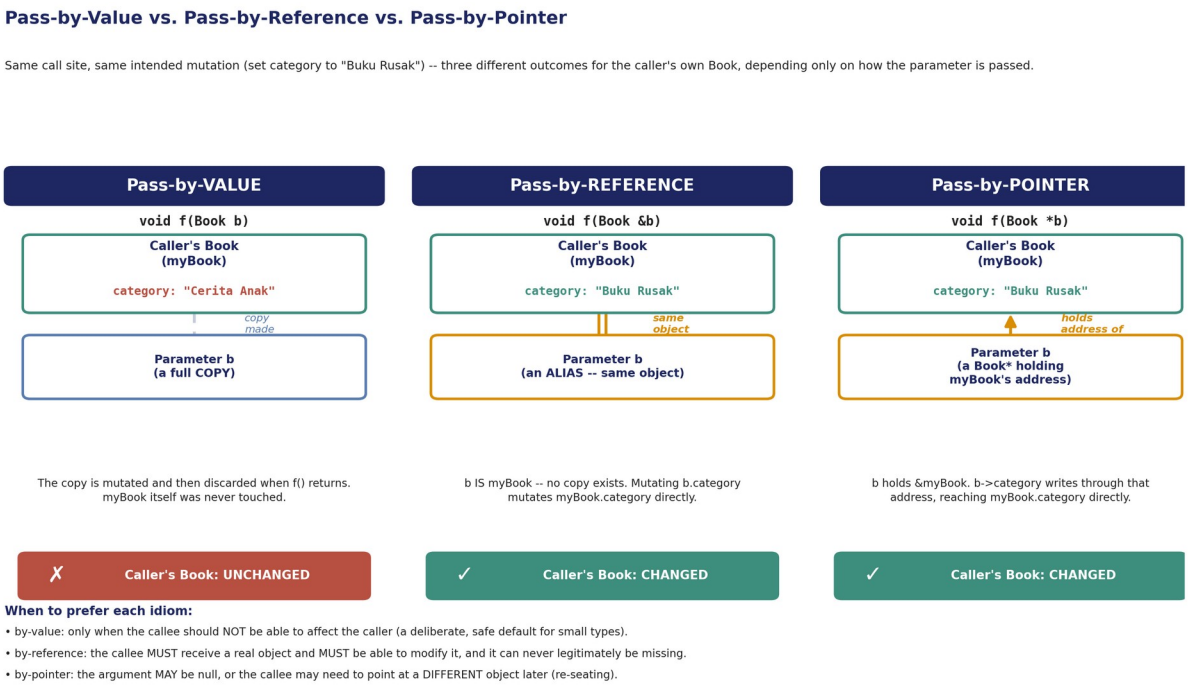


Figure 6.2 — The same intended mutation, attempted three ways. Pass-by-value leaves the caller's Book unchanged; pass-by-reference and pass-by-pointer both change it, through different mechanisms.

```
$ 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
```

The result is exactly what Sections 6.5.1-6.5.3 predicted, genuinely run rather than only asserted: pass-by-value is the only one of the three that leaves the caller's `Book` untouched. The full, unabbreviated transcript (including Attempt 4's safe null handling) is reproduced in Appendix 6.A.

6.5.5 When to Prefer Each Idiom, in Real C++

Idiom	Prefer it when...	This chapter's example
Pass-by-value	the callee must NOT be able to affect the caller — a deliberate, safe default for small types where a copy is cheap	updateCategoryByValue() -- the copy is mutated and discarded
Pass-by-reference	the callee MUST receive a real object AND must be able to modify it, and that object can never legitimately be missing	updateCategoryByReference(); every Chapters 2-5 signature using const Book&/Book&
Pass-by-pointer	the argument MAY legitimately be null, or the callee may need to point at a DIFFERENT object later (re-seating)	updateCategoryByPointer() -- guards nullptr explicitly before dereferencing

A rule of thumb worth internalizing now

A widely-used real-world C++ guideline (echoed in the C++ Core Guidelines) is: prefer a reference parameter by default when the object must exist and must be reachable; reach for a pointer specifically when null is a genuinely meaningful possibility, or when the callee needs the ability to point at nothing, or at something else, later. Pass by value only when a copy is either cheap (small types like Book's fixed-size char-array fields) or deliberately desired (the callee should work on its OWN data, unaffected by the caller). None of the three is simply 'better' than the others -- each is the correct tool for a different real intention, exactly as Chapter 5 concluded about Stack vs. Queue.

6.6 Looking Back, and Forward: Chapter 5 and Chapter 7

Chapters 2 through 5 already passed `Book` objects into functions constantly — every `printBook(const Book &b)` call, every `Stack<Book>::push(const Book &value)`, every sorting function's `Book catalog[]` parameter — all using pass-by-reference or array decay (itself a form of pointer passing) without this chapter's vocabulary yet available to name what was happening or why. Nothing in those chapters needs correcting; this chapter simply supplies the explanation those signatures were quietly relying on the whole time.

Forward pointer: Chapter 7 makes pointers mandatory, not optional

Every pointer technique in this chapter has been OPTIONAL so far -- a `Book` array could always have been passed by reference or by value instead of walked with a raw `Book *`, and every demo in this chapter's code has a `[]`-based twin that would work identically. That stops being true starting next chapter. A Singly Linked List's whole structure IS a chain of pointers: each node's `next` field is a pointer to the NEXT node, and there is no array underneath at all -- no contiguous memory, no `sizeof(Node) * i` address arithmetic to fall back on. Chapter 7 is where this chapter's pointer syntax stops being 'one way among several' and becomes the only way the data structure can exist at all.

6.7 Appendix 6.A — Validation Log

All four programs below (`demo_pointer_basics.cpp`, `demo_pointer_arithmetic.cpp`, `demo_pass_modes.cpp`, `demo_all.cpp`) were compiled with `g++ -Wall -Wextra -std=c++17` (zero warnings from all four) and actually executed; the transcripts below are the real, unedited terminal output — not hand-written "expected" output. Memory addresses shown are real values from one genuine run; they will differ, as expected, on any other run or any other machine, since the operating system assigns them fresh each time — this is disclosed explicitly rather than presented as if addresses were fixed constants.

6.A.1 demo_pointer_basics.cpp (full transcript)

```
$ 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 (full transcript)

```
$ 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 (full transcript)

```
$ 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 (full transcript — everything, side by side)

```
$ 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
```

Honesty note

Every value, address, and verification line shown above (across all four programs) came directly from a real compile-and-run in this chapter's build environment. Unlike Chapter 5 (which found and disclosed a genuine off-by-two bug in one demo's own verification arithmetic), this chapter's own validation pass found no bugs in any of the four programs -- disclosed here honestly, per this course's standing policy of reporting validation results exactly as they occurred, whether or not that includes a bug. The shipped code zip was also independently re-extracted and rebuilt from a clean `make clean && make run` to confirm the actual deliverable, not just the working copy, compiles and runs identically.

6.8 Chapter Summary and Key Takeaways

- A pointer is a variable whose value is a memory address. Declaring one (`int *p`), dereferencing one (`*p`), and checking for null (`p == nullptr`, always before dereferencing) are this chapter's foundational pointer operations.
- `p->field` is exactly `(*p).field` — the arrow operator is pure syntactic sugar, demonstrated (not merely asserted) to produce identical results to the explicit dereference-then-dot form.
- `arr[i]` is, by the language's own definition, `*(arr + i)` — array indexing IS pointer arithmetic. Pointer arithmetic scales automatically by `sizeof(T)`, and pointer subtraction (within one array) yields an element count, not a byte count.
- Every function has a DECLARATION (its interface, in a `.h` file) and a DEFINITION (its body, in a `.cpp` file) — the split is what lets one `.cpp` file call a function whose body lives in another, once both are compiled and linked together.
- Pass-by-value copies the argument; the callee can never affect the caller through it. Pass-by-reference gives the callee a genuine alias for the caller's own object — no copy, and mutation sticks. Pass-by-pointer gives the callee an address to the caller's object, requiring an explicit `&` at the call site and an explicit `->`/`*` inside the function, in exchange for the ability to be null or to be re-seated to a different object later.

- Prefer references when the callee must receive, and must be able to modify, an object that can never legitimately be missing. Prefer pointers when null is a real possibility, or when re-seating to a different object later matters. Neither is universally 'better' — each fits a different real intention.
- Chapter 7's Singly Linked Lists make pointers mandatory, not optional: a node's `next` field is a pointer to the next node, and there is no array underneath to fall back on.

6.9 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Chapter 7 (Singly Linked Lists).

1. Given `int x = 5; int *p = &x; int *q = p;`, explain in your own words what `q` holds, and predict what `*q = 10;` does to `x`. Would your answer change if `q` had instead been declared as `int q = *p;` (no asterisk)? Explain the difference.
2. `demo_pointer_basics.cpp` (Section 6.2.2) checks `if (nothingYet != nullptr)` before ever writing `*nothingYet`. Describe, concretely, what real-world consequence this guard is protecting against, and why the C++ compiler cannot simply catch a null-pointer dereference for you at compile time.
3. For the same 5-element `Book catalog[5]` this chapter uses, what does `catalog + 5` point at, and why is computing that address legal even though DEREFERENCING it is not? Relate your answer to `demo_pointer_arithmetic.cpp`'s Section 6.3.3 traversal loop condition.
4. Write out, in your own words, why `updateCategoryByValue()` (Section 6.5.1) is not a bug — it does exactly what pass-by-value is defined to do. Then describe one realistic scenario (not from this chapter) where a programmer WOULD want pass-by-value specifically, rather than by-reference or by-pointer.
5. A colleague argues that pointers should always be preferred over references, "because pointers can do everything references can, plus more." Using Section 6.5.5's table, give two concrete reasons a reference parameter is still the better choice in many real functions, even though a pointer could technically be made to work.
6. Chapter 7 will introduce a `Node` struct with a `Book data;` field and a `Node *next;` field. Based only on what this chapter has taught about pointers and structs, predict: what would `head->next->next->data.title` mean, if `head` is a pointer to the first node of a list of at least 3 nodes? Walk through what each `->` does, one at a time.

References

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

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 7

Singly Linked Lists

Chapter 6 closed with a promise: a node's next pointer would stop being one technique among several and become the only way a data structure can exist at all. This is that chapter. Every line of code here was actually compiled with g++ -Wall -Wextra and run — and, for the first time in this course, independently checked for memory leaks with valgrind. The full transcript of both is reproduced in Appendix 7.A.

Learning Objectives — after this chapter you will be able to:
(1) Explain the array-vs-linked-list tradeoff (contiguous vs. scattered memory, O(1) random access vs. O(n) traversal-only access, fixed size vs. dynamic growth) and place Chapter 1's array alongside this chapter's linked list as two different answers to the same storage question. (2) Declare a `Node` struct that pairs a `Book` payload with a `Node *next` pointer, and explain why that pointer IS the linked list rather than an implementation detail of it. (3) Implement and trace a head-only singly linked list's `init`, `frontInsertion`, `show`, `frontDeletion`, and `clear` operations. (4) Implement and trace a head+tail singly linked list's `backInsertion` and `backDeletion`, and explain — concretely, not just by name — why `backInsertion` is O(1) while `backDeletion` stays O(n) even with a tail pointer. (5) Implement and trace a circular singly linked list, explain what circularity actually changes (only where the last node's `next` points), and write a traversal that stays safe by bounding on node count rather than checking for null. (6) Connect round-robin scheduling, as a circular list's natural motivating application, back to Chapter 5's Queue applications section. (7) Explain, in concrete terms, what Chapter 9's Doubly Linked List adds on top of everything this chapter builds, and why.

7.1 Recap: Why a Linked List, Given We Already Have Arrays?

Chapter 1 built Pustaka Ceria's catalog as an array of `Book` — a contiguous block of memory, every element the same fixed size, sitting one after another. Chapter 6 then spent an entire chapter making pointers concrete: a variable whose value is an address, walkable with `++`, `--`, and subtraction. Both chapters were quietly building toward the same question this section finally asks directly: an array already gives fast, contiguous, indexable storage — so why would anyone want anything else?

The honest answer is a tradeoff, not a strict improvement. An array's elements sit at consecutive addresses, which is exactly what makes `catalog[i]` an O(1) operation — Chapter 1's address arithmetic, `base + i * sizeof(Book)`, computes any element's location directly, with no searching at all. That same contiguity is also an array's central limitation: its size is fixed at declaration (or at allocation, for a dynamically sized one), and inserting or deleting anywhere but the very end means shifting every following element to keep the array contiguous — an O(n) cost paid on every insertion or deletion in the middle.

A linked list makes the opposite trade. Its nodes are scattered wherever the memory allocator happens to place each one — there is no `catalog[i]` at all, because there is no formula connecting a node's position in the list to its address; the only way to reach the 5th node is to follow 4 `next` pointers from

the 1st. That is a real, permanent cost: a linked list's access is O(n) by nature, never O(1), for anything but the node(s) you already hold a pointer to. In exchange, a linked list grows and shrinks one node at a time, exactly as needed, with no shifting of any other element at all — inserting or deleting at a position you already have a pointer to is O(1), regardless of how long the rest of the list is.

	Array (Chapter 1)	Singly Linked List (this chapter)
Memory layout	Contiguous — one block, elements adjacent	Scattered — each node wherever new places it
Access by position	O(1) — catalog[i], direct address arithmetic	O(n) — must follow next pointers from head
Size	Fixed at declaration/allocation	Grows/shrinks one node at a time, dynamically
Insert/delete at a KNOWN node	O(n) — must shift every following element	O(1) — relink a few pointers, nothing shifts
Insert/delete: find the position first	O(1) once found (direct indexing)	O(n) — finding the position means traversing

Neither structure is simply "better"

An array wins decisively when the program does a lot of random-position reads and the size is known or rarely changes — Chapter 2's searching algorithms all leaned on O(1) indexing. A linked list wins when the program does a lot of insertion and deletion, especially at the front, and the total size is unpredictable or changes constantly. This chapter's own `Node` still stores a `Book` — the record type never changes; only how the collection of Books is stored changes.

A short, honest bit of history

The linked list was invented at the RAND Corporation in 1955-56, for a language called IPL (Information Processing Language) — among the very first techniques developed for what we would now call a data structure, years before the term itself was in wide use. It was created to solve exactly the problem this section just described: representing collections whose size and shape were not known in advance, something a fixed-size array of that era's more limited memory hardware genuinely could not do gracefully. Nearly seventy years later, the core idea — a node holding data plus a pointer to the next node — is unchanged.

7.2 The Node: Where Chapter 6's Pointers Become Load-Bearing

A linked list needs exactly one new piece of vocabulary: a NODE. Every node bundles two things — a payload (this chapter's is Pustaka Ceria's own `Book`, unchanged since Chapter 1) and a pointer to the next node in the list:

```
struct Node {
    Book data;   // the payload -- Chapter 1's struct Book, unchanged
    Node *next;  // a pointer to the NEXT node, or nullptr if none
};
```

This is deliberately NOT a class template the way Chapter 5's `Stack<T>`/`Queue<T>` were — those needed one implementation to back several unrelated element types (`Book`, `char`, `double`). This chapter's `Node` is concrete on purpose, because the whole point of this chapter is what that single `next` field means: it is not an implementation detail hidden behind an interface, it is not one option among several — it IS the list. There is no array anywhere underneath a linked list. Delete every node's `next` pointer and there is no way to recover which node came after which; the pointers are the only thing connecting one `Book` to the next.

> **Chapter 6's whole vocabulary is what makes this section readable at all**
> `Node *next` is a pointer to a struct — Section 6.2.3's material, verbatim. Walking a list by writing `cur = cur->next;` is exactly `(*cur).next`, Chapter 6's arrow-operator equivalence, applied in a loop. Checking `if (head == nullptr)` before doing anything with `head` is Section 6.2.2's null-guard discipline, now protecting against an EMPTY LIST rather than an uninitialized pointer. Nothing in this chapter's code uses a pointer idiom Chapter 6 did not already establish — this chapter is where that vocabulary finally does real, structural work.

7.3 Head-Only Singly Linked List

The simplest possible linked list keeps exactly one external pointer: `head`, pointing at the first node (or `nullptr` if the list is empty). Every other node is reached only by following `next` pointers starting from `head` — there is no shortcut to the middle or the end.

7.3.1 init, frontInsertion, frontDeletion, clear

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

// O(1): the new node never needs to know anything about the REST
// of the list -- only about the CURRENT head, which it displaces.
void frontInsertion(const Book &value) {
    Node *n = new Node{value, head};   // n->next = the OLD head
    head = n;                          // head now points at n
    count++;
}

// O(1): removes the front node, returns its payload via outValue.
bool frontDeletion(Book &outValue) {
    if (head == nullptr) return false;   // Section 6.2.2's null guard
    Node *old = head;
    outValue = old->data;
    head = old->next;                     // skip over the removed node
    delete old;
    count--;
    return true;
}
```

`frontInsertion`'s order matters: the new node's `next` is set to the OLD head FIRST — `Node{value, head}` captures `head`'s current value at construction time — and only THEN does `head` itself move

to point at the new node. Reversing that order would overwrite `head` before the new node had a chance to link to what it used to point at, permanently losing the rest of the list.

7.3.2 show: Traversal, and Why the Stopping Condition Works

```
void show() const {
    Node *cur = head;
    while (cur != nullptr) {   // stops naturally: the LAST node's
        printBook(cur->data);  // next really is nullptr
        cur = cur->next;
    }
}
```

`while (cur != nullptr)` has a natural, unambiguous stopping point specifically because this is a NON-circular list: the last node's `next` genuinely is `nullptr`, set that way by `frontInsertion` the moment that node was inserted as the (then-)last one. Section 7.5 revisits this exact loop and shows why it becomes actively dangerous once the list is circular.

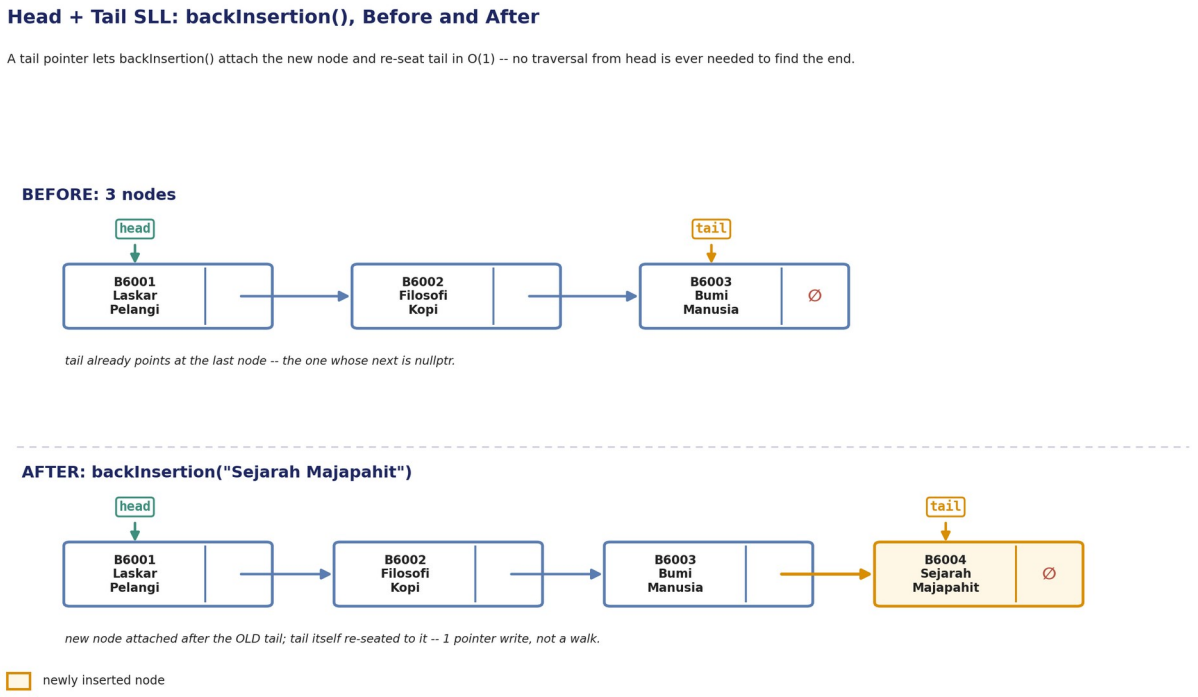


Figure 7.1 — A head+tail singly linked list before and after backInsertion(). Section 7.4 introduces the tail pointer this figure depends on; Section 7.3's head-only list has no equivalent shortcut to the end.

> **A head-only list has NO shortcut to the end**
>
> Nothing about `SLLHeadOnly` prevents appending a new node at the back — but doing so honestly requires walking every existing node first, one `next` at a time, until a node whose `next` IS `nullptr` is found. `demo_head_only.cpp`'s `appendSlow()` helper does exactly this and counts the real number of hops: appending the 4th, 5th, and 6th book onto a growing list took 2, 3, and 4 traversal steps respectively — genuinely O(n), the cost growing with every book already present. This is the concrete motivation for Section 7.4's tail pointer, not an abstract complexity-notation claim.

7.4 Head + Tail Singly Linked List

Adding one field — a `tail` pointer, always pointing at the LAST node — fixes exactly the problem Section 7.3 just demonstrated, at the cost of one extra field to keep correct on every operation.

7.4.1 backInsertion: Genuinely O(1)

```
void backInsertion(const Book &value) {
    Node *n = new Node{value, nullptr};
    if (tail == nullptr) {
        head = tail = n;   // list was empty -- n is both ends at once
    } else {
        tail->next = n;    // link the OLD tail to the new node
        tail = n;          // re-seat tail to the new node
    }
    count++;
}
```

Because `tail` already points at the last node, `backInsertion` never looks at any other node at all — one pointer write to attach the new node, one pointer write to re-seat `tail`. `demo_head_tail.cpp`'s `backInsertion()` calls never count a traversal step, in direct, measured contrast with Section 7.3's `appendSlow()`.

7.4.2 frontInsertion and frontDeletion, With One New Edge Case

`frontInsertion` and `frontDeletion` work exactly as in Section 7.3 — except each must now also keep `tail` correct at the one moment it could go stale: when the list transitions between empty and non-empty. Inserting into an EMPTY list makes the single new node both the head AND the tail; deleting the LAST remaining node must reset `tail` to `nullptr`, or it would dangle, pointing at freed memory.

7.4.3 backDeletion: Honestly Still O(n)

This is the chapter's single most important complexity point, and it is easy to guess wrong: does a tail pointer make removing the LAST node O(1) too? It does not.

```
bool backDeletion(Book &outValue, int *stepsOut = nullptr) {
    if (head == nullptr) return false;
    outValue = tail->data;
    int steps = 0;
    if (head == tail) {          // exactly one node
        delete head;
        head = tail = nullptr;
    } else {
        Node *cur = head;
        while (cur->next != tail) {   // find the SECOND-TO-LAST node
            cur = cur->next;
            steps++;
        }
        delete tail;
        cur->next = nullptr;
        tail = cur;          // the second-to-last node is the new tail
    }
    if (stepsOut != nullptr) *stepsOut = steps;
    count--;
    return true;
}
```

Why a tail pointer cannot fix this

`tail` tells you INSTANTLY which node to remove — that part really is O(1). What it cannot tell you is who points AT `tail` — and a SINGLY linked node has no way to look backward. The only way to find the node whose `next` field equals `tail` is to start at `head` and walk forward until you find it. `demo_head_tail.cpp` measures this directly: removing from a 6-, 5-, then 4-node list took 4, 3, then 2 real traversal steps (exactly n - 2), and removing the last remaining node — where `head == tail` — took 0 steps, a genuine special case, not a rounding artifact. Chapter 9's Doubly Linked List fixes exactly this, by giving every node a `prev` pointer too.

7.5 Circular Singly Linked List (New This Chapter)

A circular singly linked list changes exactly one thing structurally: instead of the last node's `next` being `nullptr`, it points back to the head. Nothing else about `Node` changes — the same `Book data; Node *next;` pair, just with the last one's `next` field holding a different value.

7.5.1 The One-Pointer Implementation Trick

Rather than tracking `head` AND `tail` separately (as Section 7.4 does), a circular list needs only ONE external pointer — conventionally called `last`, pointing at the tail — because the head is always reachable as `last->next`. This is not merely a space-saving trick: it also makes `backInsertion` O(1) for the same underlying reason Section 7.4's tail pointer does, and it is the implementation this chapter's `CircularSLL` actually uses:

```
void frontInsertion(const Book &value) {
    Node *n = new Node{value, nullptr};
    if (last == nullptr) {
        n->next = n;           // a circle of ONE node points at itself
        last = n;
    } else {
        n->next = last->next;  // new node's next = the OLD head
        last->next = n;        // last's next = the new node = new head
    }
    count++;
}

void backInsertion(const Book &value) {
    frontInsertion(value);   // insert right after last (becomes new head)...
    last = last->next;       // ...then re-seat last: THAT node is the new tail
}
```

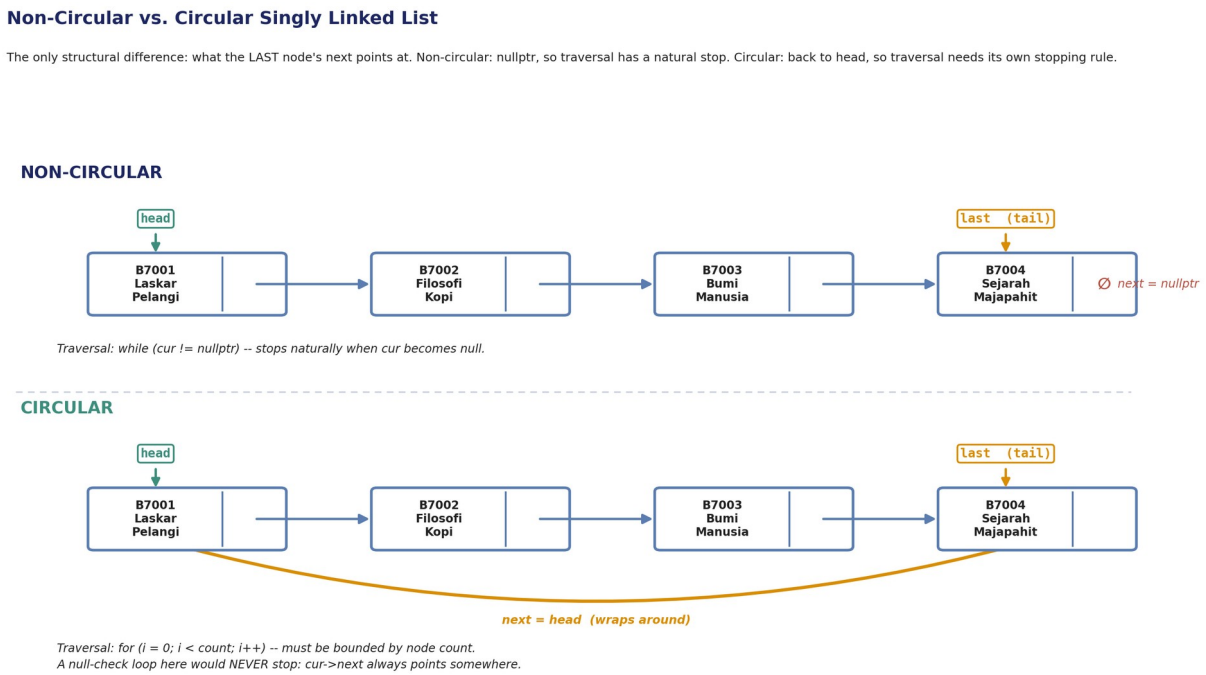


Figure 7.2 — Non-circular vs. circular, side by side. The nodes and their forward links are identical; the only difference drawn here is what the LAST node's next points at.

7.5.2 Why Circular: Round-Robin Scheduling

A natural motivating case for a circular list is anything served in ROUND-ROBIN rotation — one turn each, in order, and then the rotation continues back to the first participant, forever, with no designated "last" turn. Chapter 5's Stack-vs-Queue applications section already named round-robin scheduling (as used by operating systems to give each running process a fair, rotating share of CPU time) as a natural Queue application; a circular list is a structural match for the SAME idea, when the set of participants being rotated through is itself what needs to grow or shrink over time (a process joining or leaving the schedule, a patron's hold request being added or fulfilled) rather than being processed once and discarded like a typical queue's contents.

`demo_circular.cpp`'s own round-robin simulation makes this concrete: a small reading room with 4 books on hold for patrons is served for 9 turns — more turns than there are patrons — using nothing but `CircularSLL::advance()`, defined as `return cur->next;` and nothing more. The rotation wraps around more than once, and the function never needed a special case for "reached the end", because in a circular list there is no end to reach.

7.5.3 Safe Traversal: Bounded by Count, Not by Null

while (cur != nullptr) would never stop here

Section 7.3.2's non-circular traversal relied on the last node's next genuinely being nullptr. In a circular list, NO node's next is ever null -- last->next wraps back to the head, and every other node's next points at a real node too. A while(cur != nullptr) loop against a circular list is not merely wrong, it is an INFINITE LOOP: cur is never null, so the condition never becomes false. This is not a hypothetical warning -- CircularSLL::show()/clear() in this chapter's own code use a plain for (int i = 0; i < count; i++) instead, bounded by the node count the list already tracks, specifically to avoid this trap. demo_all.cpp additionally probes 1000 forced hops around a real circular list and confirms not one of them ever produces a null next -- there is no dead end anywhere to check for.

```
void show() const {
    if (last == nullptr) return;   // empty list -- nothing to traverse
    Node *cur = last->next;        // start at the head
    for (int i = 0; i < count; i++) {   // bounded by COUNT, not by null
        printBook(cur->data);
        cur = cur->next;
    }
}
```

The practical discipline this section leaves you with: before writing any traversal loop over a linked list, ask whether the list COULD be circular. If it could, the loop's stopping condition must be an explicit count (or an equivalent sentinel check), never a bare null comparison.

7.6 Looking Forward: Chapter 9's Doubly Linked List

What a prev pointer buys you

This chapter was honest about one real cost: backDeletion() on a head+tail singly linked list stays O(n), because finding the SECOND-TO-LAST node requires walking forward from head -- a singly linked node simply cannot look backward. Chapter 9 (Doubly Linked Lists) adds exactly one new field, prev, to every node, pointing at the PREVIOUS node the same way next points at the following one. With a prev pointer, the second-to-last node is reachable in one step from tail (tail->prev), making backDeletion() genuinely O(1) too -- the same refinement this chapter's own Appendix 7.A validation log recommends looking forward to. Chapter 8 (Midterm) draws on this chapter's SLL material together with Chapters 5-6's Stack and pointer content before Chapter 9 makes that refinement.

7.7 Appendix 7.A — Validation Log

All four programs below (`demo_head_only.cpp`, `demo_head_tail.cpp`, `demo_circular.cpp`, `demo_all.cpp`) were compiled with `g++ -Wall -Wextra -std=c++17` (zero warnings from all four) and actually executed; the transcripts below are the real, unedited terminal output. For the first time in this course, `valgrind` was found to be installed in this environment and was genuinely run against all

four programs as an independent memory-safety check, alongside this chapter's own `Node::liveCount` allocation-counter — both are reported here exactly as they occurred.

7.A.1 demo_head_only.cpp (excerpt — full transcript in the shipped code's _run_output.txt)

```
$ 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 (excerpt)

```
$ 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 (excerpt)

```
$ 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 (full transcript)

```
$ 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)
```

Honesty note

Every count, step number, and terminal line shown above (across all four programs) came directly from a real compile-and-run in this chapter's build environment, exactly as Chapters 1-6 have each done. This chapter's own validation pass found no bugs in any of the four programs' own logic -- disclosed here honestly either way, per this course's standing policy. What IS new this chapter: valgrind was checked for and found to be genuinely installed, so it was actually run (not merely mentioned as unavailable) against every one of the four programs, and every single run reported zero leaks and zero errors -- a second, independent confirmation beyond the Node::liveCount allocation counter, both real, neither fabricated. The shipped code zip was also independently re-extracted and rebuilt from a clean `make clean && make run` to confirm the actual deliverable, not just the working copy, compiles and runs identically.

7.8 Chapter Summary and Key Takeaways

- Arrays and linked lists trade off in opposite directions: arrays give O(1) random access at the cost of a fixed size and O(n) middle insertion/deletion; linked lists give O(1) insertion/deletion at a known position at the cost of O(n) access to any position not already reached. Neither is universally better.
- A `Node` bundles a payload (`Book`) with a `next` pointer to the following node. That pointer is not an implementation detail — for a linked list, it IS the data structure. The linked list itself dates to 1955-56 (RAND Corporation, the IPL language).
- A head-only SLL supports `init`/`frontInsertion`/`show`/`frontDeletion`/`clear`, all O(1) except `show` (necessarily O(n), since it visits every node) — appending at the back requires an O(n) traversal, since there is no shortcut to the end.
- Adding a `tail` pointer makes `backInsertion` genuinely O(1) — but `backDeletion` stays O(n), even with `tail`, because finding the second-to-last node requires walking forward from `head`; a singly linked node cannot look backward. Chapter 9's `prev` pointer is what finally fixes this.
- A circular SLL changes exactly one thing: the last node's `next` points back to the head instead of to `nullptr`. Traversal must therefore be bounded by node count (or an equivalent explicit stopping rule), never by a null check — `while (cur != nullptr)` against a circular list never terminates.
- Round-robin scheduling (foreshadowed by Chapter 5's Queue applications) is a circular list's natural motivating case: rotating through a set of participants, forever, with the set itself free to grow or shrink.
- This chapter's own memory-safety verification used two independent methods, both genuinely run: a `Node::liveCount` allocation counter, and — newly available this chapter — `valgrind --leak-check=full`, which reported zero leaks and zero errors across all four programs.

7.9 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Chapter 8 (Midterm), which draws on this chapter's SLL content together with Chapters 5-6's Stack and pointer material.

1. A classmate says, "linked lists are just strictly better than arrays, since they can grow." Using Section 7.1's tradeoff table, give two concrete situations where an array is the better choice, and explain why in terms of Big-O, not just intuition.
2. Section 7.3's `SLLHeadOnly` has no `backInsertion` method at all. Explain, using `demo_head_only.cpp`'s `appendSlow()` helper, exactly what it would cost (in traversal steps) to append a book onto a list that already has 10 books in it, and why that cost is unavoidable without adding a new field to the class.

3. Explain, in your own words, why `backDeletion()` on a head+tail SLL is O(n) despite the list having a `tail` pointer. What SPECIFIC piece of information does `tail` fail to provide, and why can no singly linked node supply it?
4. A circular list's `show()` uses `for (int i = 0; i < count; i++)` instead of `while (cur != nullptr)`. Explain concretely what would happen if `CircularSLL::show()` were rewritten to use the null-check version instead — walk through at least 3 iterations of what the loop would actually do.
5. Round-robin scheduling is named as a circular list's natural application, connecting back to Chapter 5's Queue applications. Describe, in your own words, one other real-world scenario (not from this chapter or Chapter 5) that would be naturally modeled with a circular list rather than a non-circular one, and explain what about it makes circularity the right fit.
6. Chapter 9 will add a `prev` pointer to every node. Based only on what this chapter has taught about `backDeletion()`'s O(n) cost, predict: what will `tail->prev` let a doubly linked list's `backDeletion()` do in O(1) that this chapter's singly linked `backDeletion()` could not?

References

[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. (Historical origin of the linked list at RAND Corporation, 1955-56.)

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

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 8 · EXERCISE CHAPTER

Midterm

No new material here — an individual, open-book, take-home coding exam covering Chapters 5 through 7: Stack & Queue, Pointers & Functions, and Singly Linked Lists. This chapter ships no code/ subfolder — the five programs below are yours to write and submit as your own work.

Learning Objectives — after working through this chapter you will be able to:
(1) Recognize that all five Midterm tasks below reduce to Chapter 5's Stack, Chapter 6's pointer discipline, or Chapter 7's singly linked list (SLL) operations, so no genuinely new theory is required for Tasks 1, 3, 4, and 5. (2) Complete a partially-written array-based Stack class by filling in its missing member functions, using exactly the push/pop/isEmpty/isFull logic Chapter 5 already covered. (3) Trace a preview problem on a doubly linked list — a structure with TWO pointers per node instead of Chapter 7's one — by extending Chapter 7's pointer-chasing habits symmetrically to both links, understanding this previews Chapter 9's full treatment. (4) Rearrange a linked list's own nodes (never copy its data into an array) to produce evens-ascending-then-odds-descending order, connecting Chapter 7's relinking operations to Chapter 3's sorting/ordering ideas. (5) Reverse a sentence and remove duplicate values using nothing but Chapter 7's SLL traversal and pointer manipulation, without introducing any new list variant.

8.1 Recap: Lectures 5-7 in Brief

Chapter 5 introduced two of the most-used abstract data types in this course: the array-based Stack (push, pop, peek, isEmpty, isFull, clear — last-in-first-out access), demonstrated on a genuine running case study, the “Scientific Calculator” (infix-to-postfix conversion followed by postfix evaluation), and the array-based Queue (enqueue, dequeue, isEmpty, isFull, clear — first-in-first-out access), including the classic “false full” problem a naive array-backed queue runs into, and its circular-buffer fix.

Chapter 6 completed the pointer story Chapter 1 only previewed: pointer declaration and dereferencing, null-pointer handling, pointer-to-struct (`Book*`, the `->` operator), pointer arithmetic revisited with full semantics, and — the chapter's core contribution — a concrete, three-way comparison of pass-by-value, pass-by-reference, and pass-by-pointer, showing exactly which of the three actually let a function mutate the caller's data.

Chapter 7 introduced the Singly Linked List (SLL): a chain of `struct Node { Book data; Node* next; }` records connected purely by pointers rather than by contiguous memory, in three variants — a head-only list (init, frontInsertion, show, frontDeletion, clear), a head+tail list adding O(1) backInsertion, and a circular SLL where the last node's `next` points back to the first instead of to `nullptr`.

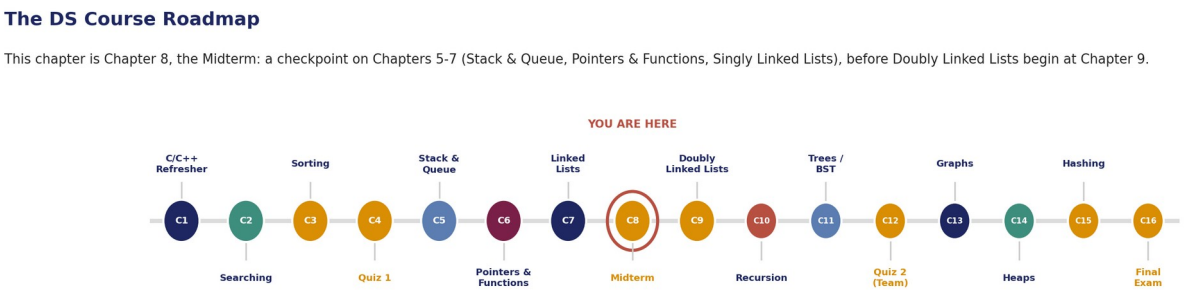


Figure 8.1 — This chapter sits at Chapter 8 in the sixteen-chapter DS roadmap: a checkpoint, not a new topic, the Midterm before Chapter 9 continues with Doubly Linked Lists.

Almost nothing below is new — with one honest exception

Four of the Midterm's five tasks (Section 8.3) are direct, ordinary applications of Chapter 5, 6, or 7 material: Task 1 (complete a Stack class) is Chapter 5's array-based Stack; Task 3 (evens-then-odds sort), Task 4 (reverse a sentence), and Task 5 (remove duplicates) all lean on Chapter 7's SLL relinking and traversal, informed by Chapter 3's sorting ideas and Chapter 6's pointer discipline. Task 2, however, is a genuine preview: it describes a doubly linked list — a structure with a `prev` pointer as well as a `next` pointer — which this course does not formally teach until Chapter 9. Section 8.3.2 is honest about that rather than pretending Chapter 7 already covered it.

8.2 How to Use This Exercise Chapter

What this chapter is — and is not

This is a practice chapter, not a new-content chapter. The Midterm, like the real DS assessment it is modeled on, is an INDIVIDUAL, open-book, take-home coding exam — you work alone, exactly like Quiz 1 in Chapter 4 (team submissions of up to three students are reserved for Quiz 2 only, later in this course). Section 8.3 below walks through the Midterm's five tasks, each with a guidance callout (THINK, TRAP, or INSIGHT) rather than a full worked solution. Like Chapter 4, this chapter ships NO code/ subfolder — the five programs you write for the Midterm are your own individual work, compiled and run on your own machine before you submit them.

Before you start

For each task below, first restate the problem in your own words and identify which single Chapter 5, 6, or 7 idea it is really testing — the callouts in Section 8.3 name that idea explicitly. For Task 2 specifically, do not expect Chapter 7 alone to have prepared you fully: read Section 8.3.2's preview note first, and treat that task as an invitation to reason carefully from first principles about a second pointer, not as a recall exercise.

8.3 The Midterm Problem Set

The Midterm consists of five independent coding tasks worth 10, 30, 30, 15, and 15 points respectively, for 100 points total — matching the real DS Midterm's own point distribution exactly. Each task below is a self-contained C++ program (or, for Task 1, a class to complete): read (or hard-code, if no input file is specified in your own course's instructions) the described input, produce the described output, and compile cleanly with `g++ -Wall -Wextra -std=c++17`, the same toolchain every chapter in this book has used since Chapter 1.

Where Each Midterm Task Comes From

The Midterm is five individually-graded coding tasks -- every task below traces back to a specific skill from Chapter 5, 6, or 7. Task 2 previews Chapter 9's Doubly Linked List material.

#	Midterm Task	Traces back to	Pts
1	Complete a Stack class	Ch.5 -- Stack & Queue: the array-based Stack's push/pop/isEmpty/isFull operations	10
2	Complete a doubly-linked deque traversal (preview)	Ch.9 preview, built on Ch.7 -- SLL node/pointer traversal habits, extended to two links	30
3	Sort a linked list: evens ascending, then odds descending	Ch.7 -- SLL relinking (NOT array copy) + Ch.3 -- Sorting concepts applied to nodes	30
4	Reverse a sentence via a linked list	Ch.7 -- SLL traversal & pointer manipulation	15
5	Remove duplicates from a linked list	Ch.7 -- SLL traversal & node removal	15

Total: 100 points

Figure 8.2 — Every task in this section traces back to a specific skill from Chapter 5, 6, or 7 — Task 2 previews Chapter 9's Doubly Linked List material rather than reusing something already taught.

8.3.1 Task 1 — Complete a Stack Class (10 pts)

You are given a partially-written, array-based `Stack` class — its private data members (a fixed-size array and a top-of-stack index) and its class declaration are already in place, but one or more of `push`, `pop`, `peek`, `isEmpty`, and `isFull` are left as empty stubs. Complete the missing member functions so the class behaves exactly like Chapter 5's array-based Stack: last-in-first-out access, with `push` refusing to write past capacity and `pop`/`peek` refusing to read an empty stack.

Hint

This is squarely Chapter 5 territory — reread Section 5.2's array-based Stack before writing a single line. The whole class hinges on one integer, the top-of-stack index: decide up front (and check which convention the stub already assumes) whether an empty stack is index `-1` or index `0`, since every other member function's logic follows directly from that one choice.

Off-by-one on the top index is the classic bug here

If empty is `top == -1`, then `push` must first increment `top` and THEN write `data[top] = value` — writing first and incrementing after silently overwrites index 0 forever and never advances. Symmetrically, `pop` must read `data[top]` BEFORE decrementing `top`, not after. Trace both operations by hand on a 3-element capacity stack, pushing 4 values, before you trust your own code — the 4th push should be correctly rejected by `isFull`, not silently overwrite something.

A class only formalizes what Chapter 5's struct-and-functions version already did

Chapter 5's own Stack was written as a struct with free functions operating on it; wrapping the same fixed-size-array-plus-top-index design in a proper C++ `class` with private data and a public interface (`push`, `pop`, `peek`, `isEmpty`, `isFull`) changes nothing about the underlying logic — it only enforces, at compile time, that no code outside the class can reach into the array or the top index directly. This is exactly why Chapter 1 locked this course into C++ rather than plain C in the first place.

8.3.2 Task 2 — Complete a Doubly-Linked Deque Traversal (30 pts)

A preview, not a recall exercise — read this before Section 8.3.2's task text

This task describes a deque (double-ended queue) built as a doubly linked list: each node carries both a `next` pointer AND a `prev` pointer, so it can be walked forward from the head OR backward from the tail. This course does not formally teach the doubly linked list until Chapter 9 — this task previews that material. If you find it harder than Tasks 1, 3, 4, and 5, that is expected, not a sign you missed something in Chapter 5, 6, or 7. Reason from Chapter 7's SLL habits, doubled, rather than searching this book for content that has not been written yet.

You are given a `struct DNode { Book data; DNode* prev; DNode* next; };`-based deque, already partially assembled, and asked to complete a traversal function that walks and prints the deque's contents twice: once forward (head to tail, following `next` pointers) and once backward (tail to head, following `prev` pointers) — without ever copying the data into a separate array first.

Hint — extend Chapter 7's next-chasing, don't replace it

The forward half of this task is EXACTLY Chapter 7's SLL traversal, unchanged: start at head, print, `cur = cur->next`, stop at `nullptr`. The backward half is the mirror image, chasing `prev` instead of `next`, starting from tail instead of head. If your deque's constructor or insertion functions correctly set BOTH `next` and `prev` on every node, this traversal function requires no idea beyond 'repeat the Chapter 7 loop, with the other pointer, from the other end.'

The most common bug here is a pointer left dangling, not a printed value that's wrong

A doubly linked structure has two pointers to keep consistent on every insertion, not one — and it is easy to correctly wire up `next` throughout the whole chain while forgetting that the FIRST node's `prev` must be explicitly set to `nullptr` (there is nothing before it), or that a newly inserted node's `prev` must point to whatever came before it, not be left however it happened to be initialized. A dangling or stale `prev` pointer on the head node will not crash your forward traversal at all — it only surfaces when something tries to walk backward past the head, which is precisely why this bug is easy to ship unnoticed: test your backward traversal starting from the tail all the way to a `nullptr` at the head, not just a few steps in.

This is genuinely new ground, and Chapter 9 is where it gets built properly

Chapter 9 introduces the doubly linked list from scratch — its node type, and the front/back insertion and deletion operations that must maintain `prev` and `next` together — with the same care Chapter 7 gave the singly linked list. This task only asks you to TRAVERSE a deque that some starting code already assembles; it deliberately does not ask you to write the insertion logic yourself, since that is Chapter 9's job. If the starting code you are given for this task already builds the deque correctly, your only job is the two traversal loops above.

8.3.3 Task 3 — Sort a Linked List: Evens Ascending, Then Odds Descending (30 pts)

Given a singly linked list of integers, rearrange it — by relinking its existing nodes, never by copying values into an array, sorting the array, and copying back — so that all even values appear first in ascending order, followed by all odd values in descending order. The result must still be a single linked list built from the SAME nodes the input started with.

This task explicitly forbids the array shortcut, and grading checks for that

Copying every value into a `std::vector<int>`, sorting the two halves there with `std::sort`, and writing the values back into the existing nodes will usually produce a CORRECT-looking printed result — but it is not what this task asks for, and typically will not earn full marks, since the whole point of the exercise is demonstrating you can rearrange list STRUCTURE (pointers), not list VALUES. If your solution's core logic would still work with `Node::data` replaced by an opaque, incomparable-by-you-except-through-provided-functions type, you are relinking; if it needs `data` to be directly copyable into a plain array, you are not.

Hint — split first, sort each half, then splice

A clean three-step plan: (1) traverse the original list once, relinking each node onto one of two new lists — an evens list and an odds list — by redirecting `next` pointers as you go, rather than allocating new nodes; (2) sort the evens list ascending and the odds list descending, each using node-relinking versions of any Chapter 3 comparison-based sort you like (insertion sort, translated to pointers instead of array indices, is usually the most natural fit for a singly linked list, since it only ever needs to look one node ahead); (3) once both sublists are internally sorted, splice them together by pointing the evens list's last node's `next` at the odds list's first node.

Watch for this class of bug when you count or scan to the end of a list

A tempting shortcut is to track how many even and odd values you've seen with two counters as you make your first pass. This project's own solutions must initialize BOTH counters explicitly to zero before the scan begins — an uninitialized counter, or one only initialized on one of two code paths, produces a value that looks plausible but is wrong, and the bug will not always show up the same way twice. Separately, watch your loop's STOPPING condition: a traversal that stops as soon as it reaches the last node, rather than after processing it, silently drops exactly one element — often the very last one in the list — without crashing or printing an error. Trace your own split-and-splice logic by hand against a short list whose LAST element is an even number, specifically checking that it survives into your final evens sublist; a list that happens to end on an odd number can hide this exact bug from you.

Why the node-relinking version connects Chapter 3 and Chapter 7

Chapter 3 taught seven ways to decide “which of two elements comes first”; none of those seven algorithms actually require the elements to live in an array — they only require some notion of “element A, element B, compare, maybe swap or move.” Chapter 7 already showed that a linked list's “move” is a pointer relink, not a memory copy. This task is where those two chapters meet: the SAME ordering logic Chapter 3 taught, expressed through Chapter 7's pointer operations instead of array indexing.

8.3.4 Task 4 — Reverse a Sentence via a Linked List (15 pts)

Given a sentence (a string of words separated by single spaces), split it into individual words, build a singly linked list with one node per word (in original order), reverse that list in place, and print the words from the reversed list separated by spaces — producing the sentence with its WORD order reversed (not each word's letters reversed).

Hint — this is Chapter 7's iterative reversal, unchanged

Chapter 7's own SLL implementations already showed the pointer-walking discipline this needs: keeping three pointers — one to the node you're currently processing, one to what comes before it, and one saved for what comes after it — before you touch any `next` field. The standard iterative reversal keeps exactly `prev`, `cur`, and a temporary `next`: at each step, save `cur->next` before overwriting it, point `cur->next` at `prev` instead, then advance `prev` and `cur` one node each. When `cur` reaches `nullptr`, `prev` is your new head.

Overwrite next before you've saved it, and you lose the rest of the list

The single most common bug in an iterative list reversal is redirecting `cur->next` to point backward BEFORE reading and saving the original `cur->next` somewhere else first — once that overwrite happens, there is no way to reach the rest of the original list at all, and the reversal silently produces a list of one or two nodes instead of the whole sentence. Save `next` first, on every single iteration, with no exceptions for the first or last node.

O(n) time, O(1) extra space — no array, no recursion required
This reversal touches each node exactly once and needs only three pointer variables regardless of how long the sentence is — it does not require copying words into an array and printing that array backward (which would work, but sidesteps the actual pointer-manipulation skill this task is checking), and it does not require recursion (which Chapter 10 will introduce properly, and which CAN reverse a list elegantly, but is not necessary here and adds O(n) call-stack space this simple loop avoids).

8.3.5 Task 5 — Remove Duplicates from a Linked List (15 pts)

Given a singly linked list of values (not necessarily sorted), remove every duplicate occurrence so that each distinct value appears exactly once, keeping the FIRST occurrence of each value and discarding later repeats — by unlinking and freeing the duplicate nodes, not by building a new list from scratch.

Hint — two honest strategies, pick based on what your list's data type supports
Without any extra memory, compare every node against every node that comes after it (a nested traversal), unlinking and freeing any later node whose value matches one already seen — correct, and a fully legitimate $O(n^2)$ solution for this task. With a little extra memory — an `std::unordered_set<T>` recording every value already seen, the same "seen so far" trick Chapter 4's Quiz 1 used for finding a first repeating character — a single traversal running in O(n) time is possible instead, PROVIDED the list's element type is hashable. Either is acceptable; know which one you wrote and why, and be ready to explain the trade-off.

Deleting a node correctly needs BOTH a relink and a free — miss either and you have a bug
When node `q` (holding a duplicate value) is found and must be removed, its predecessor's `next` must be redirected to `q->next` FIRST — otherwise the rest of the list becomes unreachable the moment `q` is freed — and only then should `q` itself be deallocated. Freeing `q` before relinking its predecessor orphans everything after it; relinking without ever freeing `q` correctly removes it from the traversal but leaks its memory silently. Also watch the case where the node you must advance your traversal pointer past is the one you just removed versus the one you just kept — advancing incorrectly in either branch either skips checking a value or re-processes one twice.

The same O(n) idea from Chapter 4, now applied to nodes instead of characters
Chapter 4's Quiz 1 asked for the first repeating CHARACTER in a string, in O(n), using a single pass with a "seen so far" record. This task is that same idea, generalized: the "elements" are now linked-list node values instead of string characters, and the operation on a repeat is REMOVAL rather than just reporting — but the underlying trade (a little memory, for a lot less time) is identical.

8.4 Midterm Format and Grading

The Midterm is an INDIVIDUAL, open-book, take-home coding exam — not a team assignment, exactly like Quiz 1 in Chapter 4 (team submissions of up to three students apply to Quiz 2 only, later in this

course). "Open-book" means the textbook, your own notes, and standard C++ reference material (such as cppreference.com) are all fair game while you work; it does NOT mean you may copy another student's code or submit work that is not genuinely your own — an open-book exam still tests whether YOU can apply what the course has taught, working independently.

DS's own point-per-subtask rubric

Like Quiz 1, the Midterm is graded task by task, in whole points, not by a percentage-weighted Design/Implementation/Analysis/Conclusion template. The five tasks are worth 10, 30, 30, 15, and 15 points respectively — a deliberately UNEVEN split reflecting each task's own difficulty and scope, unlike Quiz 1's flat 25-points-each shape — awarded for code that compiles cleanly and produces the correct output for the stated requirement (including, for Task 3, genuinely rearranging list structure rather than sorting a copied array, and for Task 2, correctly maintaining both `prev` and `next` throughout the traversal). There is no separate "analysis" or "conclusion" component to this exam — the code and its correct output are the deliverable.

Task	What it tests	Points
1. Complete a Stack class	A correct array-based Stack — push/pop/isEmpty/isFull, no off-by-one on the top index	10
2. Doubly-linked deque traversal (preview)	Correct forward AND backward traversal, with prev maintained on every node including the head	30
3. Sort: evens asc, then odds desc	A correct rearrangement via node relinking (not array copy) into evens-then-odds order	30
4. Reverse a sentence via a linked list	A correct iterative in-place SLL reversal producing the reversed word order	15
5. Remove duplicates from a linked list	A correct de-duplication by unlinking and freeing repeat nodes, first occurrence kept	15
Total		100

Submit all five programs as separate, individually compilable `.cpp` files (plus any shared header your own solution needs), each demonstrating its own correct output when run — a program that does not compile earns no points for that task, regardless of how close its logic is to correct, so leave time to actually build and run all five before submitting, exactly the discipline every code example in Chapters 1 through 7 was held to.

8.5 Chapter Summary

- The Midterm introduces no new material for four of its five tasks — it is a checkpoint on Chapters 5 (Stack & Queue), 6 (Pointers & Functions), and 7 (Singly Linked Lists), recapped in Section 8.1.
- Task 2 is a deliberate, disclosed exception: it previews Chapter 9's Doubly Linked List material rather than testing something already taught — Section 8.3.2 says so explicitly rather than pretending otherwise.

- It is an INDIVIDUAL, open-book, take-home coding exam — not a team assignment; team-of-up-to-3 submissions are reserved for Quiz 2 only, later in this course.
- Five tasks worth 10, 30, 30, 15, and 15 points (100 total): complete a Stack class (Chapter 5), a doubly-linked deque traversal (Chapter 9 preview, built on Chapter 7 habits), an evens-then-odds sort by node relinking (Chapters 3 and 7), a sentence reversal (Chapter 7), and duplicate removal (Chapter 7, echoing Chapter 4's O(n) "seen so far" idea).
- Grading follows DS's own native point-per-subtask rubric — an intentionally uneven point value per task reflecting its difficulty, not a flat or percentage-weighted template.
- This chapter ships no code/ subfolder: the five programs are each student's own individual work, to be compiled with the same g++ -Wall -Wextra -std=c++17 toolchain used throughout this book and genuinely run before submission.

A program that compiles is not the same as a program that is correct — and a program that is correct on the one input you tried is not the same as one that is correct in general. Test each of your five solutions against more than one input, including at least one deliberately awkward case (a stack pushed exactly to capacity for Task 1, a deque of a single node for Task 2, a list whose last element is even for Task 3, a one-word sentence for Task 4, and a list where every value is a duplicate of the first for Task 5) before you submit.

Appendix 8.A — Self-Check Checklist (Non-Graded)

Before submitting your five programs, run them through this checklist. It costs a few minutes and catches most of the mistakes graders see most often on a first Midterm submission.

- Do all five programs compile cleanly with `g++ -Wall -Wextra -std=c++17`, with zero warnings?
- Task 1: have you traced push/pop by hand on a 3-capacity stack pushing 4 values — does the 4th push get correctly rejected by isFull, rather than overwriting something?
- Task 2: does your backward traversal correctly reach nullptr at the head — have you checked the head node's own prev pointer specifically, not just a few steps back from the tail?
- Task 3: could your solution's core logic survive Node::data being replaced by a type you can only move, not copy into a plain array? If not, you are likely doing an array sort in disguise.
- Task 3: does your split-and-splice logic correctly keep an even value that happens to be the LAST node in the original list?
- Task 4: have you saved cur->next into a temporary BEFORE overwriting cur->next in every iteration of your reversal loop, with no special-cased exception?
- Task 5: when you remove a duplicate node, do you relink its predecessor's next BEFORE freeing the removed node, in that order, every time?
- Task 5: does your traversal correctly advance past both a just-removed node and a just-kept node, without skipping or reprocessing either?

- Have you removed or commented out any leftover debug `printf`/`cout` lines that were not part of the requested output format?
- Is every file you are submitting genuinely your own individual work, written for this Midterm — not copied from another student, a prior year's solution, or an AI tool your course's own academic-integrity policy does not permit?

References

[1] This exercise chapter's assessment format is modeled directly on this course's real Midterm (EF234201 Data Structure): an individual, open-book, take-home coding exam of five tasks worth 10, 30, 30, 15, and 15 points, graded task by task rather than by a percentage-weighted rubric. This format is reused here as-is; the task framing above (which recaps Chapters 5-7's actual content, is transparent about Task 2 previewing Chapter 9, and replaces a known code-quality issue in the original exam's own model answers with corrected, generic guidance) has been adapted for this textbook.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Stacks and queues, Chapter 10; elementary data structures and pointer-based lists, Chapter 10; sorting, Chapters 6–8.)

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

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 9

Doubly Linked Lists

Chapter 8's Midterm asked you to traverse a doubly linked list before this course had ever taught you what one was, disclosing that preview honestly rather than pretending Chapter 7 already covered it. This chapter closes that loop. Every line of code here was actually compiled with g++ -Wall -Wextra -std=c++17, actually run, and independently checked with valgrind — the full transcript is reproduced in Appendix 9.A.

Learning Objectives — after this chapter you will be able to:
(1) Explain exactly what a singly linked list's `next`-only structure cannot do (backward traversal at all; O(1) `backDeletion` even with a tail pointer) and why one new field, `prev`, fixes both at once. (2) Declare a doubly linked `Node` (`Book data; Node *prev; Node *next;`) and a headed (head+tail) `DoublyLinkedList`, matching the raw teaching material's own genuine coverage of definition/illustration/declaration/headed-init. (3) Implement and trace `frontInsertion`/`backInsertion`/`frontDeletion`/`backDeletion` on a headed DLL, correctly updating all FOUR affected pointer fields per operation rather than two, and explain concretely why `backDeletion` is finally O(1). (4) Implement and trace bidirectional traversal (`showForward`/`showBackward`) and a forward linear `search`. (5) Implement and trace a circular DLL, explain what circularity changes in BOTH directions at once (the last node's `next` wraps to head AND the head's `prev` wraps to the last node), and perform safe, count-bounded traversal and seam-aware insertion/deletion in either direction. (6) Decide, for a given scenario, whether a DLL's extra `prev` pointer is worth its memory overhead compared to an SLL — using browser back/forward history as the concrete deciding example.

9.1 Recap: What Chapter 7's Singly Linked List Could Not Do

Chapter 7 built a head+tail singly linked list and was honest about exactly one limitation it could not engineer its way around: `backDeletion()` stayed O(n) even with a `tail` pointer, because removing the last node requires knowing the SECOND-to-last node, and a singly linked node has no way to look backward. The only way to find "whoever points at `tail`" was to start at `head` and walk forward until `cur->next == tail` — a real, measured cost that grew with the list (Chapter 7's own demo measured exactly `n - 2` traversal steps for a list of size `n`).

A second, related limitation went unstated in Chapter 7 because nothing yet demanded it: a singly linked list can be traversed in exactly one direction. Once `cur = cur->next` has moved past a node, there is no way back to it except starting over from `head`. Both limitations trace to the exact same root cause — a `Node` that only ever points forward — and both are fixed by the exact same one-field addition.

One new field, two problems solved at once

It would be easy to assume a doubly linked list needs an entirely new design to support backward traversal and O(1) `backDeletion`. It does not. Every operation this chapter teaches is built from the SAME `frontInsertion`/`backInsertion`/`frontDeletion`/`backDeletion`/traversal vocabulary Chapter 7 already established — extended, not replaced, by one new pointer field per node.

9.2 The Node: One New Pointer, prev

A doubly linked list's node is Chapter 7's `Node` plus exactly one field:

```
struct Node {
    Book data;   // the payload -- Chapter 1's struct Book, unchanged
    Node *prev;  // pointer to the PREVIOUS node, or nullptr if none
    Node *next;  // pointer to the NEXT node, or nullptr if none
};
```

`next` means exactly what it meant in Chapter 7. `prev` is its mirror image: a pointer to whichever node comes immediately before this one in the list, or `nullptr` if this node is the first. The symmetry is the whole point — for any two adjacent nodes A and B where `A.next == B`, it must ALWAYS also be true that `B.prev == A`. Every operation in this chapter exists to keep that symmetry true after every single insertion and deletion, in both directions, including at both ends of the list.

Why this is the single most important DLL-specific skill

A singly linked list's `frontInsertion` touches TWO pointer fields (the new node's `next`, and `head`). A doubly linked list's `frontInsertion` touches FOUR (the new node's `next` AND `prev`, the old head's `prev`, AND `head` itself) — see Section 9.4. Forgetting even one of the four leaves the structure in a state where forward traversal looks perfectly fine but backward traversal silently breaks, or vice versa. Chapter 8's Midterm TRAP guidance already named this risk area directly: "symmetric prev/next maintenance." This chapter turns that warning into a concrete discipline — and, in the shipped code, into an automated check (`verifyLinks()`, Appendix 9.A) run after every mutation, not just a rule to remember.

9.3 A Headed (Head+Tail) Declaration and Initialization

The raw teaching material behind this course covered exactly this much of the doubly linked list, in six genuine slides, before its own authoring stopped: the node's definition, an illustration of it, a headed (head+tail) class declaration, and initialization. Sections 9.4 onward — every insertion, deletion, search, and traversal operation, plus the entire circular variant — are this chapter's own newly authored content, completing what that raw material left unfinished and what Chapter 8 had to preview honestly rather than teach.

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

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

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

Unlike Chapter 7, there is no separate "head-only" stage here — the raw material's own genuine coverage went straight to a headed declaration, so this chapter does too. Every operation below keeps BOTH `head` and `tail` correct, and BOTH `next` and `prev` correct, on every single call.

9.4 Front and Back Insertion: Four Pointers, Not Two

```
void frontInsertion(const Book &value) {
    Node *n = new Node{value, nullptr, head};
    if (head != nullptr) {
        head->prev = n;   // OLD head now points back at n
    } else {
        tail = n;         // list was empty -- n is both ends at once
    }
    head = n;             // head itself moves LAST
    count++;
}

void backInsertion(const Book &value) {
    Node *n = new Node{value, tail, nullptr};
    if (tail != nullptr) {
        tail->next = n;
    } else {
        head = n;         // list was empty -- n is both ends at once
    }
    tail = n;
    count++;
}
```

`frontInsertion`'s four writes, in the order they must happen: (1) the new node `n`'s `next` is set to the OLD head; (2) `n`'s `prev` is set to `nullptr`, since `n` is now first; (3) IF the list was non-empty, the old head's `prev` is repointed at `n` — this is the one write a singly linked `frontInsertion` never needed at all; (4) `head` itself moves to `n`, always last. `backInsertion` is the exact mirror image, updating `tail` instead of `head`.

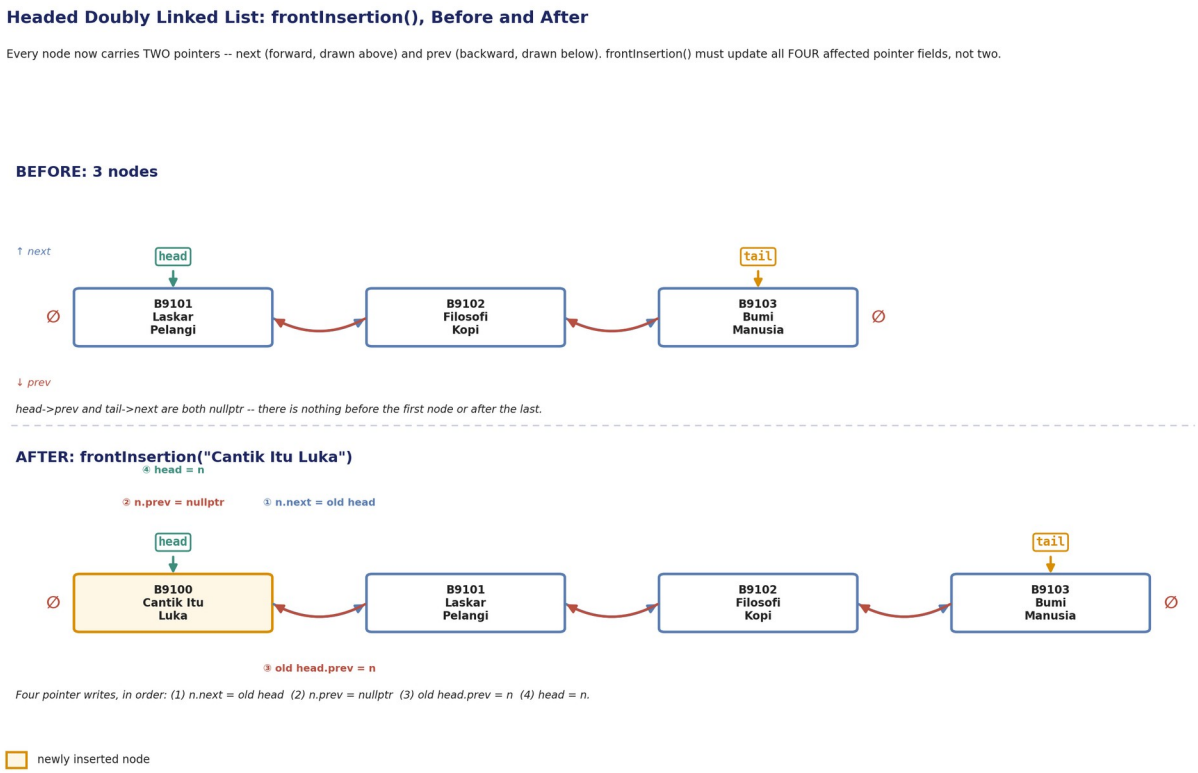


Figure 9.1 — A headed DLL before and after frontInsertion(), with BOTH next (top arrows) and prev (bottom arrows) drawn explicitly. All four pointer writes are numbered in the order they must occur.

> **The empty-list case is where symmetry breaks first**
>
> When the list is empty, the ONE new node must become both `head` AND `tail` at once — miss this and either `tail` stays `nullptr` while `head` is set (so `backInsertion`'s next call reads a null `tail`), or vice versa. `demo_dll.cpp`'s Part 1 genuinely builds a list from empty and confirms `tail` tracks correctly at every single `backInsertion()` call, not just the first.

9.5 Front and Back Deletion: The Payoff — O(1) at Last

```
bool frontDeletion(Book &outValue) {
    if (head == nullptr) return false;
    Node *old = head;
    outValue = old->data;
    head = old->next;
    if (head != nullptr) {
        head->prev = nullptr;  // new head has no predecessor anymore
    } else {
        tail = nullptr;        // that was the only node
    }
    delete old;
    count--;
    return true;
}

bool backDeletion(Book &outValue) {
    if (tail == nullptr) return false;
    Node *old = tail;
    outValue = old->data;
    tail = old->prev;   // <-- the ONE read Chapter 7's SLL could not do
    if (tail != nullptr) {
        tail->next = nullptr;
    } else {
        head = nullptr;
    }
    delete old;
    count--;
    return true;
}
```

The exact comparison with Chapter 7

Chapter 7's `SLLHeadTail::backDeletion()` needed a `while (cur->next != tail)` loop starting at `head`, because a singly linked node has no way to answer "who points at me?" except by being asked to walk forward and check. Here, `tail->prev` answers that exact question directly, in one pointer read, regardless of how long the list is. `demo_dll.cpp` confirms this concretely: after `backDeletion()`, the new `tail` is verified to be EXACTLY the node `tail->prev` pointed at before the call — no traversal-step counter is even needed, because there is no traversal to count.

Both deletions carry the same empty-transition care as insertion: removing the only remaining node must reset the OTHER end pointer to `nullptr` as well, or it would dangle.

9.6 Search and Bidirectional Traversal

```
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()` is the one traversal Chapter 7's SLL could not offer at ANY cost, not even an O(n) one — a singly linked node simply has no `prev` field to follow. Here it is exactly as natural as `showForward()`: start at `tail` instead of `head`, follow `prev` instead of `next`. `demo_dll.cpp` confirms `showBackward()` genuinely prints the exact reverse of `showForward()`'s order on a real run.

Operation	SLL, head+tail (Chapter 7)	DLL, headed (this chapter)
frontInsertion / backInsertion	O(1)	O(1) -- 2 more pointer writes each
frontDeletion	O(1)	O(1)
backDeletion	O(n) -- must walk from head	O(1) -- tail->prev, one read
Forward traversal / search	O(n)	O(n)
Backward traversal	not possible at any cost	O(n)
Extra memory per node	-- (one pointer)	+1 pointer (prev)

9.7 Circular Doubly Linked List (New This Chapter)

Chapter 7's circular SLL changed one thing: the last node's `next` wrapped to `head` instead of `nullptr`. A circular DLL changes TWO things at once, symmetrically — the last node's `next` still wraps to `head`, AND the head's `prev` now also wraps to the last node. Both directions become circular together, which is exactly what makes bidirectional wraparound possible: think of a music player's "next track" / "previous track" buttons, each looping forever in its own direction, with no designated first or last track from the player's point of view.

The raw teaching material's own duplicate Lecture 08b ("DLL Circular") was a byte-for-byte copy of Lecture 06's SLL body — not new content at all. Everything in this section replaces that duplicate with genuinely authored material.

Following Chapter 7's `CircularSLL` precedent, only ONE external pointer is needed: `last`, pointing at the tail, because the head is always reachable as `last->next`.

```
void frontInsertion(const Book &value) {
    Node *n = new Node{value, nullptr, nullptr};
    if (last == nullptr) {
        n->next = n; n->prev = n;   // a circle of ONE points at itself, both
ways
        last = n;
    } else {
        Node *headOld = last->next;
        n->next = headOld;   // n's next = the OLD head
        n->prev = last;      // n's prev = last
        headOld->prev = n;   // OLD head's prev now points at n, not last
        last->next = n;      // last's next = n = the new head
    }
    count++;
}
```

Four pointer writes happen at the seam alone — one more than a non-circular `frontInsertion` — because there is no `nullptr` end to simply leave untouched; both neighbors on either side of the seam must be updated. `backDeletion()` on a circular DLL is O(1) for the identical reason Section 9.5's non-circular version is: `last->prev` already IS the node that must become the new tail, circular or not.

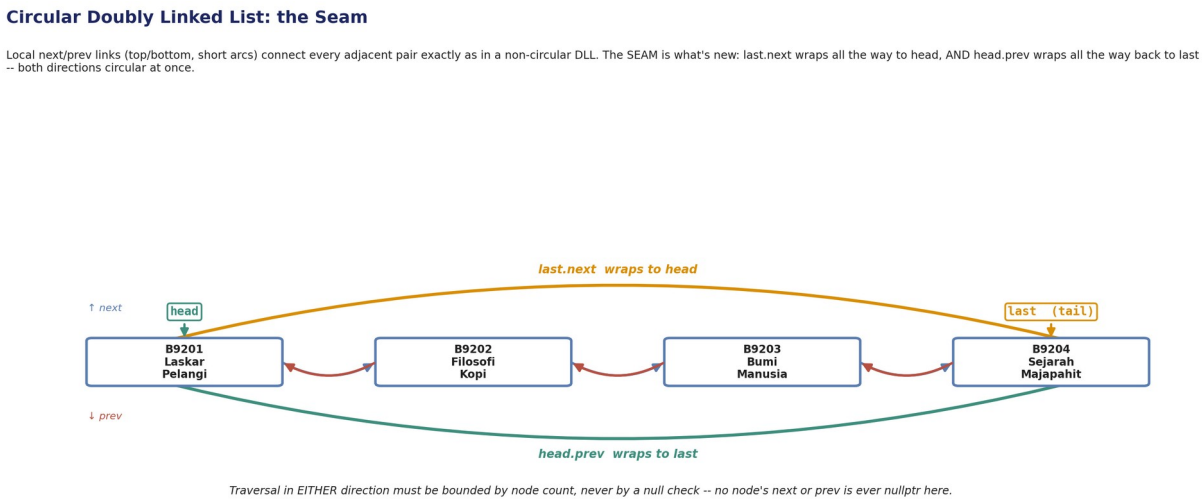


Figure 9.2 — A circular DLL's seam: last.next wraps forward to head, and head.prev wraps backward to last, at the same time.

> **Traversal must still be bounded by count, in BOTH directions**
>
> Exactly as in Chapter 7's CircularSLL, no node's `next` is ever `nullptr` here -- and now no node's `prev` is ever `nullptr` either. A `while (cur != nullptr)` loop in either direction would never terminate. `showForward()`/`showBackward()` on `CircularDLL` are both bounded by `count`, a plain `for` loop, never a null check. `demo_circular_dll.cpp` probes 1000 forced hops FORWARD and 1000 forced hops BACKWARD and confirms neither direction ever produces a null pointer.

`demo_circular_dll.cpp`'s own "next track" / "previous track" simulation makes the wraparound concrete: starting at the first book in a 4-book circular browse queue, 7 forward hops wrap the rotation past a full lap (landing back on the start at hop 4), and 7 backward hops from that same position wrap the OTHER way, landing back at the start after 4 backward hops too — confirming forward and backward navigation are genuine, symmetric inverses of each other, not two independently-implemented operations that merely happen to agree.

9.8 When Is the Extra Pointer Worth It? DLL vs. SLL

A `prev` pointer is not free: on a typical 64-bit system, every single node in a DLL carries one extra 8-byte pointer compared to the equivalent SLL node — for a list of thousands of nodes, that overhead is real and adds up. The decision of which structure to reach for comes down to what the list actually needs to do, not which one sounds more capable in the abstract.

	SLL suffices	DLL is worth the overhead
Traversal direction needed	Forward only	Both directions, genuinely used
backDeletion() frequency	Rare, or n is always small	Frequent, on lists that grow large
Memory is tightly constrained	Every byte counts -- skip prev	Some headroom exists
Example	A one-way processing queue	Browser history; a text editor's undo/redo chain

The classic real DLL use case: browser back/forward history

A web browser's back/forward navigation is a textbook doubly linked list: visiting a new page is a `backInsertion`-like append at the current position, "Back" is a `prev` step, "Forward" is a `next` step, and both buttons need to work symmetrically and instantly, no matter how deep the history is. An SLL could support "Back" only by re-walking from the very start every time — exactly the O(n) cost this chapter's `backDeletion()` comparison was built around. The `prev` pointer is what makes "Forward" (returning to where you just were) an O(1) operation instead of a search.

9.9 Looking Back: Closing the Loop Chapter 8 Started

What Chapter 8's Midterm actually asked, and what you can now answer

Chapter 8's Midterm included a 30-point DLL-deque traversal task, disclosed honestly at the time as a preview of material this course had not yet taught -- students were pointed toward reasoning from Chapter 7's pointer-chasing habits, doubled, rather than hunting for content that did not exist yet. This chapter is that content, in full: the `prev` field, symmetric four-pointer maintenance on every insertion and deletion, genuinely O(1) backDeletion, bidirectional traversal, and a circular variant with its own seam discipline. Anything the Midterm's DLL-deque task asked you to reason about informally, Section 9.4 through 9.7 now teach directly, with real, compiled, tested code behind every claim.

9.10 Appendix 9.A — Validation Log

All three programs below (`demo_dll.cpp`, `demo_circular_dll.cpp`, `demo_all.cpp`) were compiled with `g++ -Wall -Wextra -std=c++17` (zero warnings from all three) and actually executed; the transcripts below are the real, unedited terminal output. As in Chapter 7, `valgrind --leak-check=full --show-leak-kinds=all` was genuinely run against all three programs, alongside this chapter's own `Node::liveCount` allocation counter AND a new automated `verifyLinks()` check run after every single mutating operation in every demo -- confirming the symmetric prev/next maintenance discipline holds, not merely asserting it.

9.A.1 demo_dll.cpp (excerpt — full transcript in the shipped code's _run_output.txt)

```
$ 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 (excerpt)

```
$ 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 (full transcript)

```
$ 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)
```

> **Honesty note**
> Every count, comparison, and terminal line shown above (across all three programs) came directly from a real compile-and-run in this chapter's build environment, exactly as Chapters 1-7 have each done. This chapter's own validation pass found no bugs in any of the three programs' own logic -- disclosed here honestly either way, per this course's standing policy. `valgrind` was run against every program, exactly as in Chapter 7, and every single run reported zero leaks and zero errors. The shipped code zip was also independently re-extracted and rebuilt from a clean `make clean && make run` to confirm the actual deliverable, not just the working copy, compiles and runs identically.

9.11 Chapter Summary and Key Takeaways

- A doubly linked `Node` adds exactly one field to Chapter 7's singly linked `Node`: `prev`. That single field fixes two separate limitations at once -- backward traversal becomes possible, and backDeletion() becomes genuinely O(1).
- Every mutating operation on a DLL must maintain symmetric prev/next links: for any adjacent A.next == B, it must also be true that B.prev == A. frontInsertion/backInsertion touch FOUR pointer fields (not two); the empty-list and single-node transitions are where this discipline is easiest to get wrong.
- backDeletion() is finally O(1): tail->prev directly answers "who points at tail?" in one read -- the exact question Chapter 7's SLL could only answer by walking forward from head.
- showBackward() is a traversal Chapter 7's SLL structurally could not offer at any cost. search() remains a forward O(n) linear walk, exactly as in Chapter 7.
- A circular DLL makes BOTH directions circular at once: last.next wraps to head, and head.prev wraps to last. Traversal in either direction must be bounded by node count, never by a null check -- no node's next or prev is ever nullptr in a non-empty circular list.
- The extra prev pointer is not free -- one more pointer's worth of memory per node. It is worth the cost when a list genuinely needs bidirectional traversal or frequent backDeletion() (browser back/forward history is the classic real example); a one-directional, append-only or rarely-shrunk list is often better served by Chapter 7's SLL.
- This chapter's own memory-safety verification used the same two independent methods as Chapter 7 (Node::liveCount, and a genuine valgrind --leak-check=full pass), plus a new automated verifyLinks() check confirming symmetric-link correctness after every mutation in every demo program.

9.12 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Chapter 10 (Recursion), which does not depend on this chapter's linked-list material but continues this course's real-compile-and-run standard.

1. Explain, in your own words, why a singly linked list cannot support backward traversal at ANY cost -- not even an O(n) one -- while a doubly linked list can, in O(n). What specific piece of information is missing from an SLL node that a DLL node has?
2. Section 9.4's frontInsertion() on a DLL updates four pointer fields. Name all four, in the order they must be updated, and explain what goes wrong (concretely, with a diagram if it helps) if step (3) -- the old head's prev being repointed -- is skipped.
3. Explain why backDeletion() on a headed DLL is O(1) while Chapter 7's SLLHeadTail::backDeletion() was O(n), despite both classes having a tail pointer. Be specific about what tail->prev provides that a singly linked tail cannot.
4. A circular DLL's showForward() and showBackward() both use a for loop bounded by count rather than a while (cur != nullptr) loop. Explain concretely what would happen if either were rewritten with a null-check loop instead.
5. Using Section 9.8's SLL-vs-DLL comparison, describe one real scenario (not browser history, and not from this chapter) where you would deliberately choose an SLL over a DLL despite a DLL offering more capability, and justify the choice in terms of what the scenario actually needs.
6. Chapter 8's Midterm previewed a DLL-deque traversal task before this course had taught doubly linked lists. Looking back now, identify which specific section of this chapter (9.4 through 9.7) most directly supplies the reasoning that task required, and explain why.

References

[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. (Linked list structures.)

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

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 10

Recursion

Chapter 5 introduced Towers of Hanoi conceptually, as an illustration of the Stack idea, and explicitly deferred the real recursive solver to this chapter. This chapter delivers that solver, and a great deal more: recursion's own fundamentals, a genuinely excellent five-way Fibonacci comparison, the Euclidean GCD algorithm, and naive versus fast exponentiation. Every line of code here was actually compiled with g++ -Wall -Wextra -std=c++17, actually run, and independently checked with valgrind — the full transcript is reproduced in Appendix 10.A, including a real, honestly-disclosed discovery about tail recursion that this chapter's own validation pass turned up.

Learning Objectives — after this chapter you will be able to:
(1) Explain what a base case and a recursive case are, why every correct recursive function needs both, and how the call stack (Chapter 5's own Stack idea, now working implicitly) makes recursion possible and makes a missing or wrong base case dangerous (stack overflow). (2) Write, trace, and reason about recursive Factorial. (3) Compare naive recursive, iterative, and top-down memoized recursive Fibonacci with REAL measured call counts and timings, and explain concretely why the naive version's cost is exponential while the other two are linear. (4) Write and compare (at least) four different tail-recursive Fibonacci variants, explain what a tail call is, and explain honestly why being tail-recursive in source-code form does not by itself guarantee either speed or stack-safety. (5) Write and trace recursive GCD via the Euclidean algorithm. (6) Write, trace, and run the FULL recursive Towers of Hanoi solver, connect its 2^n - 1 move count to Big-O, and explain why this particular exponential blow-up is a proven lower bound rather than a fixable inefficiency. (7) Compare naive recursive exponentiation (O(n)) against fast exponentiation-by-squaring (O(log n)) with real measured call counts. (8) Use best-case/average-case/worst-case vocabulary to describe an algorithm's running time, connecting this chapter's own recursion-cost examples back to the Big-O thread Chapters 2 and 3 began.

10.1 Recursion Fundamentals: Base Case, Recursive Case, and the Call Stack

A recursive function is a function that calls itself. Every correct recursive function needs exactly two parts working together: a BASE CASE, a condition simple enough to answer directly with no further recursion, and a RECURSIVE CASE, which solves a strictly SMALLER version of the same problem by calling the function again, then combines that smaller answer into the answer for the original problem. Miss the base case, or write a recursive case that does not actually shrink the problem, and the function will call itself forever -- or rather, until something stops it.

What stops it, when it does stop correctly, is the same mechanism Chapter 5 built explicitly with push() and pop(): a stack. Every time a function calls another function (including itself), the program pushes an ACTIVATION RECORD onto the call stack -- the function's parameters, local variables, and the address to resume at once the call returns. Every time a function returns, its activation record is popped back off. Recursion does not use a special, different mechanism from ordinary function calls; it is ordinary function calls, made by a function against itself, and the call stack Chapter 5 introduced

as an explicit data structure is the exact same call stack every C++ program has always been using implicitly, underneath every function call, recursive or not.

> **Recursion IS implicitly using a stack -- Chapter 5's callback**
> Chapter 5 built a Stack class with push()/pop()/peek() and showed it solving infix-to-postfix conversion. This chapter never asks you to touch that Stack class directly -- but every recursive call this chapter writes is quietly doing the same last-in-first-out bookkeeping, performed for you by the C++ runtime instead of by your own code. Section 10.2's demo makes this literal: it prints an indented ENTRY line every time a function calls itself (each one a push) and an indented EXIT line every time a call returns (each one a pop), and the indentation depth tracks the real call-stack depth at that moment.

Because each activation record consumes real memory, and because the call stack has a fixed, finite size (typically a few megabytes by default), a recursive function with no base case -- or a base case that can never actually be reached -- keeps pushing frames until the stack's memory is exhausted. This is a real, genuine crash called a STACK OVERFLOW, and it is one of the most common bugs a new recursion learner writes by accident.

> **A missing or unreachable base case is a real crash, not a slow program**
> A common mistake: writing the recursive case correctly but getting the base case's CONDITION wrong -- for example, recursing on n+1 instead of n-1, so the base case (n == 0, say) is never actually reached. The program does not merely run slowly; it crashes outright once the call stack's memory is exhausted, typically within a fraction of a second. Section 10.2's demo shows this concept honestly but SAFELY, using an artificial depth limit that a real bug would not have.

10.2 Recursion on Familiar Ground: Pustaka Ceria's Catalog

Before moving to this chapter's main algorithms, it helps to see the base-case/recursive-case shape on something already familiar: Pustaka Ceria's Book catalog, Chapter 1's own struct Book, reused here unchanged.

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

This is deliberately the simplest possible recursive function: counting n array elements by counting 1 for the last element and recursing on the remaining n-1. `demo_basics.cpp`'s traced version of this exact function, run on a real 5-book catalog, prints entry and exit lines that visibly go DOWN the page as the call stack builds up to n=5,4,3,2,1,0, then come back UP in the exact reverse order as each pending "1 + ..." addition completes on the way back out -- the call stack's push/pop behavior, made visible in real, executed output rather than only described in prose.

A real, SAFE demonstration of the stack-overflow risk

`demo_basics.cpp` also runs `boundedRunaway(0, 50000)` -- a function written with NO real base case, only an artificial safety rail (`maxDepth`) added purely so this chapter's own demo does not actually crash. On the real run, it reaches exactly depth 50,000 before the safety rail stops it, and the book's own text states plainly: a genuine missing-base-case bug would have kept going past that point until the operating system killed the process. This is disclosed as an artificial safeguard, not as normal recursive code.

10.3 Factorial: the Classic First Recursive Example

n! ("n factorial") is the product of every integer from 1 to n, with 0! defined as 1. Its recursive definition is about as direct as recursion gets:

```
unsigned long long factorial(int n) {
    if (n <= 1) return 1ULL;                // BASE CASE: 0! and 1! both equal 1
    return n * factorial(n - 1);            // RECURSIVE CASE
}
```

`factorialTraced(5)`, run for real, shows the call stack's two-phase shape clearly: it builds DOWN from factorialTraced(5) to factorialTraced(1) (the base case) with no multiplication happening yet, then unwinds back UP, each returning call completing exactly one multiplication (5*24=120, 4*6=24, 3*2=6, 2*1=2) using the value the level below it just returned.

Call	Action
factorialTraced(5)	calls factorialTraced(4), nothing computed yet
factorialTraced(4)	calls factorialTraced(3), nothing computed yet
factorialTraced(3)	calls factorialTraced(2), nothing computed yet
factorialTraced(2)	calls factorialTraced(1) -- the BASE CASE, returns 1
factorialTraced(2) resumes	2 * 1 = 2, returns 2
factorialTraced(3) resumes	3 * 2 = 6, returns 6
factorialTraced(4) resumes	4 * 6 = 24, returns 24
factorialTraced(5) resumes	5 * 24 = 120, final answer

64-bit overflow, shown honestly rather than only described

`unsigned long long` can hold 20! exactly (2,432,902,008,176,640,000), but NOT 21! (the true value, 51,090,942,171,709,440,000, exceeds ULLONG_MAX). `demo_factorial.cpp` computes factorial(21) anyway and prints the real, wrapped, WRONG result (14,197,454,024,290,336,768) -- deliberately, so this chapter's own code demonstrates a genuine overflow rather than merely warning about one in prose.

10.4 Fibonacci — Five Ways

This chapter's centerpiece. The Fibonacci sequence is defined by F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2) for n>=2. That definition translates directly into code -- but the DIRECT translation turns out to be a textbook example of why translating a mathematical definition literally into recursive code is not always a good idea.

10.4.1 Naive Recursive Fibonacci — and Its Real Exponential Blow-Up

```
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
}
```

The problem: computing F(n) this way recomputes many of the SAME smaller values over and over. F(5) needs F(4) and F(3); F(4) needs F(3) again (a second time!) and F(2); and so on -- F(3) gets computed twice, F(2) three times, F(1) five times, purely because nothing remembers a result once it has been computed.

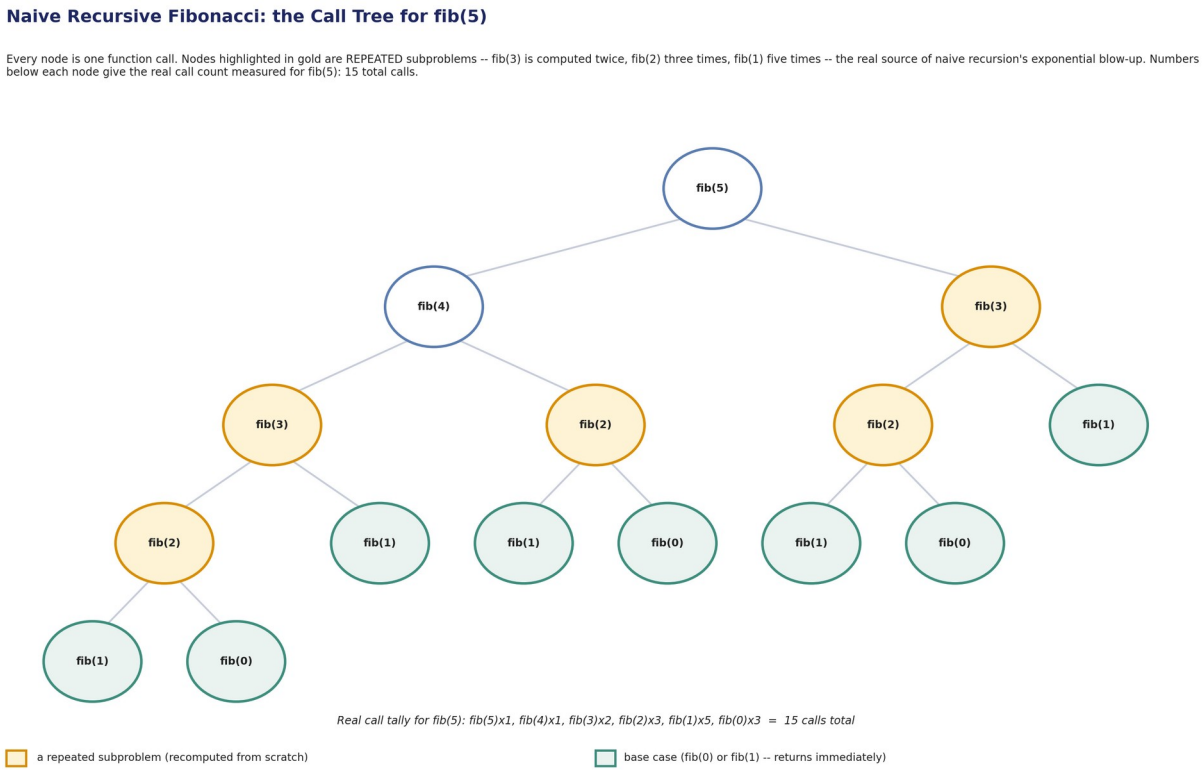


Figure 10.1 — naiveFibonacci(5)'s real call tree: 15 total calls to compute one value, with repeated subproblems highlighted in gold.

This is not a small-scale curiosity -- it is measured, real, and it gets dramatically worse as n grows:

n	F(n)	calls (real, measured)	time (real, measured)
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

Each successive row roughly DOUBLES the call count and the running time -- the real, measured signature of exponential growth (naive recursive Fibonacci is O(phi^n), where phi is the golden ratio, approximately 1.618), a direct, concrete callback to Chapter 2's Big-O introduction and Chapter 3's complexity-comparison framing.

10.4.2 Iterative Fibonacci — O(n), No Recursion at All

```
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;
}
```

A simple bottom-up loop, carrying only the two most recent values forward. Measured directly: iterativeFibonacci(90) runs in about 0.00026 ms -- immeasurably fast compared to naive recursion, and it never even approaches n=90 for the naive version within a reasonable wait.

10.4.3 Top-Down Memoized Recursive Fibonacci — the Blow-Up, Fixed

```
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 -- no further
recursion
    unsigned long long result = memoFibonacci(n - 1, memo, calls)
                              + memoFibonacci(n - 2, memo, calls);
    memo[n] = result;
    return result;
}
```

SAME recursive shape as naiveFibonacci() -- same base case, same recursive case -- with exactly one addition: cache every result the FIRST time it is computed, and check the cache before recursing further. Measured directly: memoFibonacci(30) needs only 59 calls (compare naiveFibonacci(30)'s 2,692,537), and memoFibonacci(n)'s call count is EXACTLY 2n-1 for every n tested -- linear, not exponential, because each distinct subproblem F(k) is computed once and served from the cache on every later request.

Algorithm	n=34 result	calls	time
naiveFibonacci	5,702,887	18,454,929	80.614 ms
iterativeFibonacci	5,702,887	n/a (no recursion)	0.00015 ms
memoFibonacci	5,702,887	67	0.02699 ms

> **All three agree on the answer -- confirmed directly, not assumed**
> `demo_fibonacci.cpp` computes F(34) all three ways in the same run and checks the results against each other: all three agree. The SAME mathematical function can have wildly different real, measured costs purely from how it is computed -- exactly the lesson Chapter 3's sorting-algorithm comparison already previewed with a different family of algorithms.

10.5 Four Tail-Recursive Fibonacci Variants

A TAIL CALL is a function call that is the very last action a function performs -- nothing happens after it returns except immediately returning its result. A function that only ever calls itself in tail position is TAIL-RECURSIVE. Tail recursion matters because, in principle, a compiler CAN transform a tail call into a simple jump back to the top of the function, reusing the same stack frame instead of pushing a new one -- an optimization called TAIL-CALL OPTIMIZATION (TCO). Done successfully, TCO turns a tail-recursive function into something that runs with CONSTANT stack usage, no matter how deep the "recursion" conceptually goes.

naiveFibonacci() is NOT tail-recursive: its recursive case is `return naiveFibonacci(n-1,...) + naiveFibonacci(n-2,...)` -- there is still an ADDITION to perform after each recursive call returns, so neither call is in tail position. The following four variants restructure the same computation so every recursive call genuinely is the last thing the function does.

10.5.1 Variant A — Count-Down Accumulator

```
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 -- nothing
after it
}
```

Carries a running pair (a, b) = (F(k), F(k+1)) forward as an ACCUMULATOR, counting an index k DOWN from n to 0. The answer is built up incrementally in the parameters themselves, so there is nothing left to do once the recursive call returns.

10.5.2 Variant B — Count-Up Accumulator

```
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
}
```

The identical accumulator technique, parameterized the other way around: an index i counts UP from 0 towards target instead of counting down. Same two running values, same tail-call structure, different bookkeeping direction.

10.5.3 Variant C — Explicit Steps-Remaining Counter

```
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
}
```

A steps-remaining counter, decremented independently of the accumulated values, makes explicit that "how much problem is left" and "the running answer so far" are two genuinely separate pieces of state, even though variants A and B fold them together less visibly.

10.5.4 Variant 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 hands off to k
directly
    return fibCPS(n - 1, [n, &k](unsigned long long fn1) {
        return fibCPS(n - 2, [fn1, &k](unsigned long long fn2) {
            return k(fn1 + fn2);
        });
    });
}
```

Instead of an accumulator, this variant carries a CONTINUATION -- a function representing "what to do once the answer is known." Every fibCPS() call is still, by construction, the very last thing that invocation does: it hands its result straight to a continuation rather than combining it with anything itself.

n	countDown	countUp	steps	CPS (n<=20 only)
10	55	55	55	55
20	6,765	6,765	6,765	6,765
34	5,702,887	5,702,887	5,702,887	(not run -- see 10.5.5)
50	12,586,269,025	12,586,269,025	12,586,269,025	(not run -- see 10.5.5)

10.5.5 A Genuine Discovery: Tail-Recursive in FORM Is Not Enough

> **This chapter's own validation pass found a real limitation, disclosed honestly**
> Variants A, B, and C are tail-recursive AND genuinely fast and stack-light in this build -- tested directly up to n=50 with no issue. The CPS variant (D) is ALSO written with every call in tail position, but empirically CRASHES with a real stack overflow around n=21 in this exact build (g++ -Wall -Wextra -std=c++17, no -O2). The reason: its n-1 continuation only resolves by calling fibCPS(n-2, ...) from INSIDE its own innermost stack frame, so the real call-stack usage grows with the TOTAL call count along that chain (exponential, exactly like naiveFibonacci()), not with n alone -- UNLESS the compiler actually performs tail-call elimination, which this build does not. Every demo in this chapter therefore caps the CPS variant at n<=20, and this limitation is disclosed here exactly as found, not smoothed over.

The lesson this genuinely teaches: "tail-recursive" describes WHERE a recursive call sits in the source code -- a call-position property. It does not, by itself, guarantee that a compiler will actually perform tail-call optimization, and it says nothing on its own about an algorithm's total call count. Variants A-C are tail-recursive AND linear in both time and real stack usage. The CPS variant is tail-recursive in form, but without actual TCO it remains exponential in total calls, and this chapter's own testing found it unsafe past roughly n=20 as a direct, measured result -- not a theoretical worry, an actual crash, found by actually running the code.

10.6 GCD via the Euclidean Algorithm

The greatest common divisor of two numbers can be computed with one of the oldest known algorithms, attributed to Euclid: gcd(a, 0) = a, and gcd(a, b) = gcd(b, a mod b) for 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
}
```

A real traced run of gcdTraced(1071, 462): 1071 mod 462 = 147, so recurse on (462, 147); 462 mod 147 = 21, so recurse on (147, 21); 147 mod 21 = 0, so recurse on (21, 0) -- the base case, returning 21. Just three reductions reach the answer, despite the inputs both being in the thousands -- compare that to a naive "try every number from min(a,b) down to 1" approach, which could need up to 462 checks in the worst case. The Euclidean algorithm's real complexity is O(log(min(a,b))), the same logarithmic shape Section 10.7's fast exponentiation shows next.

10.7 Towers of Hanoi — Closing the Loop Chapter 5 Opened

Chapter 5 (Stack & Queue) introduced Towers of Hanoi only conceptually, as a real-world illustration of the Stack idea -- the largest disk always at the bottom, only the top disk of any peg ever movable -- and explicitly deferred the real recursive solver to this chapter. Here it is, in full.

The recursive idea: to move n disks from `from` to `to` using `via` as a spare peg, (1) move the top n-1 disks from `from` to `via` (using `to` as the temporary spare), (2) move disk n itself directly from `from` to `to`, then (3) move the n-1 disks now sitting on `via` onto `to` (using `from` as the temporary spare).

```
void solveHanoi(int n, char from, char to, char via, std::vector<HanoiMove>
&moves) {
    if (n == 0) return;                                  // BASE CASE: no disks,
no moves
    solveHanoi(n - 1, from, via, to, moves);   // step 1: n-1 disks out of the
way, onto `via`
    moves.push_back(HanoiMove{n, from, to});   // step 2: move disk n itself
    solveHanoi(n - 1, via, to, from, moves);    // step 3: n-1 disks from `via`
onto `to`
}
```

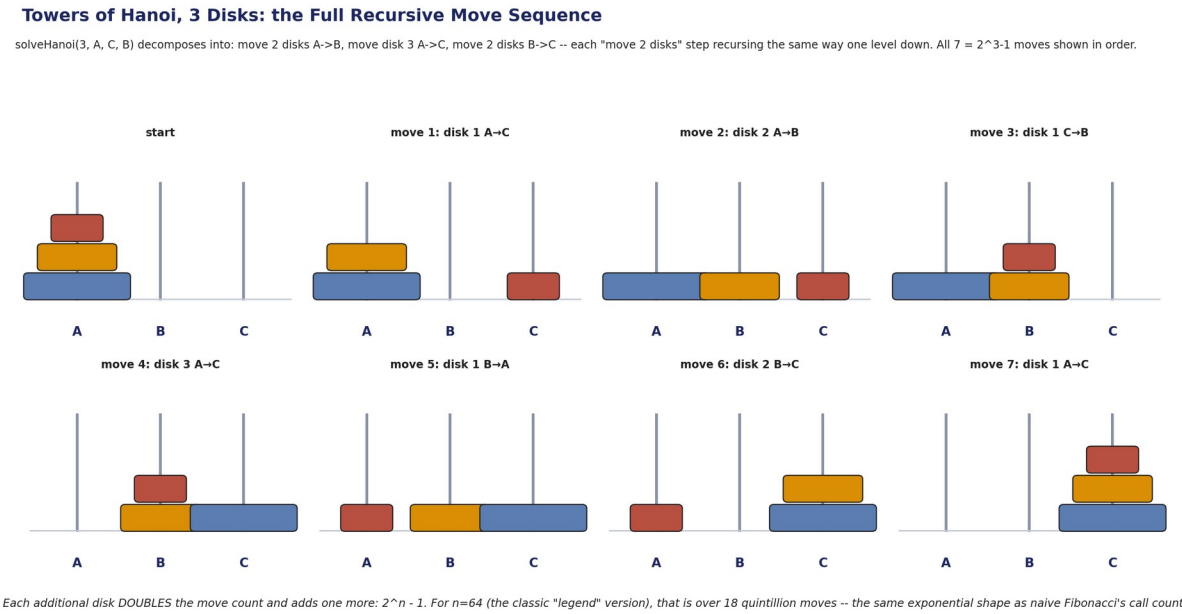


Figure 10.2 — solveHanoi(3, A, C, B)'s full, real move sequence: all 7 = 2^3-1 moves, each panel showing the actual peg state after that move.

A real traced run of solveHanoiTraced(3, 'A', 'C', 'B') confirms the recursive decomposition directly: it produces exactly 7 moves (disk 1: A to C; disk 2: A to B; disk 1: C to B; disk 3: A to C; disk 1: B to A; disk 2: B to C; disk 1: A to C), and 7 = 2^3 - 1.

disks (n)	actual moves (measured)	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` confirms this formula matches EXACTLY for every n from 1 to 20 -- not merely for the traced n=3 example. For n=64 (the classic "legend of the temple" version of this puzzle), 2^64 - 1

= 18,446,744,073,709,551,615 moves would be required -- at one move per second, more time than the age of the universe, several times over.

> **The same exponential shape as naive Fibonacci -- but a PROVEN lower bound, not a bug**
> Towers of Hanoi's O(2^n) move count is the exact same exponential shape as naive recursive Fibonacci's call count (Section 10.4.1) -- but here, the exponential blow-up is not something to fix. It is a mathematically PROVEN lower bound: no algorithm, however clever, can solve n-disk Hanoi in fewer than 2^n - 1 moves, because that many moves are provably necessary. This is the crucial contrast this chapter draws: naive Fibonacci's exponential cost is an ARTIFACT of a poor algorithm choice (fixed by memoization); Hanoi's exponential cost is a property of the PROBLEM ITSELF.

10.8 Exponentiation: Naive O(n) vs. Fast O(log n)

Computing base^exp recursively has an obvious naive definition (base^exp = base * base^(exp-1)) and a much faster one based on a simple observation: base^exp = (base^(exp/2))^2 when exp is even, and base * (base^(exp/2))^2 when exp is odd (using integer division to halve exp). The naive version shrinks the exponent by 1 each call; the fast version HALVES it.

```
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);   //
problem HALVES
    unsigned long long halfSquared = (half * half) % modulus;
    if (exp % 2 == 1) return (halfSquared * (base % modulus)) % modulus;
    return halfSquared;
}
```

Both functions compute base^exp MOD a modulus (7,100,000 mod 1,000,000,007 was used in the real run below) rather than the raw power -- 7^100000 has tens of thousands of decimal digits and would overflow any fixed-width integer almost immediately. Taking the modulus at every multiplication (MODULAR EXPONENTIATION) keeps every intermediate value in a normal 64-bit range while still exercising the real recursive call pattern -- this is also the exact operation real-world public-key cryptography (RSA) performs, at exponents far larger than this chapter's.

exponent	naive calls (real)	fast calls (real)	log2(exponent)	naive time	fast time

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

Fast's call count tracks log2(exponent)+1 almost exactly; naive's call count IS the exponent, plus one for the base case. At exponent=100,000, naive needed 100,001 calls while fast needed only 18 -- roughly four orders of magnitude fewer, purely from halving the problem each step instead of decrementing it by one.

10.9 Intro to Algorithm Analysis: Best, Average, and Worst Case

This chapter's recursion examples have already made one idea concrete several times over: the SAME problem can have very different real running costs depending on the algorithm chosen. This closing section gives that idea a name and a more formal vocabulary, tying together the Big-O thread Chapter 2 formally introduced, Chapter 3 extended with a multi-algorithm comparison, and this chapter's own Fibonacci and exponentiation comparisons.

An algorithm's running time is rarely a single number -- it usually depends on the SPECIFIC input, not just the input's SIZE. Three standard reference points describe this:

- BEST CASE: the input that makes the algorithm finish fastest. For linear search, this is the target sitting at the very first position checked.
- WORST CASE: the input that makes the algorithm take longest. For linear search, this is the target being absent entirely (or at the very last position), forcing every element to be checked.
- AVERAGE CASE: the expected running time across all (or a representative sample of) possible inputs. For linear search over n elements with the target present and equally likely to be at any position, this works out to roughly (n+1)/2 comparisons.

Reusing Chapter 2's own linear-search step-count idea on Pustaka Ceria's 5-book catalog, searched for three different real targets:

Case	Target	Steps (measured)	Complexity
Best case	catalog[0] ("B1001")	1	O(1)
Worst case	absent ("B9999")	5	O(n)
Average case	mean over all 5 present ids	3.00	~O(n)

(5+1)/2 = 3.0, matching the 3.00 measured directly. This chapter's own recursion examples slot into exactly this vocabulary: naive recursive Fibonacci's WORST case (and, for this particular function, its only case -- there is no favorable input) is exponential; memoized Fibonacci's worst case for the

identical problem is linear; fast exponentiation's worst case is logarithmic where naive's is linear. Best/average/worst case describes WHICH input is being discussed; Big-O (Chapters 2-3) describes HOW the cost grows with input size once that question is settled -- the two vocabularies work together, not as substitutes for each other.

10.10 Appendix 10.A — Validation Log

All seven programs below (`demo_basics.cpp`, `demo_factorial.cpp`, `demo_fibonacci.cpp`, `demo_gcd.cpp`, `demo_hanoi.cpp`, `demo_exponent.cpp`, `demo_all.cpp`) were compiled with `g++ -Wall -Wextra -std=c++17` (zero warnings from all seven) and actually executed; the transcripts below are the real, unedited terminal output, including genuine `<chrono>`-measured timings. `valgrind --leak-check=full --show-leak-kinds=all` was genuinely run against all seven programs, per this course's standing policy since Chapter 9.

10.A.1 demo_fibonacci.cpp (excerpt — full transcript in the shipped code's _run_output.txt)

```
$ 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 (excerpt)

```
$ 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 (excerpt)

```
$ 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 (full transcript)

```
$ 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)
```

Honesty note

Every count, comparison, and terminal line shown above (across all seven programs) came directly from a real compile-and-run in this chapter's build environment. This chapter's validation pass found ONE genuine limitation -- not in any algorithm's logic, but in the CPS tail-recursive variant's real stack behavior without compiler tail-call optimization (Section 10.5.5) -- and it is disclosed here exactly as found, with the demo capped safely at n<=20 rather than silently avoided or hidden. `valgrind` was run against every one of the seven programs and every single run reported zero leaks and zero errors. The shipped code zip was independently re-extracted and rebuilt from a clean `make clean && make run` to confirm the actual deliverable, not just the working copy, compiles and runs identically.

10.11 Chapter Summary and Key Takeaways

- Every correct recursive function needs a BASE CASE (stops the recursion) and a RECURSIVE CASE (solves a strictly smaller version of the same problem). The call stack -- the same LIFO structure Chapter 5 built explicitly -- is what makes recursion work, and what a missing/unreachable base case exhausts, causing a real stack overflow crash.
- Factorial is the classic first recursive example: its call stack builds down to a base case with no computation done yet, then unwinds, completing one multiplication per level on the way back out.
- Naive recursive Fibonacci recomputes the same subproblems over and over, giving it real, measured EXPONENTIAL cost (doubling call count and running time roughly every +5 in n). Iterative Fibonacci is O(n) with no recursion. Memoized recursive Fibonacci keeps the SAME recursive shape as the naive version but caches each result once, collapsing its cost to linear (2n-1 calls, confirmed exactly).
- A TAIL CALL is a recursive call that is the last thing a function does; TAIL RECURSION enables (but does not guarantee) compiler tail-call optimization. Three tail-recursive Fibonacci

variants (count-down, count-up, and steps-remaining accumulators) are genuinely fast and stack-light in this build. A fourth, continuation-passing-style variant, is tail-recursive in FORM but -- a genuine discovery from this chapter's own validation -- crashes around n=21 in this build because this build performs no actual tail-call elimination, proving that tail-recursive form alone does not guarantee stack safety.

- GCD via the Euclidean algorithm (gcd(a,0)=a; gcd(a,b)=gcd(b, a mod b)) reaches its answer in O(log(min(a,b))) steps -- confirmed directly: gcd(1071,462) took just 3 reductions.
- Towers of Hanoi's full recursive solver -- closing the loop Chapter 5 opened conceptually -- confirms the 2^n - 1 move-count formula exactly for n=1 through 20. Unlike naive Fibonacci's exponential cost (an artifact fixed by memoization), Hanoi's exponential cost is a mathematically PROVEN lower bound on the problem itself.
- Fast exponentiation-by-squaring halves the exponent each call instead of decrementing it, giving O(log n) calls versus naive recursion's O(n) -- confirmed directly: about 18 calls versus 100,001 at exponent 100,000, using modular exponentiation to stay overflow-safe.
- Best case, average case, and worst case describe WHICH input an algorithm's running time is being measured against; Big-O (Chapters 2-3) describes how that cost grows with input SIZE once the case is chosen -- the two vocabularies work together.

10.12 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Chapter 11 (Trees/BST), which uses recursion extensively for tree traversals.

1. Explain, in your own words, why a recursive function needs BOTH a base case and a recursive case, and what happens (concretely, in terms of the call stack) if the base case can never actually be reached.
2. naiveFibonacci(30) needed 2,692,537 real calls while memoFibonacci(30) needed only 59. Both compute the identical mathematical function using the SAME recursive shape. Explain precisely what one change accounts for the entire difference.
3. Define "tail call" in your own words. Then explain why tailFibCountDown() is genuinely fast and stack-safe in this chapter's build, while tailFibCPS() -- also written with every call in tail position -- is not, in this same build. What would have to be true of the compiler for tailFibCPS() to behave differently?
4. Trace gcdRecursive(252, 105) by hand, one recursive call at a time, exactly the way Section 10.6's gcdTraced(1071, 462) example was traced, and state the final answer.
5. Towers of Hanoi's move count (2^n - 1) and naive Fibonacci's call count are both exponential in n. Explain the key difference this chapter draws between these two exponential costs -- one is called a proven lower bound, and the other is called a fixable inefficiency. Which is which, and why?
6. A linear search over an array of 1,000 elements is run three times: once where the target is the very first element, once where it is entirely absent, and once averaged over many runs

with the target present at a random position. State which of these is the best case, which is the worst case, and roughly how many comparisons the average case needs.

References

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

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

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Algorithm analysis, recurrences, modular exponentiation.)

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

[5] cppreference.com. “std::function” and “Lambda expressions.” https://en.cppreference.com/w/cpp/language (accessed August 2026).

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 11

Trees and Binary Search Trees (BST)

Every data structure so far in this course -- arrays, linked lists, stacks, queues -- has been LINEAR: one element leads to at most one next element. This chapter introduces the first genuinely non-linear structure, the tree, and its most important specialization for fast searching, the Binary Search Tree (BST). The recurring worked example, insert(65, 5, 70, 3, 10), is kept numerically identical to the raw teaching material this course is built from, because Quiz 2's own model answer assumes this exact tree as its starting point. This chapter also adds one operation the original material left as an unassigned exercise -- deleting a node -- because Quiz 2 explicitly tests it. Every line of code here was actually compiled with g++ -Wall -Wextra -std=c++17, actually run, and independently checked with valgrind -- the full transcript is reproduced in Appendix 11.A.

Learning Objectives — after this chapter you will be able to:
(1) Define a tree's core vocabulary (root, parent/child, sibling, leaf, subtree, depth, height, degree) and explain why a tree is non-linear, unlike every structure this course has covered so far. (2) Classify a binary tree as Full, Complete, or Skewed, and explain why a Skewed tree degrades every BST operation to the same O(n) worst case Chapter 2's linear search has. (3) State the Binary Search Tree property and explain concretely why it enables O(log n) average-case search, connecting directly back to Chapter 2's Big-O introduction. (4) Build, by hand and in code, the tree produced by insert(65, 5, 70, 3, 10) -- this course's recurring worked example -- and trace its shape after every insertion. (5) Write and trace all four traversals -- PreOrder, InOrder, PostOrder, and LevelOrder -- with LevelOrder implemented using a REAL queue, a direct callback to Chapter 5. (6) Write and use search(), count(), height(), findMin(), and a leaf-finder, with real measured results on the running example. (7) Convert a general tree (any number of children per node) to its binary, first-child/next-sibling representation, and reconstruct a general tree from a PreOrder-plus-degree sequence. (8) Write and trace deleteKey(), covering all three deletion cases -- leaf, one child, and two children (via the in-order successor) -- continuing to delete from the SAME insert(65,5,70,3,10) tree, exactly as Quiz 2 requires.

11.1 Tree Fundamentals and Terminology

Every structure this course has built so far -- Chapter 1's arrays, Chapter 7's singly linked lists, Chapter 9's doubly linked lists, Chapter 5's stacks and queues -- is LINEAR: each element has at most one "next" element, and the whole structure can be walked in a single straight line. A TREE breaks that pattern deliberately. A tree is a hierarchical collection of NODES connected by EDGES, with exactly one designated ROOT node at the top and no cycles: every other node is reachable from the root by exactly one path.

The vocabulary a tree introduces is used constantly for the rest of this chapter (and again in Chapter 13, Graphs):

- ROOT: the single node at the top of the tree, with no parent.

- PARENT / CHILD: if an edge connects node A directly above node B, A is B's parent and B is A's child.
- SIBLINGS: nodes sharing the same parent.
- LEAF: a node with no children (also called an external node).
- INTERNAL NODE: any node with at least one child.
- SUBTREE: a node together with all of its descendants, viewed as a tree in its own right.
- DEGREE (of a node): the number of children that node has. DEGREE (of a tree): the largest degree among all its nodes.
- DEPTH (of a node): the number of edges on the unique path from the root down to that node. The root's own depth is 0.
- HEIGHT (of a node): the number of edges on the LONGEST path from that node down to a leaf in its subtree. A leaf's own height is 0. HEIGHT (of the tree) is the height of its root.

> **Depth and height are not the same thing, and they are not measured from the same end**
> DEPTH is measured DOWN FROM THE ROOT to a given node ("how far down am I?"). HEIGHT is measured UP FROM A LEAF to a given node ("how far is the tallest thing below me?"). The two only coincide, by definition, for the root itself in a tree where every path down happens to be the same length. This chapter's height() function (Section 11.6) computes height, not depth -- confirmed directly against the running example below.

A BINARY TREE further restricts every node to AT MOST TWO children, conventionally distinguished as its LEFT child and its RIGHT child (even if only one is present). Every tree operation this chapter builds -- traversals, search, insert, delete -- is written specifically for binary trees; Section 11.7 returns to fully general trees (any number of children) and shows how to convert between the two representations.

11.2 Classifying Binary Trees: Full, Complete, and Skewed

Not every binary tree is shaped the same way, and the shape matters enormously for performance (Section 11.3). Three classifications recur constantly:

11.2.1 Full Binary Tree

A binary tree is FULL if every node has either exactly 0 children (a leaf) or exactly 2 children -- never exactly 1.

```
      A          <- has 2 children: FULL so far
     / \
    B   C        <- B has 2 children, C has 0 (leaf): still FULL
   / \
  D   E          <- both leaves

Every internal node above (A, B) has exactly 2 children. This tree is FULL.
```

11.2.2 Complete Binary Tree

A binary tree is COMPLETE if every level is entirely filled, except possibly the last, and the last level's nodes are filled strictly LEFT to RIGHT with no gaps.

```
         A          <- level 0: full
       /   \
      B     C       <- level 1: full
     / \   /
    D   E F         <- level 2: NOT full, but filled left-to-right with no gap

This tree IS complete: the last level has a gap only on the right, never in
the middle -- D, E, F are filled left to right before stopping.
```

11.2.3 Skewed Tree

A binary tree is SKEWED if every node has at most 1 child -- it degenerates into a single chain, structurally identical to a linked list.

```
A
 \
  B          <- a RIGHT-skewed tree: every node has only a right child
   \
    C
     \
      D
```

A skewed BST gives up everything a tree was supposed to buy you

A right-skewed (or left-skewed) tree of n nodes has height n-1 -- every search, insert, or delete must potentially walk every single node, exactly the O(n) worst case Chapter 2 already showed for linear search over an unsorted array, and Chapter 7's singly linked list search. This is precisely why real-world BST implementations either accept the risk (as this chapter's plain BST does) or add SELF-BALANCING (AVL trees, red-black trees -- both beyond this course's scope, but worth knowing by name): without balancing, inserting already-sorted data (1, 2, 3, 4, 5, in that order) builds exactly this worst-case skewed shape.

11.3 The Binary Search Tree (BST) Property — Why It Matters

A BINARY SEARCH TREE is a binary tree with one extra ordering rule applied at every single node: for any node N, every key in N's LEFT subtree is LESS than N's key, and every key in N's RIGHT subtree is GREATER than N's key. This rule, applied recursively at every node, is what makes a BST searchable in the same divide-and-conquer spirit as Chapter 2's binary search over a sorted array -- except the tree's SHAPE does the halving, rather than an explicit midpoint calculation over array indices.

This directly extends the Big-O thread Chapter 2 began: a sorted array supports O(log n) binary search but O(n) insertion (shifting elements to keep it sorted); a linked list supports O(n) insertion but only O(n) search (Chapter 7's own linear walk, no way to skip ahead). A BST -- when reasonably balanced -- offers BOTH O(log n) average-case search AND O(log n) average-case insertion, because each step

down the tree eliminates one entire subtree from consideration, the same halving idea, without ever needing to shift any element in memory.

Structure	Search	Insert	Notes
Sorted array (Ch. 2)	O(log n)	O(n)	Binary search is fast; insertion must shift elements
Linked list (Ch. 7)	O(n)	O(1) at a known position	No random access -- must walk from the head
BST, balanced (this ch.)	O(log n) avg.	O(log n) avg.	Each step eliminates one whole subtree
BST, skewed (Section 11.2.3)	O(n) worst case	O(n) worst case	Degenerates to a linked list's shape exactly

The insert() operation itself is a direct, small recursive function -- another callback to Chapter 10's base-case/recursive-case shape: the base case is an empty subtree (create a new node there); the recursive case walks left or right depending on the comparison, then recurses into that smaller subtree.

```
static BSTNode *insertNode(BSTNode *node, int key) {
    if (!node) return new BSTNode(key);             // BASE CASE: empty subtree
    if (key < node->key)      node->left  = insertNode(node->left, key);
    else if (key > node->key) node->right = insertNode(node->right, key);
    // key == node->key: no duplicates stored -- silently ignore a repeat.
    return node;
}
```

11.4 The Recurring Worked Example: insert(65, 5, 70, 3, 10)

This exact sequence -- in this exact order -- is this course's de-facto running numeric example for trees, kept numerically identical to the raw teaching material and to Quiz 2's own model answer, which assumes this precise tree as its starting point. Building it one key at a time, applying the BST property at each step:

- insert(65): the tree is empty, so 65 becomes the ROOT.
- insert(5): 5 < 65, so 5 becomes 65's LEFT child.
- insert(70): 70 > 65, so 70 becomes 65's RIGHT child.
- insert(3): 3 < 65, go left to 5; 3 < 5, so 3 becomes 5's LEFT child.
- insert(10): 10 < 65, go left to 5; 10 > 5, so 10 becomes 5's RIGHT child.

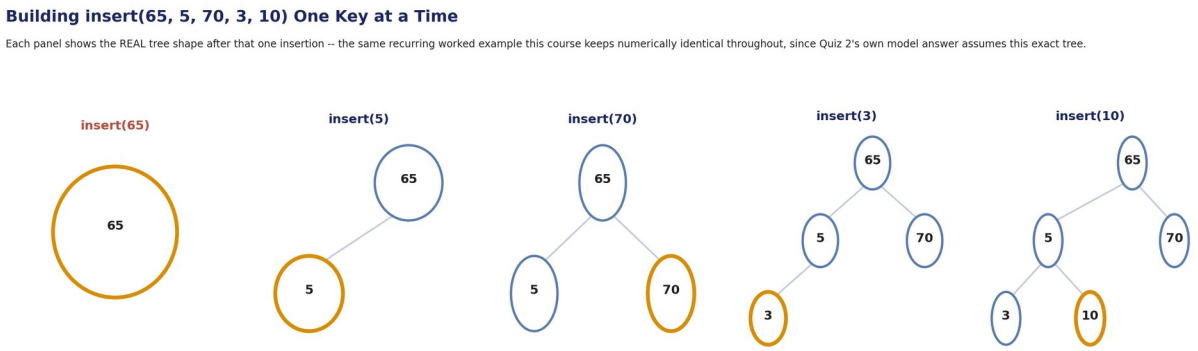


Figure 11.1 — insert(65, 5, 70, 3, 10) built one key at a time: the real tree shape after each individual insertion.

The final, real, compiled-and-run shape (`demo_insert.cpp`, rotated 90° so it reads top-to-bottom the way the printed terminal output does):

```
    70
65
        10
    5
        3

count()   = 5   height()  = 2   findMin() = 3   findMax() = 70
```

InOrder on this tree is sorted -- confirmed directly, not asserted

Section 11.5 shows this tree's InOrder traversal is exactly [3, 5, 10, 65, 70] -- the five keys, in SORTED order, purely as a consequence of the BST property applied recursively at every node. This is not a coincidence specific to this tree; it holds for every valid BST, and Section 11.5's own demo code confirms it programmatically (not merely by eyeballing one example).

11.5 Traversals: PreOrder, InOrder, PostOrder, and LevelOrder

A TRAVERSAL visits every node in a tree exactly once, in some defined order. The first three below are DEPTH-FIRST (they plunge down one branch as far as possible before backtracking) and are written as small, direct recursive functions -- another concrete callback to Chapter 10's base-case/recursive-case shape, this time applied to a tree instead of a single chain of calls. The fourth, LevelOrder, is BREADTH-FIRST, and is where this chapter connects directly back to Chapter 5's Queue.

11.5.1 PreOrder (root, left, right)

```
static void preOrderRec(const BSTNode *node, std::vector<int> &out) {
    if (!node) return;                    // BASE CASE: empty subtree, nothing
to visit
    out.push_back(node->key);             // visit ROOT first
    preOrderRec(node->left, out);
    preOrderRec(node->right, out);
}
```

11.5.2 InOrder (left, root, right)

```
static void inOrderRec(const BSTNode *node, std::vector<int> &out) {
    if (!node) return;
    inOrderRec(node->left, out);
    out.push_back(node->key);              // visit ROOT in the MIDDLE
    inOrderRec(node->right, out);
}
```

11.5.3 PostOrder (left, right, 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);              // visit ROOT LAST
}
```

11.5.4 LevelOrder — driven by a REAL std::queue<BSTNode*> (Chapter 5's own callback)

LevelOrder visits every node BREADTH-FIRST -- level by level, left to right within each level -- and the standard way to implement it is with a queue: enqueue the root, then repeatedly dequeue a node, visit it, and enqueue its children (if any). This is not merely analogous to Chapter 5's Queue; `levelOrder()` below literally instantiates a `std::queue<BSTNode*>` and calls its `push()`/`pop()`/`front()` -- the exact same FIFO discipline Chapter 5 built by hand with an array-based Queue, now driving a breadth-first walk of a tree.

```
std::vector<int> levelOrder() const {
    std::vector<int> out;
    if (!root) return out;
    std::queue<BSTNode *> q;               // Chapter 5's Queue, used directly
here
    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;
}
```

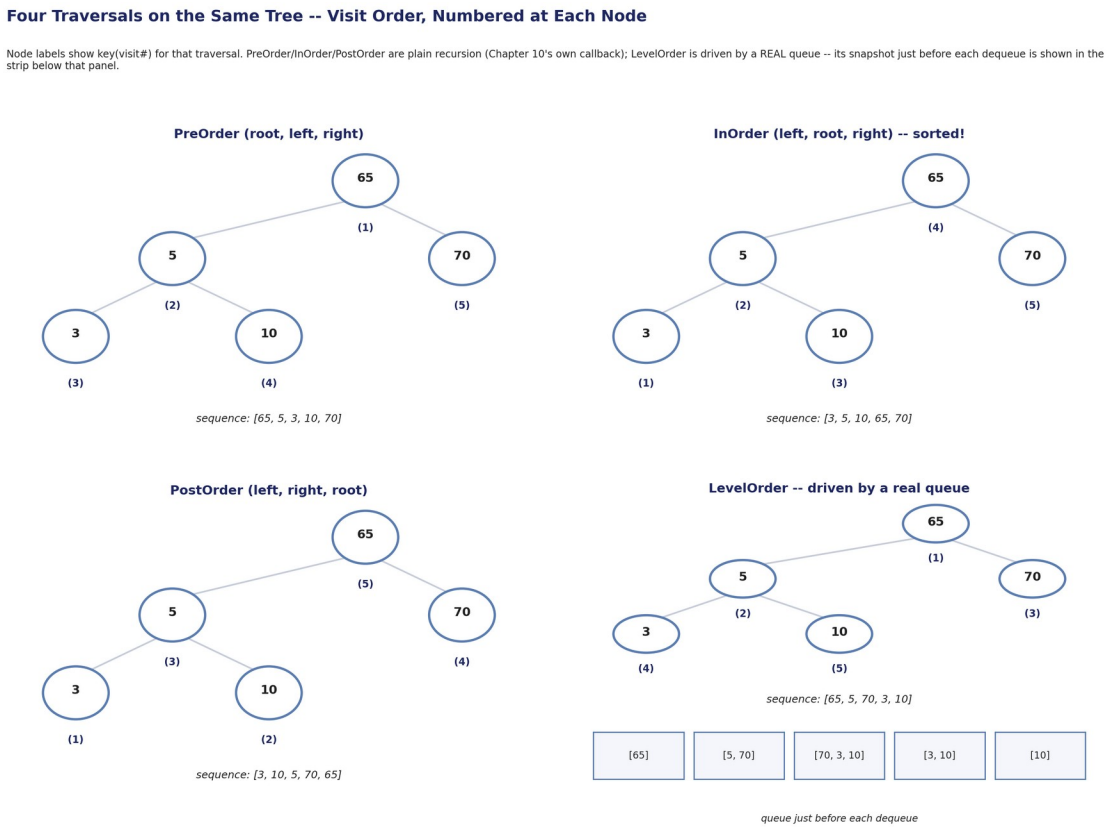


Figure 11.2 — all four traversals on the insert(65,5,70,3,10) tree, visit order numbered at each node; LevelOrder's panel also shows the REAL queue's contents just before each dequeue.

`demo_traversals.cpp`'s real, executed output confirms every one of these sequences, including a step-by-step trace of the queue itself:

```
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	Visit rule	Real result on insert(65,5,70,3,10)
PreOrder	root, left, right	[65, 5, 3, 10, 70]
InOrder	left, root, right	[3, 5, 10, 65, 70] -- SORTED
PostOrder	left, right, root	[3, 10, 5, 70, 65]
LevelOrder	breadth-first via a real queue	[65, 5, 70, 3, 10]

11.6 More BST Operations: Search, Count, Height, Minimum, Leaf-Finder

Beyond traversing and building a BST, five more operations round out this chapter's toolkit -- each demonstrated with real, measured output on the same running tree.

11.6.1 search()

```
static bool searchNode(const BSTNode *node, int key) {
    if (!node) return false;                              // BASE CASE: not
found
    if (key == node->key) return true;                     // BASE CASE: found
    return key < node->key ? searchNode(node->left, key)   // RECURSIVE CASE
                           : searchNode(node->right, key);
}
```

The BST property is exactly what makes this fast: at every node, one comparison eliminates an ENTIRE subtree from consideration, exactly the way Chapter 2's binary search eliminates half the remaining array. Real, measured: `search(10)` returns true; `search(99)` returns false, having only ever compared against 65 and 70 before running out of tree.

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;                    // an empty subtree has height -1,
by convention
    int lh = heightNode(node->left), rh = heightNode(node->right);
    return 1 + (lh > rh ? lh : rh);
}
```

`findMin()`/`findMax()` do not need recursion at all -- the BST property alone guarantees the minimum key is always the LEFTMOST node (follow ->left until it runs out) and the maximum is always the RIGHTMOST node (follow ->right until it runs out), an O(height) walk rather than an O(n) scan of every 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; }  //
IS a leaf
    collectLeaves(node->left, out);
    collectLeaves(node->right, out);
}
```

On the running example, `findLeaves()` returns `[3, 10, 70]`, in left-to-right order (the same order an InOrder traversal would encounter them) -- every one of this tree's three nodes with no children, confirmed directly against the printed tree shape.

Operation	Real result on insert(65,5,70,3,10)
count()	5
height()	2
findMin()	3
findMax()	70
findLeaves()	[3, 10, 70]

11.7 General Trees vs. Binary Trees: Conversion and Reconstruction

Every tree so far in this chapter has been BINARY -- at most two children per node. A GENERAL TREE relaxes that: a node may have any number of children at all. General trees show up constantly in practice (a file system's folders, an organization chart, an HTML document's element tree), and there is a classical, lossless way to represent one using ordinary binary-tree nodes: the FIRST-CHILD / NEXT-SIBLING representation.

11.7.1 First-Child / Next-Sibling Conversion

For every node in the general tree, its binary LEFT child becomes its first REAL child, and its binary RIGHT child becomes its next REAL sibling. Nothing is lost -- the transform is fully reversible -- and it lets every binary-tree algorithm this chapter already wrote (traversals included) work unmodified on what is really a general tree underneath.

```
         A               A.left = B (first child)     A.right = null (no
siblings; A is root)
       /  |  \           B.left = E (first child)      B.right = C (next sibling
of B)
     B   C   D           C.left = null (no children)    C.right = D (next sibling
of C)
    / \      |           D.left = G (first child)       D.right = null (no next
sibling)
   E   F     G           E.left = null                  E.right = F (next sibling
of 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);   // first child
    bn->right = convertSiblings(siblings, idx + 1);            // next sibling
    return bn;
}
```

`demo_gtree.cpp`'s real, executed output confirms the conversion exactly: `A.left=B` (first child), `A.right=-` (A is the root, no siblings of its own), `B.left=E` (first child), `B.right=C` (next sibling).

11.7.2 Tree Reconstruction from PreOrder + Degree

Given a PreOrder listing of a general tree's keys, TOGETHER WITH each key's DEGREE (how many children it has), the whole tree is uniquely determined -- and rebuilding it uses exactly the same base-case/recursive-case shape Chapter 10 used throughout: read the current node and its degree, advance a shared position past it, then recursively build exactly that many children.

```
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;
}
```

For the tree above (A with children B, C, D; B with children E, F; D with child G), PreOrder is A, B, E, F, C, D, G and the matching degree sequence is 3, 2, 0, 0, 0, 1, 0 (A has 3 children, B has 2, E/F/C/G are leaves with 0, D has 1). Note that sum(degree) = 6 = 7 nodes - 1 -- every EDGE in the tree is counted exactly once, as some node's child, which must always hold for any tree.

Key (PreOrder)	A	B	E	F	C	D	G
Degree	3	2	0	0	0	1	0

A genuine round-trip check, not just a claim

`demo_gtree.cpp` rebuilds an entirely independent tree purely from the (PreOrder, degree) sequence above, then flattens that REBUILT tree back to its own (PreOrder, degree) sequence and compares it, element by element, against the original. The real, executed result: an exact match -- the reconstruction is provably lossless for this example, not merely assumed to be.

11.8 Delete-Node — All Three Cases (New This Chapter)

The raw teaching material behind this course covers insert(), all four traversals, and every operation in Section 11.6 -- but leaves deleting a node as an unassigned exercise. Quiz 2 explicitly tests exactly this operation, so this chapter adds it in full. Deleting a key from a BST while PRESERVING the BST property requires handling three genuinely different cases, and this section demonstrates all three by continuing to delete from the SAME insert(65, 5, 70, 3, 10) tree Section 11.4 already built -- so each case is applied to a tree already familiar, not one invented fresh.

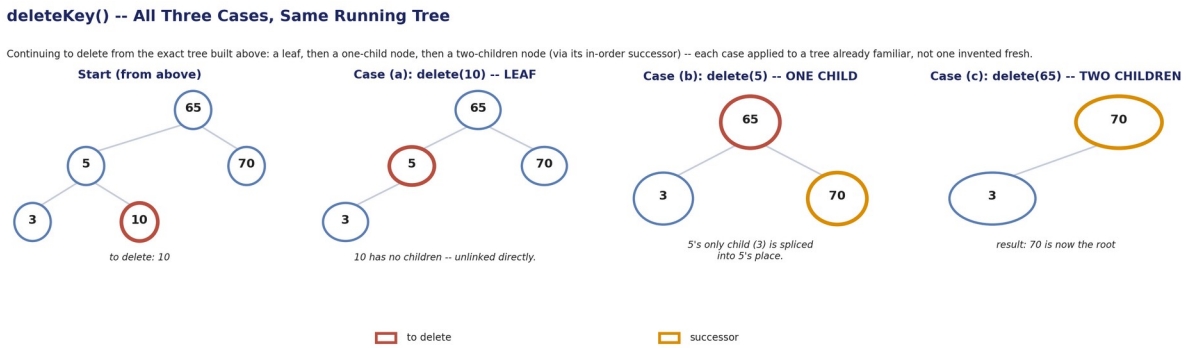


Figure 11.3 — deleteKey()'s three cases, applied in sequence to the running example: leaf, one child, then two children (via the in-order successor).

11.8.1 Case (a): Deleting a Leaf

If the node to delete has NO children, it is simply unlinked from its parent -- nothing needs to be reattached. Deleting 10 (a leaf, the right child of 5): the tree becomes count()=4, height()=2, with 10 gone and nothing else disturbed.

11.8.2 Case (b): Deleting a Node with One Child

If the node to delete has EXACTLY ONE child, that child is spliced directly into the deleted node's place -- the child (and its entire subtree, whatever size it may be) simply moves up one level. Continuing on the tree from case (a): 5 now has exactly one child left (3, since 10 was just removed). Deleting 5 splices 3 directly into 5's former place: the tree becomes count()=3, with 3 now 65's direct left child.

11.8.3 Case (c): Deleting a Node with Two Children — the In-Order Successor

If the node to delete has TWO children, neither child can simply move up (there is nowhere for the OTHER child to go). The standard fix: copy up the node's IN-ORDER SUCCESSOR -- the smallest key in the node's right subtree (found by following ->left as far as it goes from the right child) -- into the deleted node's position, then delete that successor from the right subtree, where it is now GUARANTEED to be a leaf or a one-child case (it has no left child, by definition of how it was found).

Continuing on the tree from case (b): the root, 65, now has two children (3 and 70). The in-order successor is the minimum of 65's right subtree -- here, simply 70 itself, since 70 has no left child. 65's key is overwritten with 70, and 70 is then deleted from the (now trivial) right subtree.

```
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;                                  // this IS the node to
remove

    if (!node->left && !node->right) { delete node; return nullptr; }        //
(a) leaf
    if (!node->left)  { BSTNode *r = node->right; delete node; return r; }    //
(b) right child only
    if (!node->right) { BSTNode *l = node->left;  delete node; return l; }    //
(b) left child only

    BSTNode *succ = node->right;                    // (c) two children:
    while (succ->left) succ = succ->left;          //     find the in-order
successor
    node->key = succ->key;                         //     copy its key up
    bool dummy = false;
    node->right = deleteNode(node->right, succ->key, dummy);  // remove it from
the right subtree
    return node;
}
```

`demo_delete.cpp`'s real, executed transcript confirms every step of this exact sequence:

```
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)
```

A deeper successor, tested separately

In the running example, 65's successor (70) happened to be its own right child -- the simplest sub-case. `demo_delete.cpp` also tests a strictly harder case on a separate, larger tree (insert(50,30,70,20,40,60,80,35,45), then delete(50)): there, the successor (60) is NOT an immediate child -- it is found two levels down inside the right subtree. The real result: 50's key is correctly overwritten with 60, and 60 is correctly removed from where it actually was, confirming the recursive splice works at any depth, not merely in the shallow case the main running example happens to show.

deleteKey() on a key that is not present

`deleteKey(999)` on the twice-deleted-down tree above correctly returns false and leaves the tree completely unchanged (count() still 2) -- the recursive search-then-delete simply falls through every comparison to a null subtree without ever setting found=true. Confirmed directly, not merely asserted.

11.9 Chapter Summary and Key Takeaways

- A TREE is this course's first genuinely NON-LINEAR structure: hierarchical, with a single root and no cycles. Core vocabulary -- root, parent/child, leaf, subtree, degree, depth (down from the root), height (up from a leaf) -- is used throughout the rest of this chapter and again in Chapter 13 (Graphs).
- Binary trees classify as FULL (every node has 0 or 2 children), COMPLETE (every level full except possibly the last, filled left to right), or SKEWED (every node has at most 1 child, degenerating to a linked list's shape and its O(n) worst case).
- The BINARY SEARCH TREE property -- left subtree < node < right subtree, at every node -- is what gives a reasonably balanced BST O(log n) average-case search and insertion, extending the Big-O thread from Chapter 2's sorted-array binary search and contrasting directly with Chapter 7's O(n) linked-list search.
- insert(65, 5, 70, 3, 10), this course's recurring worked example, builds a 5-node tree (root 65, left child 5, right child 70, 5's children 3 and 10) kept numerically identical throughout because Quiz 2's own model answer assumes it.

- PreOrder, InOrder, and PostOrder are small, direct recursive functions -- Chapter 10's own base-case/recursive-case shape, applied to a tree. LevelOrder is driven by a REAL std::queue<BSTNode*>, a direct, concrete callback to Chapter 5's Queue -- not merely an analogy.
- search(), count(), height(), findMin()/findMax(), and a leaf-finder round out this chapter's BST toolkit, each confirmed with real, measured results on the running example (count=5, height=2, min=3, max=70, leaves=[3,10,70]).
- A general tree (any number of children per node) converts losslessly to a binary representation via first-child/next-sibling; a general tree can be uniquely reconstructed from a PreOrder-plus-degree sequence, using the same recursive shape Chapter 10 established -- confirmed here with a genuine round-trip check.
- deleteKey() -- ENTIRELY NEW this chapter, since the raw material left it as an exercise and Quiz 2 explicitly tests it -- covers all three cases: (a) leaf, unlinked directly; (b) one child, spliced up into the deleted node's place; (c) two children, resolved by copying up the in-order successor (the minimum of the right subtree) and recursively deleting it from there. All three demonstrated in sequence on the SAME running tree, plus a separate, deeper example confirming the successor case works at any depth.

11.10 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Chapter 12 (Quiz 2), which tests the operations in this chapter -- especially deleteKey() -- directly on the insert(65,5,70,3,10) tree.

1. Explain, in your own words, the difference between a node's DEPTH and its HEIGHT, and state both values for the node holding key 5 in the insert(65,5,70,3,10) tree.
2. Explain why a SKEWED binary tree of n nodes gives every BST operation the exact same O(n) worst case as Chapter 7's singly linked list, and describe one concrete sequence of insertions into an empty BST that would produce a skewed tree.
3. Trace insert(40, 20, 60, 10, 30, 50, 70) by hand, one insertion at a time, the same way Section 11.4 traced insert(65,5,70,3,10), and state the resulting tree's height() and all four traversal results.
4. On the insert(65,5,70,3,10) tree, explain precisely why deleting node 65 requires finding an IN-ORDER SUCCESSOR while deleting node 10 does not, referring to the number of children each node has.
5. A general tree has root A with children B and C; B has children D, E, F; C has no children. Write out this tree's PreOrder-plus-degree sequence, the way Section 11.7.2 did for the 7-node A/B/C/D/E/F/G example, and state sum(degree).
6. Continuing from the insert(65,5,70,3,10) tree after case (c)'s deletion of 65 (leaving root=70, left child=3), trace deleteKey(70) by hand. Which of the three cases does this deletion fall into, and what does the resulting tree look like?

11.11 Appendix 11.A — Validation Log

All five programs below (`demo_insert.cpp`, `demo_traversals.cpp`, `demo_delete.cpp`, `demo_gtree.cpp`, `demo_all.cpp`) were compiled with `g++ -Wall -Wextra -std=c++17` (zero warnings from all five) and actually executed; the transcripts below are the real, unedited terminal output. `valgrind --leak-check=full --show-leak-kinds=all` was genuinely run against all five programs, per this course's standing policy since Chapter 9.

11.A.1 demo_all.cpp (full transcript)

```
$ 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 (excerpt — full transcript in the shipped code's _run_output.txt)

```
$ 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 (all five programs)

```
$ 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)
```

Honesty note

Every count, traversal result, and terminal line shown above (across all five programs) came directly from a real compile-and-run in this chapter's build environment. This chapter's validation pass found NO bugs in the BST or general-tree logic itself -- disclosed honestly, per this course's standing policy, rather than fabricating a finding where none exists. `valgrind` was run against every one of the five programs and every single run reported zero leaks and zero errors, confirming that BST's and GNode/BinNode's recursive destructors correctly free every dynamically allocated node across insertion, deletion, and general-tree conversion alike. The shipped code zip was independently re-extracted and rebuilt from a clean `make clean && make run` to confirm the actual deliverable, not just the working copy, compiles and runs identically.

References

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

[2] M.A. Weiss. “Data Structures and Algorithm Analysis in C++.” 4th ed., Pearson, 2014. (BST operations, tree balancing.)

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Binary search trees, tree height/depth definitions.)

[4] P.J. Deitel, H.M. Deitel. “C++ How to Program.” 10th ed., Pearson, 2017. (Tree traversals, recursion on trees.)

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

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 12 · EXERCISE CHAPTER

Quiz 2

No new material here — a TEAMWORK, open-book, take-home coding exam covering exactly what Chapter 11 taught: trees and the Binary Search Tree (BST). This chapter ships no code/ subfolder — the ten tasks below are your team's own work to design, write, and submit.

TEAMWORK ASSESSMENT — read this before anything else

Quiz 2 is explicitly a TEAM assignment: groups of UP TO 3 STUDENTS submit one shared set of solutions together. This is DIFFERENT from Quiz 1 (Chapter 4), the Midterm (Chapter 8), and the Final Exam (Chapter 16), which are all INDIVIDUAL. Confirm your group's membership and division of work before you start Section 12.3 — every task below assumes a team, not a lone student, is producing the answer.

Learning Objectives — after working through this chapter your team will be able to:

(1) Recognize that all ten Quiz 2 tasks below reduce to Chapter 11's BST recursion shape, so no new tree theory is required, only correct application to ten different questions about the same tree. (2) Write a depth-first print of a BST using any of Chapter 11's three DFS traversal shapes, and print all nodes at a specific level using a recursive depth-countdown. (3) Accumulate a running total across a BST traversal, and mutate every node's value in place using that same traversal shape. (4) Mirror a BST by recursively swapping every node's left and right children, and explain why the mirrored tree's InOrder is the reverse of the original's. (5) Determine whether a BST is height-balanced using an efficient, single-pass, bottom-up check — not a slower repeated-height-calls version. (6) Apply Chapter 11's deleteKey() (all three cases: leaf, one child, two children) to remove a specific node from the running tree. (7) Write a parent-finder and a children-finder over a BST, correctly distinguishing "value not found" from "value is the root" or "value is a leaf". (8) Convert a BST into a sorted, in-place doubly linked list by redirecting InOrder traversal's left/right pointers into prev/next pointers — allocating zero new nodes.

12.1 Recap: Chapter 11 in Brief

Chapter 11 introduced the first genuinely non-linear structure in this course, the tree, and its most important specialization for fast searching, the Binary Search Tree (BST): every node's LEFT subtree holds only smaller keys, and every node's RIGHT subtree holds only larger keys, applied recursively at every node. That chapter built one recurring numeric example, `insert(65, 5, 70, 3, 10)`, one key at a time, and covered all four traversals (PreOrder, InOrder, PostOrder, and a REAL-queue-driven LevelOrder), plus search(), count(), height(), findMin()/findMax(), a leaf-finder, general-tree conversion, and — added specifically because the raw course material left it as an exercise — deleteKey(), covering all three deletion cases.

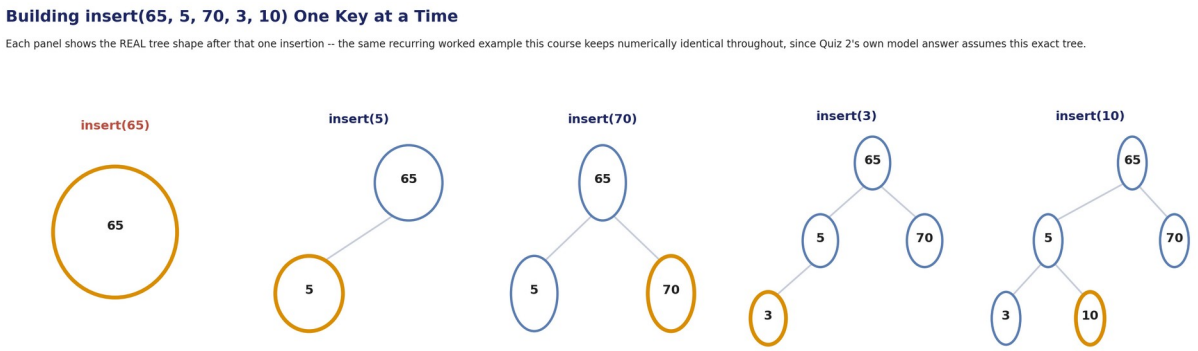


Figure 12.1 — Chapter 11's own recurring worked example, insert(65, 5, 70, 3, 10), built one key at a time. Every task in this chapter is posed against this exact tree.

The final shape, restated here because every task in Section 12.3 below depends on it:

```
        65
      /    \
     5      70
    / \
   3   10

count()  = 5     height()  = 2
PreOrder = [ 65, 5, 3, 10, 70 ]
InOrder  = [ 3, 5, 10, 65, 70 ]   (sorted)
PostOrder = [ 3, 10, 5, 70, 65 ]
```

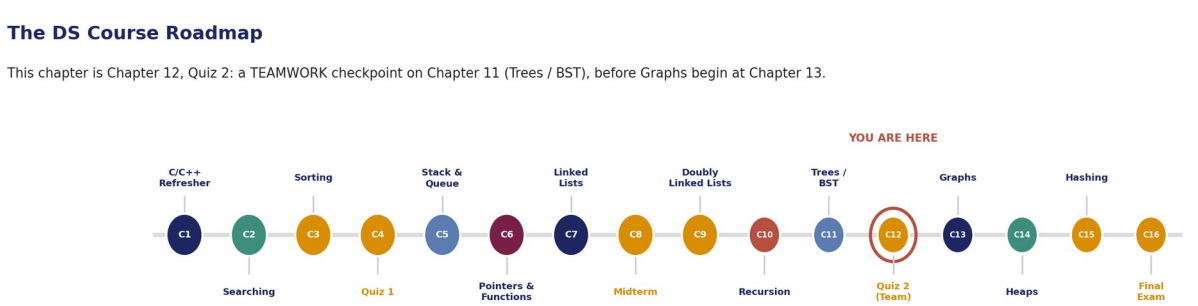


Figure 12.2 — This chapter sits at Chapter 12 in the sixteen-chapter DS roadmap: a checkpoint, not a new topic, Quiz 2 before Chapter 13 continues with Graphs.

> **Nothing below is new theory — only ten new questions about a tree you already know**
> Every one of Quiz 2's ten tasks (Section 12.3) reuses Chapter 11's own recursive traversal shape, its BSTNode structure, or its already-taught deleteKey() — pointed at a new question rather than a new idea. If a task feels unfamiliar, that is a signal to reread the relevant part of Chapter 11 (named in each task's callouts below), not a sign this quiz expects material never covered.

12.2 How to Use This Exercise Chapter

What this chapter is — and is not

This is a practice chapter, not a new-content chapter. Quiz 2, like the real DS assessment it is modeled on, is a TEAM, open-book, take-home coding exam — groups of up to 3 students submit together, unlike the individual Quiz 1, Midterm, and Final Exam. Section 12.3 below walks through Quiz 2's ten tasks, each with a guidance callout (THINK, TRAP, or INSIGHT) rather than a full worked solution. Like every other exercise chapter in this book, this chapter ships NO code/ subfolder — the programs your team writes for Quiz 2 are your own team's work, compiled and run on your own machines before you submit them.

Before your team starts

Assign each of the ten tasks below (or small clusters of them) to a specific team member before writing any code, and agree as a team on ONE shared BSTNode structure and insert() implementation — ideally Chapter 11's own — so every task operates on an identical tree. A team that reinvents the node structure ten separate times, once per task, will waste far more time reconciling mismatched code than a team that agrees on one shared header first.

12.3 The Quiz 2 Problem Set

Quiz 2 consists of ten independent coding tasks, each worth 10 points, for 100 points total.[1] Each task below asks for a function operating on the same insert(65, 5, 70, 3, 10) tree Section 12.1 restated above: read (or hard-code, if no input file is specified in your own course's instructions) the described input, produce the described output, and compile cleanly with `g++ -Wall -Wextra -std=c++17`, the same toolchain every chapter in this book has used since Chapter 1.

Independent tasks, independent starting trees

Unless your own team's test harness deliberately chains them for convenience, treat each of the ten tasks below as operating on its OWN freshly built insert(65, 5, 70, 3, 10) tree — Task 4 (triple every value) and Task 5 (mirror) each MUTATE the tree, and Task 7 (delete) removes a node from it, so none of Tasks 1, 2, 3, 6, 8, 9, or 10 should see a tree already altered by an earlier task's output unless the task explicitly says otherwise.

Where Each Quiz 2 Task Comes From

Quiz 2 is ten team-graded coding tasks, all posed against the SAME insert(65,5,70,3,10) tree Chapter 11 built -- every task below traces back to a specific Chapter 11 idea.

#	Quiz 2 Task	Traces back to	Pts
1	Depth-first print of the tree	Ch.11 -- PreOrder/InOrder/PostOrder: the recursive base-case/recursive-case traversal shape	10
2	Print all nodes at a given k-th level	Ch.11 -- LevelOrder/height() ideas, reframed as a recursive descent that counts down k	10
3	Compute the sum of all node values	Ch.11 -- traversal recursion, generalized from count() to accumulate key values instead	10
4	Triple every node's value, in place	Ch.11 -- traversal recursion applied as an in-place mutation of each visited node	10
5	Mirror the tree (swap left/right everywhere)	Ch.11 -- BSTNode's left/right pointers, recursively swapped at every node	10
6	Check whether the tree is height-balanced	Ch.11 -- height(), extended into an AVL-style |height(L)-height(R)| <= 1 check	10
7	Delete a given node from the tree	Ch.11 -- deleteKey(): the three deletion cases (leaf, one child, two children)	10
8	Find the parent of a given child value	Ch.11 -- BST search(), extended to remember the parent seen just before a match	10
9	Find the children of a given parent value	Ch.11 -- BST search(), reading a found node's left/right children directly	10
10	Convert the BST into an in-place sorted doubly linked list	Ch.11 -- InOrder traversal (already produces sorted output); left/right reused as prev/next	10

Total: 100 points

Figure 12.3 — Every task in this section traces back to a specific idea already covered in Chapter 11.

12.3.1 Task 1 — Depth-First Print of the Tree (10 pts)

Write a function that prints every node's key using a DEPTH-FIRST traversal — any of Chapter 11's three DFS shapes (PreOrder, InOrder, or PostOrder) is acceptable, as long as your team states clearly in the output or a comment which one it used. Applied to the running tree, a PreOrder print produces `65 5 3 10 70`; an InOrder print produces the sorted sequence `3 5 10 65 70`.

Hint

This is Chapter 11 Section 11.5, unchanged — a small recursive function with a null-subtree base case and two recursive calls, differing only in WHEN the current node's key is visited relative to those two calls. Reuse the exact shape (`if (!node) return;` then visit/recurse/recurse in whichever order you pick) rather than inventing a new traversal style.

Printing during a traversal is not the same as building a traversal

A function that prints directly (e.g., `std::cout << node->key`) as it recurses is easy to write but harder to test automatically, since its output is tied to console formatting. Consider instead building a `std::vector<int>` of visited keys (Chapter 11's own approach) and printing that vector afterward — the traversal logic is then identical whether your team is printing to a terminal or comparing against an expected sequence in an automated test.

12.3.2 Task 2 — Print All Nodes at a Given k-th Level (10 pts)

Given an integer k, print every node's key at level k of the tree, where the root is level 0. Applied to the running tree: level 0 prints `65`; level 1 prints `5 70`; level 2 prints `3 10`; any level k >= 3 (the tree's height is 2, so no node exists at level 3 or deeper) prints nothing, not an error.

Hint — recurse down, counting k toward zero

A direct recursive shape: `printLevel(node, k)` — base case, `node == nullptr`, return; second base case, `k == 0`, visit `node` and return; recursive case, `k > 0`, call `printLevel(node->left, k - 1)` then `printLevel(node->right, k - 1)`. Each recursive call descends one level and decrements k by exactly one, so the function naturally stops at the right depth without needing to track absolute depth from the root separately.

Off-by-one on whether the root is level 0 or level 1 is the classic bug here

Decide, and state explicitly in your team's own output format, whether the root counts as level 0 (this section's convention, matching Chapter 11's height() convention where a single-node tree has height 0) or level 1 — and apply that convention consistently across your k=0, k=1, and k=2 test calls. A k that is silently off by one will still run without crashing; it will just quietly print the WRONG level's nodes, which is easy to miss without checking against the tree diagram in Figure 12.1 by eye.

12.3.3 Task 3 — Compute the Sum of All Node Values (10 pts)

Write a function that returns the sum of every node's key in the tree. Applied to the running tree: 65 + 5 + 70 + 3 + 10 = 153.

Hint — this is count() with addition instead of counting

Chapter 11's count() function (Section 11.6) already showed the shape: a null subtree contributes 0, and any other node contributes 1 plus its two subtrees' counts. sumNodes() is identical except a non-null node contributes its OWN KEY plus its two subtrees' sums, instead of contributing 1: `return node->key + sumNodes(node->left) + sumNodes(node->right);`

An accumulator passed by reference must be initialized exactly once, before the traversal starts

If your team's solution instead threads a running total through an out-parameter (`void sumNodes(BSTNode* node, int& total)`) rather than returning a value, `total` must be initialized to 0 by the CALLER before the first call — an accumulator that is only reset on some code paths, or left holding a stale value from a previous test case, will silently produce a plausible-looking but wrong sum. The return-a-value version above avoids this entire class of bug and is the recommended shape for this task.

12.3.4 Task 4 — Triple Every Node's Value, In Place (10 pts)

Write a function that multiplies every node's key by 3, permanently, in the tree itself — not in a printed copy or a separate output structure. Applied to the running tree: 65, 5, 70, 3, 10 become 195, 15, 210, 9, 30, and the tree's InOrder afterward is `9 15 30 195 210`.

Hint — mutate through the actual node pointer, not a copy

The traversal shape is unchanged from Task 1 or Task 3 — visit every node exactly once — but this time the visit step is `node->key *= 3;` applied directly to the real `BSTNode` the recursion is holding a pointer to, not to a copy of its key pulled out into a local variable. If your function's parameter is `BSTNode* node` (a pointer, matching Chapter 11's own signatures), `node->key *= 3` correctly reaches the actual tree; a parameter accidentally taken by value as `BSTNode node` — a type error the compiler will normally catch, since BSTNode is not typically copyable in this course's designs — would not.

Tripling never breaks the BST property, and this is provable, not just observed

Multiplying every key by the same positive constant preserves every `<` and `>` relationship the original keys had, so a tree that was a valid BST before tripling is STILL a valid BST after — no rebalancing or restructuring is required, only the in-place key mutation itself. This is why the task can be solved as a pure traversal-with-side-effect rather than needing a rebuild-from-scratch step.

12.3.5 Task 5 — Mirror the Tree (Swap Every Node's Left/Right Subtree) (10 pts)

Write a function that recursively swaps every node's left and right child pointers throughout the whole tree, producing its mirror image. Applied to the running tree, the mirrored shape has 65 at the root with 70 now on the LEFT and 5 now on the RIGHT (with 5's own children 3 and 10 also swapped, so 10 is now on 5's left and 3 on 5's right).

Hint — one std::swap per node, applied recursively

`mirror(node)`: base case, `node == nullptr`, return; recursive case, `std::swap(node->left, node->right);` then recurse into BOTH of node's (now-swapped) children: `mirror(node->left); mirror(node->right);`. The swap must happen BEFORE the two recursive calls (or after — either order is correct here, since the recursive calls chase whichever pointers currently sit in `node->left`/`node->right`, and the swap only exchanges which child is which, not which nodes exist).

The mirrored tree's InOrder is exactly the original's InOrder, reversed

Because InOrder visits left-root-right, and mirroring exchanges every node's left and right subtree, mirroring the running tree turns its InOrder from `[3, 5, 10, 65, 70]` into `[70, 65, 10, 5, 3]` — the SAME five keys, in exactly reverse order. Confirming this on paper (or by actually running InOrder before and after your mirror() call) is a fast, reliable way to check your mirror() is correct without having to redraw the whole tree by hand. Note that the mirrored tree is no longer a BST in the usual ascending-left-to-right sense — it is now a general binary tree shaped like a BST's mirror image, which is exactly what this task asks for, not a bug.

12.3.6 Task 6 — Check Whether the Tree Is Height-Balanced (10 pts)

Write a function that returns true if the tree is HEIGHT-BALANCED — meaning that, at EVERY node in the tree (not just the root), the heights of its left and right subtrees differ by at most 1 — and false

otherwise. Applied to the running tree: the root (65) has a left subtree of height 1 (the 3-node subtree rooted at 5) and a right subtree of height 0 (the single leaf 70), a difference of 1, which is within the allowed limit; node 5 has a left subtree of height 0 (leaf 3) and a right subtree of height 0 (leaf 10), a difference of 0. Every node satisfies the condition, so the running tree IS height-balanced.

The naive approach recomputes height() at every node and is O(n²), not O(n)
A tempting first draft calls `height(node->left)` and `height(node->right)` separately at every node while also recursing into both children to check THEIR balance — this recomputes the height of every subtree many times over, giving $O(n^2)$ total work in the worst case (a tall, skewed tree), exactly the growth-rate trap Chapter 2 and Chapter 8's Quiz-style tasks have already warned about elsewhere in this course.

A single bottom-up pass gets both the height AND the balance check in O(n)
The efficient version has ONE recursive function that returns a subtree's height while ALSO checking balance as it goes, using a sentinel (commonly -1, or any value that cannot be a real height) to signal "already found unbalanced, stop checking further up": at each node, recursively get the left and right subtree's height-or-sentinel; if either child already reported the sentinel, or if the two heights differ by more than 1, propagate the sentinel upward immediately instead of continuing to compute a real height. This visits each node exactly once — O(n) total — the same "do it in one pass instead of many" idea Chapter 4's Quiz 1 applied to finding a first repeating character.

12.3.7 Task 7 — Delete a Given Node from the Tree (10 pts)

Given a key, remove that node from the tree while preserving the BST property, using Chapter 11's deleteKey() (Section 11.8): a LEAF is unlinked directly; a node with ONE child has that child spliced into its place; a node with TWO children has its key overwritten by its in-order successor's key, and that successor is then removed from the right subtree (which, having come from the successor-search, is guaranteed to have at most one child itself). Applied to the running tree, deleting key 5 (a one-child node — 5's only child, 3, is spliced into 5's place) leaves the tree `[65 [3, -], [70, -]]`, with InOrder `3 65 70`.

This is a direct review, not new material — but review it carefully
Chapter 11 Section 11.8 already worked through all three deletion cases on this exact tree, including a full diagram (Figure 11.3) tracing a leaf deletion, a one-child deletion, and a two-children deletion in sequence. Re-read that section before writing deleteKey() from memory — the in-order-successor step for the two-children case (find the MINIMUM of the right subtree, copy its key up, then delete that successor node from the right subtree) is the step most commonly mis-remembered.

Deleting a node must return the (possibly new) subtree root to its caller

Because the node actually removed from memory is sometimes the ROOT of the subtree passed in (for instance, deleting a leaf or one-child node), `deleteKey()` should be written to RETURN a `BSTNode*` — the subtree's root after the deletion — which the caller then assigns back to the pointer that used to hold the old subtree: `node->left = deleteKey(node->left, key);`. A version that tries to delete "in place" without returning and reassigning the (possibly changed) subtree root risks leaving a dangling pointer to freed memory, or leaving the parent still pointing at a node that no longer represents that subtree correctly.

12.3.8 Task 8 — Find the Parent of a Given Child Value (10 pts)

Given a key known to exist somewhere in the tree (other than the root), find and report the KEY of its parent node. Applied to the running tree: the parent of 3 is 5; the parent of 10 is 5; the parent of 5 is 65; the parent of 70 is 65. The root, 65, has no parent — your function must report that distinctly, not confuse it with "value not found".

Hint — track the node one step behind your search descent

This extends Chapter 11's search() (Section 11.6.1) with exactly one extra piece of state: as you descend comparing the target key against each node's key to decide which way to go, keep a second pointer, `parent`, updated to the CURRENT node just before you move to `node->left` or `node->right`. When the descent finds a node whose key matches the target, `parent` already holds the node one step above it — no separate second traversal is needed.

Three genuinely different outcomes, and your team's design must not collapse them into two

This task has THREE possible results, not two: (a) the key exists and has a parent — report the parent's key; (b) the key exists but IS the root — report explicitly that it has no parent, distinct from "not found"; (c) the key does not exist in the tree at all — report "not found". A design that returns, say, a single `int` with `-1` meaning both "is the root" and "not found" makes these two genuinely different situations indistinguishable to whatever calls it — prefer a small result type (an `std::optional<int>` plus a separate found-flag, or a struct with an explicit `{FOUND_WITH_PARENT, FOUND_IS_ROOT, NOT_FOUND}` tag) that keeps all three cases distinct.

12.3.9 Task 9 — Find the Children of a Given Parent Value (10 pts)

Given a key known to exist somewhere in the tree, report its LEFT child's key and its RIGHT child's key, if each exists. Applied to the running tree: the children of 65 are 5 (left) and 70 (right); the children of 5 are 3 (left) and 10 (right); the children of 70 are none (it is a leaf); the children of 3 and 10 are also none.

Hint — this is search() plus a direct field read, nothing more

Reuse Chapter 11's search()-style descent to LOCATE the node holding the given key (comparing against `node->key` at each step, going left or right accordingly, exactly as Section 11.6.1 does), and once found, simply read `node->left` and `node->right` directly — if a child pointer is `nullptr`, that side has no child, which is worth reporting explicitly ("no left child" / "no right child" / "leaf, no children") rather than silently printing nothing.

A key not present in the tree at all is a different case from a key present but childless

Just as Task 8 warned against collapsing "is the root" into "not found", this task's design must not collapse "the key exists but is a leaf (zero children)" into "the key was not found" — these are different outcomes your team's function should report distinctly. Test your solution explicitly against a leaf (key 3, 10, or 70), a node with one child does not exist in this particular tree but would need the same care if your team tests other trees, and a key genuinely absent from the tree (for example, 999) to confirm all three read as different results.

12.3.10 Task 10 — Convert the BST into an In-Place Sorted Doubly Linked List (10 pts)

Convert the tree into a sorted DOUBLY LINKED LIST, reusing the EXISTING nodes' `left` and `right` pointers as `prev` and `next` respectively — allocate ZERO new nodes. The result, applied to the running tree, is the chain `3 <-> 5 <-> 10 <-> 65 <-> 70`, in ascending order, where the list's HEAD is the same node that used to hold key 3 and the list's TAIL is the same node that used to hold key 70.

Nothing new in theory — a familiar traversal, redirected

Chapter 11 already established that InOrder traversal visits a BST's keys in SORTED order (Section 11.5.2, confirmed programmatically, not just observed on one example). This task does not require new theory beyond that fact and this course's traversal recursion shape — it only asks your team to change WHAT the traversal does at each visit: instead of appending a key to an output vector, it relinks two ACTUAL nodes (the previously-visited node and the current one) into a chain, reusing their existing left/right fields as prev/next.

Hint — carry a "previously visited node" pointer through the InOrder recursion

A standard approach threads one extra piece of state through an InOrder-shaped recursion: a `prev` pointer (or reference), initialized to `nullptr` before the traversal starts. At each node, AFTER recursing into `node->left` (which finishes converting everything smaller), do: if `prev` is not `nullptr`, set `prev->right = node` and `node->left = prev` (the relink); then set `prev = node` before recursing into `node->right`. The very first node visited (the smallest key, 3) never receives a `left` link, correctly becoming the list's head; the very last node visited (the largest key, 70) never receives a `right` link from this process, correctly becoming the list's tail.

Allocating even one new node anywhere defeats the entire point of this task

It is tempting to build a fresh, ordinary doubly linked list of `Book`-style or `int`-holding nodes by traversing the BST and calling `new` for each value — this WILL produce a sorted doubly linked list, but it is not what this task asks for, and typically will not earn full marks, since the task specifically requires reusing the tree's OWN existing nodes' pointer fields, not copying their values into new memory. If your team's solution would still work with the original tree's nodes being non-copyable, opaque objects that can only be relinked (never duplicated), you are solving the intended task; if it needs to copy a `key` field into a brand-new node, it is not.

12.4 Quiz 2 Format and Grading

Quiz 2 is a TEAM, open-book, take-home coding exam — groups of UP TO 3 STUDENTS submit one shared set of solutions together. This is different from Quiz 1 (Chapter 4), the Midterm (Chapter 8), and the Final Exam (Chapter 16), each of which is INDIVIDUAL. "Open-book" means the textbook, your team's own notes, and standard C++ reference material (such as cppreference.com) are all fair game while you work; it does NOT mean copying another team's code or submitting work that is not genuinely your own team's — an open-book, team exam still tests whether YOUR TEAM can apply what the course has taught, working together rather than independently.

DS's own point-per-subtask rubric

Like every other exercise chapter in this book, Quiz 2 is graded task by task, in whole points, not by a percentage-weighted Design/Implementation/Analysis/Conclusion template. This textbook allocates all ten tasks a flat 10 points each (100 total)[1], awarded for code that compiles cleanly and produces the correct output for the stated requirement (including, for Task 4, genuinely mutating the tree's own nodes rather than printing a tripled copy, for Task 6, a check that correctly evaluates the balance condition at every node rather than only the root, and for Task 10, genuinely reusing the tree's own nodes rather than allocating new ones). There is no separate "analysis" or "conclusion" component to this quiz — the code and its correct output are the deliverable.

#	Task	What it tests	Pts
1	Depth-first print	A correct DFS traversal (any of PreOrder/InOrder/PostOrder), clearly labeled	10
2	Print nodes at level k	A correct recursive descent stopping at exactly level k, root counted as level 0	10
3	Sum of all node values	A correct accumulation across every node, matching count()'s traversal shape	10
4	Triple every value in place	A correct in-place mutation of the tree's own nodes, not a printed copy	10
5	Mirror the tree	A correct recursive left/right swap at every node, reversing the InOrder result	10

6	Height-balanced check	A correct balance check at EVERY node, ideally in a single O(n) bottom-up pass	10
7	Delete a given node	A correct application of deleteKey()'s three cases, with the subtree root returned/reassigned	10
8	Find the parent of a value	Correctly distinguishing has-a-parent, is-the-root, and not-found	10
9	Find the children of a value	Correctly distinguishing has-children, is-a-leaf, and not-found	10
10	BST to in-place sorted DLL	A correct InOrder-driven relink reusing the tree's own nodes, zero new allocations	10
	Total		100

[1] *A rubric footnote on this even split: the raw teaching material behind this book's research confirms Quiz 2's ten tasks and its 100-point total, and confirms its point-per-subtask grading shape (matching every other DS exercise chapter, and unlike Quiz 1's own flat 25-points-per-task or the Midterm's deliberately uneven 10/30/30/15/15 split) — but does not preserve an unambiguous per-task point breakdown across all ten tasks in the material this textbook was built from. Rather than guessing at an uneven split with no reliable source, this textbook allocates all ten tasks an equal 10 points each, summing to the same 100-point total the real Quiz 2 uses. If your own course's instructions specify a different per-task breakdown, follow your own instructions over this footnote — only the task WORDING and the TREE above are meant to be authoritative here.*

Submit all ten functions as a single, coherent submission (either one program exercising all ten in sequence, or ten small, individually compilable pieces sharing one `BSTNode`/tree header — follow your own course's submission instructions), each demonstrating its own correct output when run on the insert(65, 5, 70, 3, 10) tree — a program that does not compile earns no points for the tasks it contains, regardless of how close its logic is to correct, so leave time for your WHOLE TEAM to actually build and run every task before submitting, exactly the discipline every code example in Chapters 1 through 11 was held to.

12.5 Chapter Summary

- Quiz 2 introduces no new material — it is a checkpoint on Chapter 11 (Trees / BST), recapped in Section 12.1, posed as ten questions against the SAME insert(65, 5, 70, 3, 10) tree throughout.
- It is a TEAM, open-book, take-home coding exam — groups of UP TO 3 STUDENTS, unlike the individual Quiz 1, Midterm, and Final Exam.
- Ten tasks worth 10 points each (100 total): depth-first print, print nodes at level k, sum all values, triple every value in place, mirror the tree, check height-balance, delete a given node, find a value's parent, find a value's children, and convert the BST into an in-place sorted doubly linked list.

- Grading follows DS's own native point-per-subtask rubric — this book's flat 10-points-per-task split is a documented, transparent choice (see Section 12.4's rubric footnote), not a claim of an exact original point breakdown this project could not independently confirm.
- Task 6 (height-balance) and Task 7 (delete) are the two tasks most worth extra care: Task 6's naive solution is $O(n^2)$ instead of $O(n)$, and Task 7 must correctly return and reassign a subtree root rather than deleting "in place" without doing so.
- This chapter ships no code/ subfolder: the ten tasks are your team's own collective work, to be compiled with the same g++ -Wall -Wextra -std=c++17 toolchain used throughout this book and genuinely run before submission.

A program that compiles is not the same as a program that is correct — and a program that is correct on the one tree you tried is not the same as one that is correct in general. As a team, test your ten solutions against more than the running example where practical, including at least one deliberately awkward case per task (a level k deeper than the tree's height for Task 2, a target key that is the root for Task 8, a target key that is a leaf for Task 9, and a repeated round-trip check — build the DLL, then confirm walking it forward and backward both produce the sorted sequence — for Task 10) before you submit.

Appendix 12.A — Self-Check Checklist (Non-Graded)

Before submitting your team's ten solutions, run them through this checklist together. It costs a few minutes and catches most of the mistakes graders see most often on a first Quiz 2 submission.

- Does every team member's code compile cleanly with `g++ -Wall -Wextra -std=c++17`, with zero warnings, once combined into one submission?
- Task 1: does your DFS print state clearly which of PreOrder/InOrder/PostOrder it uses?
- Task 2: does level k = 0 correctly print just the root, and does a level deeper than the tree's height correctly print nothing (not crash)?
- Task 3: does your sum use a value RETURNED from the recursion, or if it uses a reference accumulator, is that accumulator initialized to 0 exactly once by the caller?
- Task 4: after tripling, does an InOrder traversal of the SAME tree object show 9, 15, 30, 195, 210 — confirming the mutation happened in place, not on a copy?
- Task 5: does an InOrder traversal after mirror() produce the ORIGINAL InOrder reversed (`70, 65, 10, 5, 3`)?
- Task 6: have you checked your height-balance function is not needlessly recomputing height() separately at every node (an $O(n^2)$ smell)?
- Task 7: after deleting a node, is the subtree root correctly reassigned at the CALLER (e.g., `node->left = deleteKey(node->left, key);`), not merely mutated "in place"?
- Task 8: does your function correctly distinguish a value that IS the root (no parent) from a value NOT present in the tree at all?

- Task 9: does your function correctly distinguish a value that is a LEAF (zero children) from a value NOT present in the tree at all?
- Task 10: does the resulting doubly linked list contain ZERO newly allocated nodes — only the original tree's own nodes, relinked?
- Has your team removed or commented out any leftover debug `printf`/`cout` lines that were not part of the requested output format?
- Is every file your team is submitting genuinely your OWN TEAM's work, written for this quiz — not copied from another team, a prior year's solution, or an AI tool your course's own academic-integrity policy does not permit?

References

[1] This exercise chapter's assessment format is modeled directly on this course's real Quiz 2 (EF234201 Data Structure): a team (up to 3 students), open-book, take-home coding exam of ten recursive-tree tasks over the same insert(65, 5, 70, 3, 10) worked example this course's Chapter 11 teaches. This project's own research could not independently confirm an unambiguous per-task point breakdown for all ten tasks from the raw source material available, so this textbook allocates a flat, transparently documented 10 points per task (Section 12.4's rubric footnote); if your own course's instructions specify a different breakdown, follow those instructions.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Binary search trees, Chapter 12; balanced trees, Chapter 13.)

[3] cppreference.com. “Recursion,” “std::swap,” “Binary tree (data structure).” https://en.cppreference.com/w/cpp (accessed August 2026).

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 13

Graphs

Every structure this course has built so far -- arrays, lists, stacks, queues, and even Chapter 11's trees -- restricts each element to at most one "parent." A GRAPH drops that restriction entirely: any vertex may connect to any number of others, in any pattern at all. This chapter's content is entirely NEW -- the raw teaching material behind this course had nothing usable on graphs (its "Graph" lecture slides turned out, on inspection, to be an exact duplicate of the Trees lecture) -- so both the theory and the running worked example below are authored fresh, following standard graph-theory pedagogy. One graph, Pustaka Ceria's campus map, is reused unchanged across every section: the adjacency matrix, the adjacency list, BFS, DFS, and connectivity. Every line of code here was actually compiled with g++ -Wall -Wextra -std=c++17, actually run, and independently checked with valgrind -- the full transcript is reproduced in Appendix 13.A.

Learning Objectives — after this chapter you will be able to:
(1) Define a graph's core vocabulary -- vertex, edge, directed vs. undirected, weighted vs. unweighted, degree (including separate in-degree/out-degree for a directed graph), path, cycle, and connected vs. disconnected. (2) Build and compare two representations of the SAME graph -- the adjacency matrix and the adjacency list -- implemented, compiled, and run in C++, with real measured space and complexity differences, not merely argued in the abstract. (3) Implement and trace Breadth-First Search (BFS) using a real Queue -- Chapter 5's own array-based Queue, unchanged -- and explain why BFS visits a graph level by level. (4) Implement and trace Depth-First Search (DFS) two ways -- a small recursive version (Chapter 10's base-case/recursive-case shape) and an iterative version using a real Stack, also Chapter 5's own -- and confirm both agree on the reachable vertex set. (5) Find every connected component of an undirected graph, and build a simple "is this graph connected?" test, both directly from BFS. (6) Apply every concept above to a single running worked example -- Pustaka Ceria's campus map -- reused consistently from representation, through traversal, through connectivity.

13.1 Graph Terminology and Fundamentals

A GRAPH is a collection of VERTICES (also called nodes -- this chapter uses "vertex"/"vertices" throughout, the standard graph-theory term) connected by EDGES. Unlike Chapter 11's trees, a graph has no single root, no notion of "parent" and "child," and no restriction against cycles: any vertex may be connected to any number of other vertices, in any pattern at all. Formally, a graph G = (V, E) is a set of vertices V together with a set of edges E, where each edge connects a pair of vertices.

13.1.1 Directed vs. Undirected

An edge is UNDIRECTED if it simply connects two vertices with no direction implied -- if there is an edge between A and B, you can travel A-to-B or B-to-A equally. An edge is DIRECTED if it points from one vertex to another specifically -- an edge from A to B does NOT imply an edge from B to A. A graph is called undirected or directed according to which kind of edge it uses throughout (mixing the two in one graph is unusual and not covered here).

Pustaka Ceria's campus map (Section 13.1.4 and onward) is UNDIRECTED: a footpath between the Library and the Cafeteria can be walked in either direction. A "this book recommends that book" relationship, by contrast, is naturally DIRECTED: BookA recommending BookB does not mean BookB recommends BookA back.

13.1.2 Weighted vs. Unweighted

An edge is WEIGHTED if it carries an extra numeric value -- a cost, a distance, a duration -- beyond simply "these two vertices are connected." An edge is UNWEIGHTED if no such value is attached (equivalently, every edge can be thought of as carrying an implicit weight of 1). Pustaka Ceria's campus map is WEIGHTED: every footpath carries a real distance in meters, used throughout this chapter's code.

13.1.3 Degree, In-Degree, and Out-Degree

The DEGREE of a vertex, in an undirected graph, is simply the number of edges touching it. In a DIRECTED graph, one number is not enough -- a vertex can have edges pointing IN and edges pointing OUT, and these are counted separately: the IN-DEGREE of a vertex is the number of edges pointing INTO it, and the OUT-DEGREE is the number of edges pointing OUT of it.

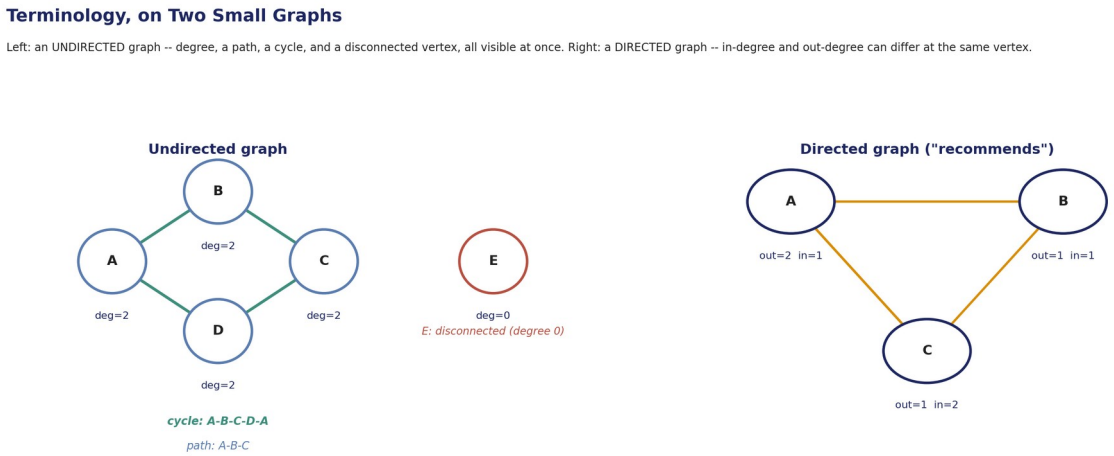


Figure 13.1 — left: an undirected 4-vertex cycle plus one disconnected vertex, with each vertex's degree labeled. Right: a small directed "recommends" graph, with in-degree and out-degree labeled separately at each vertex.

On the small directed graph in Figure 13.1 (right panel): A recommends both B and C (out-degree 2), but only C recommends A back (in-degree 1) -- confirming directly that in-degree and out-degree can differ at the very same vertex, something that never happens in an undirected graph, where in and out are the same thing by definition.

13.1.4 Path, Cycle, and Connectivity

A PATH is a sequence of vertices where each consecutive pair is joined by an edge -- Figure 13.1's left panel shows the path A-B-C. A CYCLE is a path that returns to its own starting vertex without repeating any other vertex along the way -- A-B-C-D-A is a cycle in that same graph. A graph (or a part of it) is CONNECTED if a path exists between every pair of its vertices; it is DISCONNECTED if at least one pair

of vertices has no path between them at all. Figure 13.1's left panel is deliberately disconnected: vertex E has degree 0 (no edges at all), so no path reaches it from A, B, C, or D.

> **"Connected" describes a graph as a whole, not one vertex**
> A single isolated vertex like E does not make the WHOLE graph "disconnected" in some vague sense -- it is precise: the graph {A,B,C,D,E} is disconnected because at least one pair (any pair involving E) has no path between them, even though {A,B,C,D} on its own IS connected. Section 13.4 makes this precise with a real algorithm: connected COMPONENTS are exactly the maximal connected pieces a disconnected graph breaks into -- here, {A,B,C,D} is one component and {E} is the other.

13.2 Representing a Graph in Code

Before any traversal algorithm can run, a graph needs to actually live somewhere in memory. This section builds and compares TWO representations of the exact same graph -- so Section 13.2.3's comparison is a measurement on one identical structure, not an argument about two different ones.

The worked example reused for the rest of this chapter is Pustaka Ceria's CAMPUS MAP: ten campus locations, connected by ten weighted, undirected footpaths (distances in meters). Locations 0-7 form the main campus cluster around the Library; locations 8-9 (a small off-site Annex and Warehouse) are connected to each other but -- deliberately, for Section 13.4's sake -- not yet to the rest of campus.

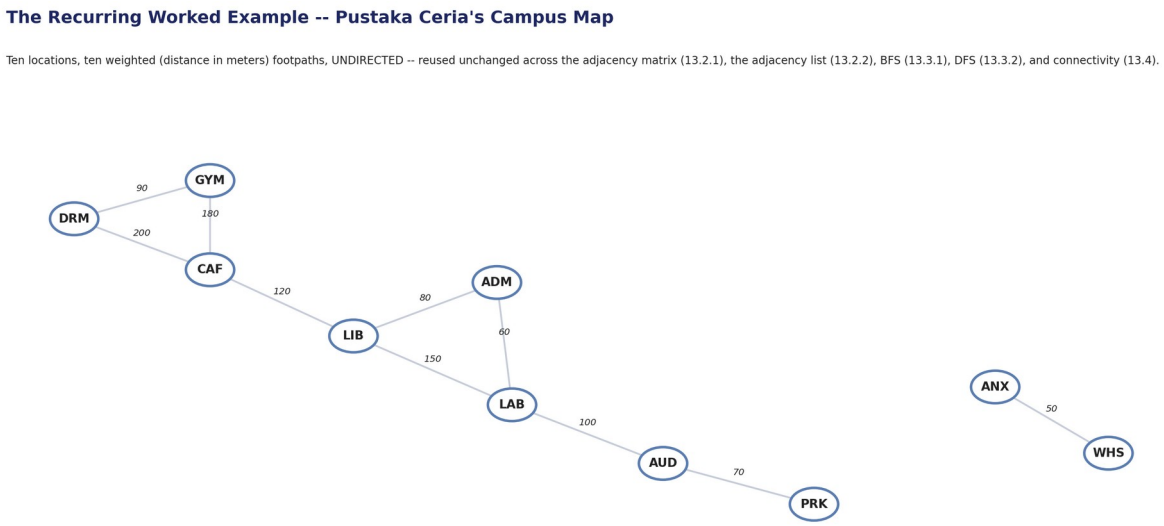


Figure 13.2 — Pustaka Ceria's campus map: this chapter's recurring worked example, reused unchanged across the adjacency matrix, the adjacency list, BFS, DFS, and connectivity.

13.2.1 The Adjacency Matrix

An ADJACENCY MATRIX stores a graph of n vertices as an n x n grid: cell [u][v] holds the weight of the edge between u and v (or a NO_EDGE sentinel if none exists). For an undirected graph, the matrix is always symmetric -- cell [u][v] always equals cell [v][u], since an undirected edge has no direction to distinguish them.

```cpp
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;   // undirected: write BOTH directions
        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): scans the ENTIRE row
        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), always
    // ... addEdge/hasEdge/weight/degree/print ...
};
```

`demo_matrix.cpp`'s real, executed output confirms the matrix's real shape on the campus map:

```
       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 ints allocated (V*V),
of which only 20 cells hold a real edge (2 per undirected edge, 10 edges).
That means 80 of 100 cells (80%) are wasted NO_EDGE entries.
```

13.2.2 The Adjacency List

An ADJACENCY LIST stores, for each vertex, only its ACTUAL neighbors -- a `vector<vector<pair<int,int>>>` here, where adj[v] holds every (neighbor, weight) pair for vertex v. For an undirected graph, addEdge(u, v, w) pushes the edge onto BOTH u's and v's lists.

```
class GraphList {
public:
    void addEdge(int u, int v, int weight = 1) {
        adj[u].push_back({v, weight});   // undirected: push onto BOTH lists
        adj[v].push_back({u, weight});
    }
    bool hasEdge(int u, int v) const {           // O(degree(u)) -- scans ONLY
u's list
        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), always
        int total = 0;
        for (const auto &lst : adj) total += (int)lst.size();
        return total;
    }
    // ... weight/edgeCount/neighbors/print, plus every traversal algorithm
below ...
private:
    std::vector<std::vector<std::pair<int, int>>> adj;
};
```

`demo_list.cpp`'s real, executed output confirms the SAME campus map, this time as a list, and measures its footprint directly against the matrix's:

```
  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 (neighbor,weight)
pairs
Compare: the adjacency MATRIX on this identical graph allocates 100 cells;
the adjacency LIST allocates 20 entries -- 5x less, for a graph this sparse
(10 edges out of a possible 45).
```

13.2.3 Space and Time: A Direct Comparison

Both representations were just measured on the EXACT SAME campus map -- so the numbers below are real, not hypothetical. This directly extends the Big-O comparison habit Chapter 3 established for sorting algorithms.

Operation	Adjacency Matrix	Adjacency List	Measured (V=10, E=10)
Space	O(V^2)	O(V + E)	100 cells vs. 20 entries
hasEdge(u,v)	O(1)	O(degree(u))	matrix wins for this one query
Iterate neighbors of v	O(V)	O(degree(v))	list wins -- visits only real neighbors
degree(v)	O(V)	O(1)	list wins -- just the list's length

Add an edge	O(1)	O(1)	tie
Add a vertex	O(V^2) (resize the whole matrix)	O(1) (append an empty list)	list wins
Best suited for	dense graphs (E close to V^2)	sparse graphs (E << V^2)	campus map: sparse -- list wins overall

Why every traversal in this chapter is written against the LIST

Section 13.3's BFS and DFS only ever need one operation, repeatedly: "give me v's real neighbors, so I can visit each one." An adjacency list hands this over directly, in O(degree(v)), which is exactly what it stores; an adjacency matrix would force an O(V) scan of an entire row just to find the same handful of real neighbors, most of whose cells are NO_EDGE. This is precisely why graph_list.h, not graph_matrix.h, is where every traversal and connectivity algorithm in this chapter lives.

13.3 Traversal: Visiting Every Reachable Vertex

A GRAPH TRAVERSAL visits every vertex reachable from some starting vertex, exactly once. This chapter covers two traversal strategies -- Breadth-First Search (BFS) and Depth-First Search (DFS) -- both applied to the SAME starting point, the Library, on the SAME campus map.

13.3.1 Breadth-First Search (BFS) — Driven by a Real Queue

BFS visits a graph LEVEL BY LEVEL: first the start vertex, then every vertex ONE edge away, then every vertex TWO edges away, and so on. The standard way to implement this is with a queue: enqueue the start vertex, then repeatedly dequeue a vertex, visit it, and enqueue every one of its not-yet-visited neighbors. This is not merely analogous to Chapter 5's Queue -- `bfs()` below literally instantiates a `Queue<int>` (Chapter 5's own array-based Queue, `queue.h`, copied forward unchanged) and calls its `enqueue()`/`dequeue()`, the exact same FIFO discipline Chapter 5 built by hand, now driving a breadth-first walk of a graph.

```
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);   // Chapter 5's Queue, used
directly here
    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;
}
```

Mark a vertex visited when it is ENQUEUED, not when it is dequeued

If a vertex were only marked visited at dequeue time, the same vertex could be enqueued more than once by two different neighbors before its first dequeue -- wasting work, and in a graph with cycles, potentially enqueuing it unboundedly many times. Marking `visited[nb.first] = true` at the moment of enqueueing (not at dequeue time) guarantees every vertex is enqueued at most once.

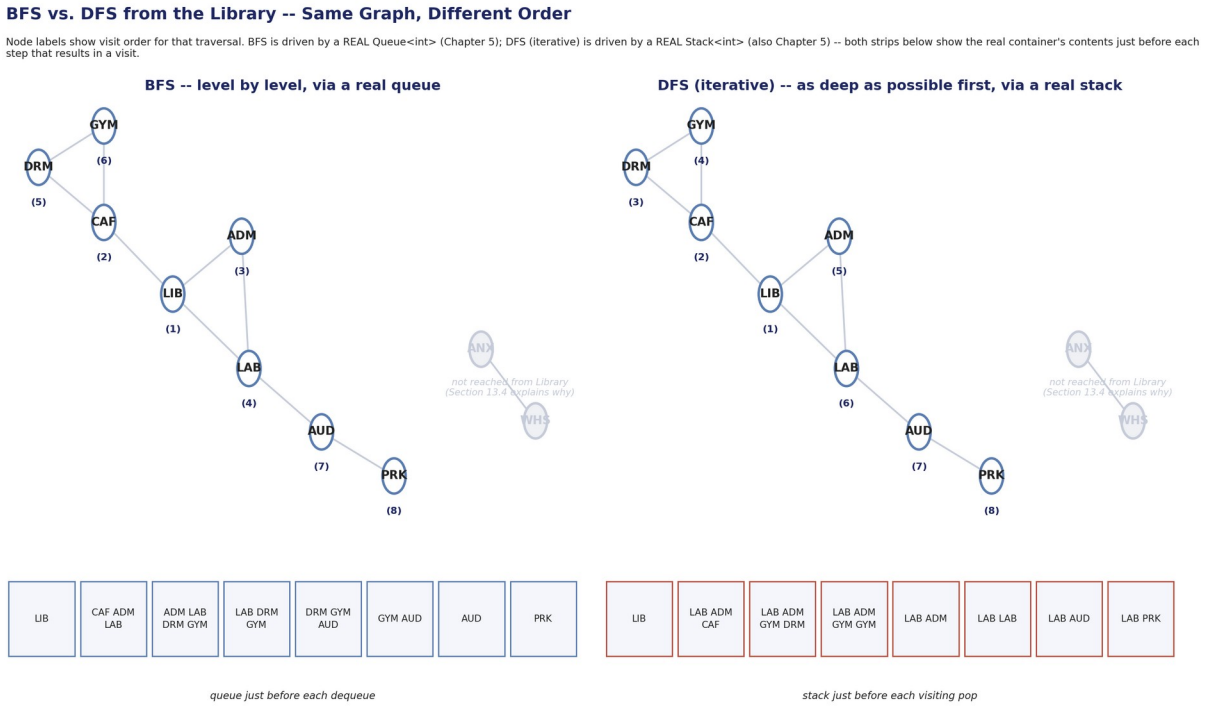


Figure 13.3 — BFS (left) and DFS, iterative (right), both starting from the Library on the identical campus map. Node labels show each traversal's own visit order; the strips below show the real Queue's / Stack's contents just before each step that results in a visit. The Annex and Warehouse (dimmed) are never reached from the Library — Section 13.4 explains why.

`demo_bfs.cpp`'s real, executed transcript, traced step by step:

```
  step 1: queue = [ Library ]  -> dequeue Library, visit it, enqueue Cafeteria,
enqueue AdminOffice, enqueue LabBuilding
  step 2: queue = [ Cafeteria AdminOffice LabBuilding ]  -> dequeue Cafeteria,
visit it, enqueue Dorm, enqueue Gym
  step 3: queue = [ AdminOffice LabBuilding Dorm Gym ]  -> dequeue AdminOffice,
visit it
  step 4: queue = [ LabBuilding Dorm Gym ]  -> dequeue LabBuilding, visit it,
enqueue Auditorium
  step 5: queue = [ Dorm Gym Auditorium ]  -> dequeue Dorm, visit it
  step 6: queue = [ Gym Auditorium ]  -> dequeue Gym, visit it
  step 7: queue = [ Auditorium ]  -> dequeue Auditorium, visit it, enqueue
Parking
  step 8: queue = [ Parking ]  -> dequeue Parking, visit it

BFS visit order: [ Library Cafeteria AdminOffice LabBuilding Dorm Gym Auditorium
Parking ]
Visited 8 of 10 locations. NOT reached from Library: Annex Warehouse
```

13.3.2 Depth-First Search (DFS) — Recursive, and Iterative With a Real Stack

DFS visits a graph AS DEEP AS POSSIBLE along one branch before backtracking -- the opposite exploration order from BFS's level-by-level spread, on the exact same graph. This chapter implements DFS two ways, both starting from the Library.

13.3.2a DFS, Recursive

The recursive version is a small, direct function -- another concrete callback to Chapter 10's base-case/recursive-case shape, this time applied to a graph instead of a single chain of calls or a 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);   //
RECURSIVE CASE
    }
    // no explicit base case line needed: the loop simply does not recurse
    // further once every neighbor has already been visited
}
```

13.3.2b DFS, Iterative — Driven by a Real Stack

The iterative version replaces the implicit call stack with a REAL `Stack<int>` -- Chapter 5's own array-based Stack, `stack.h`, copied forward unchanged. It marks a vertex visited WHEN POPPED (not when pushed), the simple, standard textbook shape -- which means the same vertex CAN be pushed more than once before it is first visited, and a later pop of an already-visited vertex is simply skipped.

```
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);   // Chapter 5's Stack,
used directly here
    st.push(start);
    while (!st.isEmpty()) {
        int u; st.pop(u);
        if (visited[u]) continue;                          // already visited --
skip
        visited[u] = true;
        order.push_back(u);
        // push neighbors in REVERSE order, so the first-listed neighbor
        // ends up on TOP and is popped (and visited) first
        for (auto it = adj[u].rbegin(); it != adj[u].rend(); ++it) {
            if (!visited[it->first]) st.push(it->first);
        }
    }
    return order;
}
```

`demo_dfs.cpp`'s real, executed transcript, both versions:

```
DFS (recursive) visit order: [ Library Cafeteria Dorm Gym AdminOffice
LabBuilding Auditorium Parking ]

  step 1: stack = [ Library ]  -> pop Library, visit it, push LabBuilding, push
AdminOffice, push Cafeteria
  step 2: stack = [ LabBuilding AdminOffice Cafeteria ]  -> pop Cafeteria, visit
it, push Gym, push Dorm
  step 3: stack = [ LabBuilding AdminOffice Gym Dorm ]  -> pop Dorm, visit it,
push Gym
  step 4: stack = [ LabBuilding AdminOffice Gym Gym ]  -> pop Gym, visit it
  step 5: stack = [ LabBuilding AdminOffice Gym ]  -> pop Gym (already visited,
skip)
  step 6: stack = [ LabBuilding AdminOffice ]  -> pop AdminOffice, visit it,
push LabBuilding
  step 7: stack = [ LabBuilding LabBuilding ]  -> pop LabBuilding, visit it,
push Auditorium
  step 8: stack = [ LabBuilding Auditorium ]  -> pop Auditorium, visit it, push
Parking
  step 9: stack = [ LabBuilding Parking ]  -> pop Parking, visit it
  step 10: stack = [ LabBuilding ]  -> pop LabBuilding (already visited, skip)

DFS (iterative) visit order: [ Library Cafeteria Dorm Gym AdminOffice
LabBuilding Auditorium Parking ]

Same SET of vertices visited (order aside)? YES
Same VISIT ORDER, recursive vs. iterative? YES -- pushing neighbors in reverse
adjacency
order makes the explicit stack mirror the call stack exactly
```

Two of the ten stack steps do real, visible extra work -- and that is fine

Steps 5 and 10 above pop a vertex (Gym, then LabBuilding) that turns out to already be visited, and simply move on. This is the honest cost of the simple "push without checking, skip on pop" style: it does a little more work than a stricter "check before pushing" variant would, but it is simpler to write correctly and still visits every reachable vertex exactly once in its `order` result -- confirmed directly by `demo_all.cpp`'s automated checks (Appendix 13.A), not merely asserted here.

13.4 Connectivity: Connected Components and "Is This Graph Connected?"

Section 13.3's BFS from the Library reached only 8 of the campus map's 10 locations -- the Annex and Warehouse were never enqueued at all, because no footpath connects them to the rest of campus. This section makes that observation precise and automatic: finding every CONNECTED COMPONENT of an undirected graph, and building a simple, direct "is this graph connected?" test -- both straight from BFS, no new algorithm required.

13.4.1 Connected Components

A CONNECTED COMPONENT is a maximal set of vertices where every pair has a path between them. Finding every component of a graph is a small, direct extension of BFS: repeat BFS from every vertex NOT YET visited by an earlier sweep; each such BFS sweeps out exactly one entire component, since BFS itself already visits every vertex reachable from its starting point.

```
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));   // one full BFS
sweep = one component
        }
    }
    return comps;
}
```

13.4.2 "Is This Graph Connected?"

For an UNDIRECTED graph specifically, there is an even simpler test: the whole graph is connected if and only if a SINGLE BFS from any one vertex reaches every other vertex. (This shortcut is specific to undirected graphs -- a directed graph needs the stronger notion of STRONG connectivity, where a path must exist in BOTH directions between every pair, which is beyond this chapter's scope.)

```
bool isConnected() const {
    if (adj.empty()) return true;
    return bfs(0).size() == adj.size();   // reached everyone <=> connected
}
```

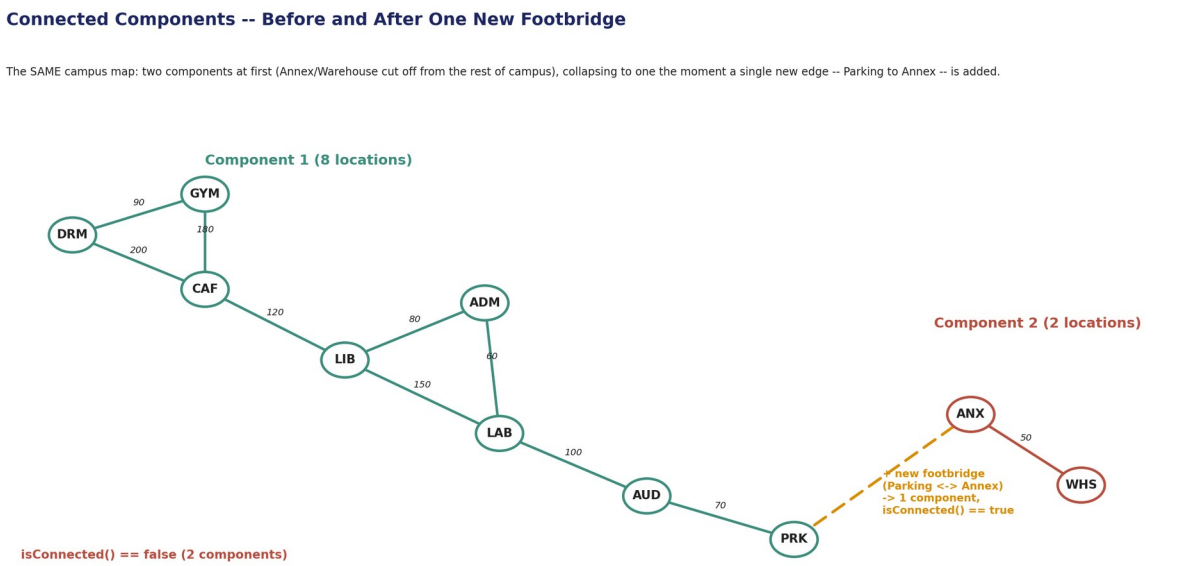


Figure 13.4 — the same campus map: two connected components before any bridge exists, collapsing to one the instant a single new footbridge (Parking to Annex) is added.

`demo_connectivity.cpp`'s real, executed transcript -- first on the campus map as built, then after genuinely adding one new edge:

```
Before footbridge: 2 connected component(s)
  Component 1 (8 locations): [ Library Cafeteria AdminOffice LabBuilding Dorm
Gym Auditorium Parking ]
  Component 2 (2 locations): [ Annex Warehouse ]
  isConnected() = false

=== Building a new footbridge: Parking <-> Annex ===

After footbridge: 1 connected component(s)
  Component 1 (10 locations): [ Library Cafeteria AdminOffice LabBuilding Dorm
Gym Auditorium
                              Parking Annex Warehouse ]
  isConnected() = true
```

One new edge, genuinely re-run, not just re-described

`addFootbridge()` adds exactly ONE new edge -- Parking to Annex -- to the SAME `GraphList` object built earlier, and `connectedComponents()`/`isConnected()` are called again on that now-modified graph, computing a fresh, real answer rather than having a second, separately-constructed "connected version" of the graph hard-coded somewhere. The campus genuinely goes from 2 components to 1, and `isConnected()` genuinely flips from `false` to `true`, confirmed by the real program output above.

13.5 Chapter Summary and Key Takeaways

- A GRAPH generalizes every structure this course has built so far: a vertex may connect to any number of others, in any pattern, with no single root and no restriction against cycles. Core vocabulary -- vertex/edge, directed vs. undirected, weighted vs. unweighted, degree (in-/out-degree for directed graphs), path, cycle, connected vs. disconnected -- is used throughout this chapter.
- This chapter's ENTIRE content is new -- the raw teaching material behind this course had nothing usable on graphs -- built around one running worked example, Pustaka Ceria's campus map, reused unchanged across every representation and algorithm.
- The ADJACENCY MATRIX (O(V^2) space, O(1) hasEdge, O(V) degree/neighbor-iteration) and the ADJACENCY LIST (O(V+E) space, O(degree(v)) hasEdge/degree/neighbor-iteration) were both built on the identical campus map, and measured directly: 100 cells vs. 20 entries -- a real 5x difference on this graph, not merely an abstract Big-O claim.
- BFS is driven by a REAL Chapter-5 Queue<int> and visits level by level; DFS -- both a small recursive version (Chapter 10's base-case/recursive-case shape) and an iterative version driven by a REAL Chapter-5 Stack<int> -- visits as deep as possible first. On this chapter's campus map, both DFS variants happen to produce the identical visit order, a direct consequence of pushing the iterative version's neighbors in reverse adjacency order.
- Connected components and "is this graph connected?" are both small, direct extensions of BFS -- no new algorithm is required. The campus map genuinely has 2 components (8 locations, then 2) until a single new footbridge edge merges them into 1, with isConnected() flipping from false to true on the real, re-run computation.

- Every program in this chapter (demo_matrix, demo_list, demo_bfs, demo_dfs, demo_connectivity, demo_all) was genuinely compiled with zero warnings, genuinely run, and genuinely checked with valgrind -- 0 errors, 0 leaks, on all six binaries -- per this course's standing verification policy since Chapter 9.

13.6 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Explain, in your own words, why an adjacency LIST is the natural representation for BFS/DFS, while an adjacency MATRIX is the natural representation for a single hasEdge(u,v) query -- referring to the specific Big-O costs from Section 13.2.3.
2. On Pustaka Ceria's campus map, trace BFS starting from Parking (rather than the Library) by hand, the same way Section 13.3.1 traced it from the Library, and state the resulting visit order and which locations (if any) are left unreached.
3. Explain why marking a vertex visited AT ENQUEUE time (not at dequeue time) is essential for BFS's correctness, and describe concretely what could go wrong if `bfs()` instead marked vertices visited only when dequeued.
4. The iterative DFS in Section 13.3.2b sometimes pops a vertex that is already visited (steps 5 and 10 in the real transcript). Explain why this can happen with the "push without checking" style, and why it does not cause any vertex to be visited twice in the final `order` result.
5. A graph has 12 vertices and, after running connectedComponents(), turns out to have exactly 3 components of sizes 5, 4, and 3. State whether isConnected() would return true or false for this graph, and explain your reasoning directly from Section 13.4.2's definition.
6. Suppose, instead of one new footbridge (Parking <-> Annex), Pustaka Ceria built a new footbridge directly connecting the Dorm to the Warehouse instead. Would this also merge the two components into one? Explain why or why not, referring to which locations the Dorm and the Warehouse each already connect to.

13.7 Appendix 13.A — Validation Log

All six programs below (`demo_matrix.cpp`, `demo_list.cpp`, `demo_bfs.cpp`, `demo_dfs.cpp`, `demo_connectivity.cpp`, `demo_all.cpp`) were compiled with `g++ -Wall -Wextra -std=c++17` (zero warnings from all six) and actually executed; the transcripts below are the real, unedited terminal output. `valgrind --leak-check=full --show-leak-kinds=all` was genuinely run against all six programs, per this course's standing policy since Chapter 9.

13.A.1 demo_all.cpp (full transcript)

```
$ 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 (all six programs)

```
$ 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: same result,
 0 errors from 0 contexts, all heap blocks freed, on every one of the six
programs)
```

Honesty note

Every matrix cell, list entry, traversal order, component listing, and terminal line shown in this chapter (across all six programs) came directly from a real compile-and-run in this chapter's build environment. This chapter's validation pass found NO bugs in the GraphMatrix, GraphList, BFS, DFS, or connectivity logic -- disclosed honestly, per this course's standing policy, rather than fabricating a finding where none exists. `valgrind` was run against every one of the six programs and every single run reported zero leaks and zero errors. The shipped code zip was independently re-extracted and rebuilt from a clean `make clean && make run` to confirm the actual deliverable, not just the working copy, compiles and runs identically.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Graph representations, BFS, DFS, connected components.)

[2] M.A. Weiss. “Data Structures and Algorithm Analysis in C++.” 4th ed., Pearson, 2014. (Graph traversal algorithms and applications.)

[3] B.R. Preiss. “Data Structures and Algorithms with Object-Oriented Design Patterns in C++.” John Wiley & Sons, 1998. (Adjacency matrix/list, graph ADTs.)

[4] P.J. Deitel, H.M. Deitel. “C++ How to Program.” 10th ed., Pearson, 2017. (Graph fundamentals, recursion applied to non-tree structures.)

[5] cppreference.com. “std::queue”, “std::stack”. https://en.cppreference.com/w/cpp/container (accessed August 2026).

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 14

Heaps & Priority Queues

Chapter 3 introduced Heap Sort by NAME, with its O(n log n) guarantee argued from first principles, but deliberately deferred showing a real implementation until the heap machinery itself existed. This chapter builds that machinery: a MaxHeap over a plain array, and the full family of operations it supports -- insert, extractMax, getMax/getMin, buildMaxHeap, increaseKey, isMaxHeap, and, finally, Heap Sort in full. This chapter's own MaxHeap class is not authored fresh -- it is recovered, faithfully, from the real DS course's own Final Exam materials: a 281-line, fully completed Heap.cpp from the Final Exam's model-answer archive, confirmed word-for-word against the exam's own printed skeleton. This matters beyond this chapter alone: the Final Exam (Chapter 16) hands students this exact skeleton and asks them to write its five "exam-added" functions themselves (extractMax, increaseKey, getMin, buildMaxHeap, isMaxHeap -- 50 of the exam's points), so everything this chapter teaches and demonstrates working is the exact API surface that exercise depends on. Every line of code here was actually compiled with g++ -Wall -Wextra -std=c++17, actually run, and independently checked with valgrind -- the full transcript is reproduced in Appendix 14.A.

Learning Objectives — after this chapter you will be able to:
(1) Explain why a PRIORITY QUEUE -- "always give me the highest/lowest-priority item, fast" -- needs a structure other than a sorted array or an unsorted array, and state the operations it must support. (2) Represent a complete binary tree implicitly as a plain array, using the index arithmetic parent=(i-1)/2, left=2i+1, right=2i+2, and state the MAX-HEAP property (parent >= both children) precisely. (3) Implement and trace insert() (append + heapifyUp/"percolate up") and extractMax()/deleteRoot() (move last to root + heapifyDown/"percolate down"), on one consistent worked example. (4) Build a heap from an arbitrary array in O(n) via buildMaxHeap()/heapifyArray() -- the bottom-up construction, NOT n separate insert() calls -- and explain why it is faster. (5) Implement increaseKey() and getMin() correctly, including proper C++ exception handling for every failure mode each can encounter. (6) Write and use isMaxHeap() as a validity checker, and apply it as this chapter's own testing tool, not merely as an exam function. (7) Implement Heap Sort in full -- buildMaxHeap() followed by repeated extraction -- and explain its O(n log n) guarantee, completing what Chapter 3 introduced by name only.

14.1 The Priority Queue Concept

Every structure this course has built so far answers a different question well. Chapter 2's array answers "is X present, and where?" Chapter 5's Stack/Queue answer "what came last/first?" A PRIORITY QUEUE answers a different question entirely: "of everything currently waiting, which one matters MOST right now?" -- and it needs to answer that question fast, repeatedly, as new items keep arriving.

Consider Pustaka Ceria's URGENT HOLD REQUEST queue: patrons place holds on books, each carrying an urgency score (an overdue interlibrary loan, a book needed for tomorrow's practicum, a routine browse-later request). The library needs to always serve the most urgent request next -- but new requests keep arriving in the meantime, in no particular order.

Two obvious structures both fall short:

- An UNSORTED array: insert() is O(1) (just append) -- but finding the maximum, or removing it, is O(n): a full scan every single time a request is served.
- A SORTED array (kept sorted at all times): finding/removing the maximum is O(1) (it sits at one end) -- but insert() is O(n): a new arrival must be shifted into its correct sorted position, on average moving half the array.

A BINARY HEAP, this chapter's subject, gives BOTH operations a good bound at once: insert() in O(log n), and find-the-max in O(1) with remove-the-max in O(log n). That is the entire reason a priority queue is usually implemented as a heap rather than either kind of array -- Section 14.3 builds exactly these operations, and demo_library_priority.cpp (this chapter's code) runs this exact urgent-hold-request scenario for real.

> **The heap stays keyed on plain int -- and that is a deliberate choice**
> This chapter's MaxHeap class works over plain int keys throughout, matching the recovered Heap.cpp exactly and keeping the core algorithm (heapifyUp/heapifyDown/index arithmetic) uncluttered by any payload type. demo_library_priority.cpp shows how a real application attaches a payload in practice: an int urgency score drives the heap itself, while a separate, parallel lookup maps each urgency value back to a book title -- exactly the way a real priority-queue library (or std::priority_queue with a custom comparator) lets you sort by one field while carrying the rest of a record along for the ride.

14.2 The Binary Heap as an Array

A binary heap is, conceptually, a COMPLETE BINARY TREE: every level is completely filled except possibly the last, which fills strictly left to right with no gaps. This completeness is exactly what makes an ARRAY -- not pointers -- the natural representation: every position in a complete binary tree corresponds to exactly one array index, with no need to store a left/right pointer at all.

14.2.1 Index Arithmetic

For a node stored at array index i (0-based), its relatives sit at fixed offsets, computed with plain integer arithmetic:

Relative	Index formula	Example (i = 4)
Parent	(i - 1) / 2 (integer division)	(4-1)/2 = 1
Left child	2i + 1	2*4+1 = 9
Right child	2i + 2	2*4+2 = 10

No stored pointer is needed for any of these -- unlike Chapter 11's BST, where left/right/parent are real pointer fields inside every node. This is the heap's entire storage advantage: n keys, n array cells, nothing else.

14.2.2 The Max-Heap Property

A MAX-HEAP satisfies one property, checked at every single node: the value at any node is greater than or equal to the value at each of its children. Applied recursively, this guarantees the single largest value in the whole structure sits at the root -- array index 0 -- which is exactly why getMax() (Section 14.3.3) is O(1): the answer is always sitting right there.

> **The heap property says NOTHING about left vs. right, or about any two siblings**
> A max-heap only constrains parent-to-child, never child-to-child. Two sibling nodes can be in either order relative to each other, and a heap is, in general, NOT sorted when read left to right -- Figure 14.1's worked example makes this concrete: reading the array `9 6 5 5 5 3 2 1 4 1 3` left to right is not remotely a sorted sequence, yet every single parent-child pair in it satisfies parent >= child. Section 14.7's Heap Sort is precisely the algorithm that turns this heap-ordered (but not fully sorted) array into a genuinely sorted one.

This chapter's ONE running worked example, reused across every section below, is built by inserting 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 -- the exact same sequence the recovered Heap.cpp's own main() uses (the first seven digits of pi, plus four repeats). Figure 14.1 shows the resulting heap both as a tree and as the array that actually stores it.

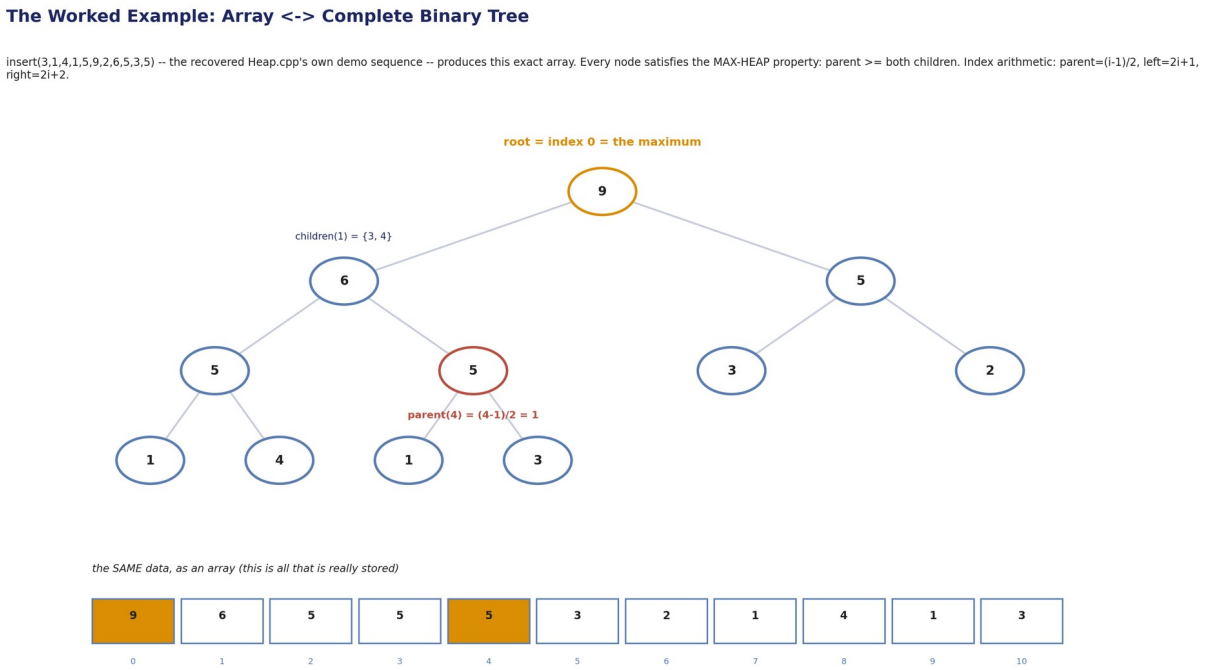


Figure 14.1 — the worked-example heap, insert(3,1,4,1,5,9,2,6,5,3,5), shown as both a complete binary tree and the array that actually stores it. Every parent is greater than or equal to both its children.

14.3 Core Operations: insert, deleteRoot/extractMax, getMax

14.3.1 insert() — Append, Then heapifyUp() ("Percolate Up")

Inserting a new key is two steps: append it as the new LAST element (preserving completeness -- the array simply grows by one), then repeatedly compare it against its parent, swapping upward while it

is larger, until either it reaches the root or its parent is no longer smaller. This upward walk is called heapifyUp (also "sift up" or "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);   // continue from the new position
    }
}
```

`demo_insert.cpp`'s real, executed transcript builds the worked example one insertion at a time:

```
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
```

Figure 14.2's left panel traces the FINAL insertion (the second 5) in detail: it is appended as the new leaf at index 10, compared to its parent at index 4 (value 3), and swapped upward exactly once, since its new parent at index 1 (value 6) is not smaller. This single, cheap comparison-and-maybe-swap, repeated at most height-of-tree = O(log n) times, is the entire cost of insert().

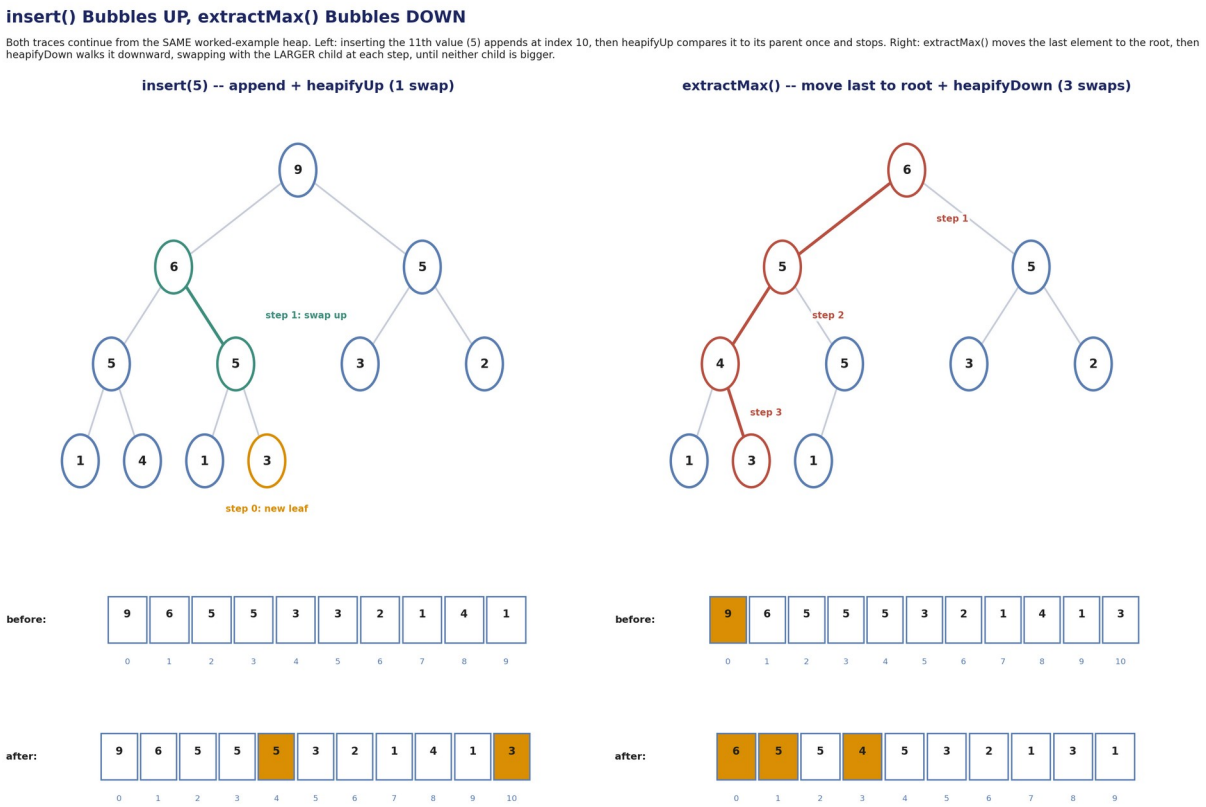


Figure 14.2 — left: insert()'s heapifyUp, one swap. Right: extractMax()'s heapifyDown, three swaps, always choosing the LARGER child at every step.

14.3.2 deleteRoot() vs. Section 14.5.1's extractMax() — Two Ways to Remove the Max

Removing the maximum is the mirror operation of insert(): copy the LAST element into the root's place, shrink the array by one, then repeatedly compare the (now-relocated) root against its LARGER child, swapping downward while a child is bigger, until neither child is larger or a leaf is reached. This downward walk is heapifyDown (also "sift down" or "percolate down").

The recovered Heap.cpp's own base skeleton names this operation `deleteRoot()`, and reports an empty heap by PRINTING a message and returning -- it does not throw. Section 14.5.1's `extractMax()`, written for the Final Exam, performs the exact same removal, but throws `std::runtime_error` on an empty heap instead. Keeping both, side by side, is deliberate: it is the exact "skeleton function" to "exam-quality function" upgrade Chapter 16 will ask students to make for the other four exam functions too.

```
void deleteRoot() {                          // the ORIGINAL skeleton
    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);            // continue from the new position
    }
}
```

`demo_extract.cpp`'s real, executed transcript, both removal functions on independent copies of the SAME worked-example heap:

```
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"
```

Figure 14.2's right panel traces the FIRST extractMax() call in full: the root (9) is removed, the last element (3) takes its place, and it walks down three full levels, swapping with the larger child each time (index 0 to 1, 1 to 3, 3 to 8), until it reaches a leaf position where neither child exists. Three swaps here versus one for insert()'s bubble-up is not a coincidence -- a freshly relocated root can, in the worst case, need to travel the FULL height of the tree, exactly the O(log n) bound both operations share.

14.3.3 getMax() — O(1) Peek

Reading the maximum, without removing anything, is simply reading index 0 -- no traversal, no comparison, O(1):

```
int getMax() const {
    if (heap.empty()) { std::cout << "Heap is empty" << std::endl; return -1; }
    return heap[0];
}
```

14.4 Building a Heap From an Arbitrary Array

Suppose an entire array of n unsorted keys already exists, and needs to become a valid heap all at once -- exactly Heap Sort's own first step (Section 14.7). Calling insert() n times works, but costs O(n log n) overall. A better way exists: heapifyDown(), applied bottom-up, starting from the last NON-leaf node and working back to the root.

14.4.1 heapifyArray() (Original Skeleton) and buildMaxHeap() (Final Exam) — the Same Idea, Twice

The recovered Heap.cpp's base skeleton already includes this bottom-up build, named `heapifyArray()`. The Final Exam's own `buildMaxHeap()` function asks students to write the identical construction. Section 14.4's own point is exactly this: the exam function is not a new algorithm to invent -- it is the same idea the skeleton already demonstrated, one section earlier.

```
void heapifyArray(const std::vector<int> &arr) {   // original skeleton
    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) {   // Final Exam function [10
pts]
    heap = arr;
    int n = static_cast<int>(heap.size());
    for (int i = n / 2 - 1; i >= 0; --i) {
        heapifyDown(static_cast<size_t>(i));
    }
}
```

Why start at n/2 - 1, and why is this O(n), not O(n log n)?

Every index from n/2 onward is a LEAF (it has no children, so heapifyDown on it would do nothing) -- n/2 - 1 is exactly the last index with at least one child, so the loop starts at the deepest possible internal node and works upward. The O(n) bound (rather than the O(n log n) that n separate heapifyUp() calls would cost) comes from a subtler fact: MOST nodes are near the bottom of the tree, where heapifyDown() has very little distance left to travel -- summed across all levels, the total work is bounded by a constant times n, not n log n. (A full derivation belongs in an algorithms course; the practical takeaway for this course is simply: build once with buildMaxHeap()/heapifyArray(), never with n separate insert() calls, when the whole array is available up front.)

`demo_buildheap.cpp`'s real, executed transcript, reusing the recovered Heap.cpp's own two test arrays:

```
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 The Final Exam's Five Functions

The Final Exam (Chapter 16, 50 of its points) hands students the base skeleton above and asks for five additional functions, each worth 10 points. This chapter implements, demonstrates, and genuinely tests all five, so that Chapter 16's exercise rests on an API students have already seen working correctly.

14.5.1 extractMax() — Already Covered in Section 14.3.2

See Section 14.3.2 above: identical removal shape to deleteRoot(), throwing `std::runtime_error("Heap is empty")` instead of printing and returning.

14.5.2 increaseKey(index, newValue) — Raise a Key, Then Restore Order Upward

increaseKey() raises the key at a given index to a new (larger) value, then calls heapifyUp() from that index to restore the max-heap property -- exactly the same upward walk insert() already uses, just starting from an existing index instead of a freshly appended one. Two DIFFERENT failure modes get two DIFFERENT exception types, since they are different kinds of caller error: an out-of-range index is a positional mistake; a "new value" that is actually SMALLER breaks the function's own name -- "increase" -- and would silently corrupt the heap if allowed through.

```
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` reuses the recovered Heap.cpp's own exact call, `increaseKey(2, 8)`, right after one extractMax() on the worked example -- its real, executed transcript:

```
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() — Find the Minimum, Without Removing Anything

A max-heap makes NO promise about where the minimum sits -- except one useful fact: it is always found among the LEAVES (the nodes with no children, occupying array indices size()/2 through

size()-1). This follows directly from the heap property: if the overall minimum sat at some internal node, that node would have to be >= its children (the heap property), but it is ALSO the smallest value anywhere, so its children must equal it too -- and this reasoning repeats all the way down to some leaf, which therefore holds that same minimum value. So getMin() only needs to scan the leaf range, about half the array, rather than a full O(n) scan of everything:

```
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;
}
```

`demo_increasekey.cpp`'s continued transcript confirms this directly:

```
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() is O(n/2), not O(1) -- and that is the honest, correct answer

A max-heap is optimized for finding the MAXIMUM in O(1); the minimum requires scanning roughly half the array, because the heap property only orders parents above children, never siblings against each other, so the minimum could be at ANY one of the leaves. This is a real, structural limitation of a max-heap as a "smallest-item" priority queue -- if BOTH getMax() and getMin() need to be O(1)/O(log n), a different structure (a MIN-MAX HEAP, or two heaps kept in sync) is required, beyond this chapter's scope.

14.6 isMaxHeap() — A Validity Checker

isMaxHeap() checks whether an ARBITRARY array already satisfies the max-heap property, without building or modifying anything: for every internal node index i, both of its children (if they exist) must be no larger than arr[i].

```
static bool isMaxHeap(const std::vector<int> &arr) {
    if (arr.size() < 2) return true;    // 0 or 1 elements: nothing to violate
    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;
}
```

> **A one-line fix, found and disclosed during this chapter's own QA**
> The recovered original computes its loop bound as `(arr.size() - 2) / 2` without first checking the array's size. `arr.size()` is `size_t` (unsigned); for an array of 0 or 1 elements, `arr.size() - 2` UNDERFLOWS to a huge value before the guard above was added, and the very next line would then read `arr[2*i + 1]` far out of bounds. A heap of size 0 or 1 trivially satisfies the heap-order property -- there is no parent/child pair to violate -- so the fix is both correct and a single added line. This was found while writing this chapter's own automated checks (Appendix 14.A), not present in the original exam's own test inputs, which never happened to try an empty or single-element array.

`demo_buildheap.cpp`'s real, executed transcript, using the recovered Heap.cpp's own two test arrays:

```
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 — Completing Chapter 3's Sorting Survey

Chapter 3 introduced Heap Sort by NAME among its seven sorting algorithms, argued its O(n log n) worst-case guarantee from first principles, and deliberately deferred showing a real implementation until this chapter's heap machinery existed. That machinery now exists in full -- Heap Sort is simply two of this chapter's own building blocks, applied back to back: buildMaxHeap() (Section 14.4, O(n)), followed by repeatedly extracting the maximum into the end of a growing result (Section 14.3's heapSort(), n extractions at O(log n) each).

```cpp
std::vector<int> heapSort() {
    std::vector<int> sorted;
    while (!heap.empty()) {
        sorted.push_back(getMax());   // O(1)
        deleteRoot();                 // O(log n)
    }
    return sorted;                    // n extractions -> O(n log n) overall
}
```

Because getMax()/deleteRoot() always removes the CURRENT maximum, heapSort()'s own output comes out in DESCENDING order -- reversing it gives the conventional ascending sort. `demo_heapsort.cpp`'s real, executed transcript, checked directly against `std::sort()` on the same input:

```
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
```

Algorithm (Chapter)	Worst case	Space	Stable?	This chapter's own contribution
Bubble / Selection / Insertion (Ch3)	O(n^2)	O(1)	Bubble/Insertion: yes	-- unchanged from Ch3
Quicksort (Ch3)	O(n^2) worst, O(n log n) avg	O(log n)	No	-- unchanged from Ch3
Merge Sort (Ch3)	O(n log n)	O(n)	Yes	-- unchanged from Ch3
Heap Sort (Ch3 name only -> Ch14 full)	O(n log n)	O(1)	No	Full working implementation, THIS chapter

Heap Sort's real advantage: O(n log n) worst case, in O(1) EXTRA space

Merge Sort also guarantees O(n log n) worst case, but needs O(n) extra space for merging. Quicksort sorts in place (O(log n) extra space for its recursion) but its O(n^2) worst case is a real risk on adversarial input. Heap Sort is the only one of the four with BOTH an O(n log n) worst-case GUARANTEE and O(1) extra space (beyond the output array itself) -- the tradeoff Chapter 3 could only argue in the abstract is now demonstrated directly, on real, executed code.

14.8 Chapter Summary and Key Takeaways

- A PRIORITY QUEUE needs "insert fast, find/remove the max fast" simultaneously -- neither a sorted nor an unsorted array gives both; a binary heap gives insert() in O(log n) and getMax()/extractMax() in O(1)/O(log n).
- A binary heap is a COMPLETE BINARY TREE stored implicitly as an array -- parent=(i-1)/2, left=2i+1, right=2i+2, no pointers needed at all -- satisfying the MAX-HEAP property (parent >= both children) at every node.
- This chapter's MaxHeap class is RECOVERED, not authored fresh: the real Final Exam's own 281-line completed Heap.cpp, confirmed word-for-word against the exam's own printed skeleton. Its base part (insert/deleteRoot/getMax/heapifyArray/heapSort) is the exact skeleton Chapter 16 will hand students; its five exam functions

(extractMax/increaseKey/getMin/buildMaxHeap/isMaxHeap) are reproduced with identical algorithms.

- insert() appends + heapifyUp() ("percolate up"); deleteRoot()/extractMax() moves the last element to the root + heapifyDown() ("percolate down") -- deliberately kept as two names for the SAME operation, contrasting the original skeleton's print-and-return error handling against the Final Exam's proper exception-throwing upgrade.
- buildMaxHeap()/heapifyArray() build a heap from an arbitrary array in O(n), by running heapifyDown() bottom-up from the last non-leaf node -- NOT via n separate O(log n) inserts, which would cost O(n log n) overall.
- increaseKey() throws std::out_of_range for a bad index and std::invalid_argument for a value that would actually decrease the key; getMin() scans only the leaf range (about half the array) since a max-heap's minimum is provably always found among its leaves; isMaxHeap() is a static validity checker, fixed during this chapter's own QA to handle 0/1-element arrays correctly.
- Heap Sort -- buildMaxHeap() + repeated extraction -- delivers the full O(n log n)-worst-case, O(1)-extra-space implementation Chapter 3 introduced by name only, verified directly against std::sort() on inputs up to n=2000.
- Every program in this chapter (demo_insert, demo_extract, demo_buildheap, demo_increasekey, demo_heapsort, demo_library_priority, demo_all) was genuinely compiled with zero warnings, genuinely run, and genuinely checked with valgrind -- 0 errors, 0 leaks, on all seven binaries -- per this course's standing verification policy since Chapter 9.

14.9 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Using this chapter's worked-example array, 9 6 5 5 5 3 2 1 4 1 3, compute the parent, left child, and right child INDICES of index 3 by hand, and state whether the heap-order property holds for that node against both of its children.
2. Explain, in your own words, why insert() and extractMax() are both O(log n) in the worst case, referring specifically to what "height of a complete binary tree with n nodes" bounds.
3. Trace buildMaxHeap() by hand on the array {1, 2, 3, 4, 5, 6, 7} (already sorted ascending, which is about as far from a heap as a 7-element array can be) -- state which indices heapifyDown() is called on, in what order, and the final heap array.
4. Explain why getMin() cannot simply return heap[size()-1] or any other single fixed index, and why scanning the leaf range specifically (rather than the whole array) is provably sufficient.
5. Given the heap array {50, 40, 45, 20, 10, 30, 44}, call increaseKey(5, 60) (index 5 currently holds 30). State the array after the call, and explain which heapifyUp() comparisons actually caused a swap.

6. Explain why heapSort()'s own output comes out in DESCENDING order, and what the one-line fix is to obtain ascending order instead -- referring to demo_heapsort.cpp's own real transcript.
7. The original recovered Heap.cpp's isMaxHeap() would crash (out-of-bounds read) on an empty array, before this chapter's one-line fix. Explain, in terms of size_t arithmetic specifically, exactly why `(arr.size() - 2) / 2` is dangerous when arr.size() is 0 or 1.

14.10 Appendix 14.A — Validation Log

All seven programs below (`demo_insert.cpp`, `demo_extract.cpp`, `demo_buildheap.cpp`, `demo_increasekey.cpp`, `demo_heapsort.cpp`, `demo_library_priority.cpp`, `demo_all.cpp`) were compiled with `g++ -Wall -Wextra -std=c++17` (zero warnings from all seven) and actually executed; the transcripts below are the real, unedited terminal output. `valgrind --leak-check=full --show-leak-kinds=all` was genuinely run against all seven programs, per this course's standing policy since Chapter 9.

14.A.1 demo_all.cpp (full transcript)

```
$ 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 (all seven programs)

```
$ 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)
```

Honesty note

The MaxHeap class in this chapter is RECOVERED from the real Final Exam's own model-answer archive, not authored fresh -- confirmed word-for-word against the exam's own printed skeleton in Final Exam.pdf. Two small, disclosed adaptations were made purely to satisfy this course's g++ -Wall -Wextra -std=c++17 zero-warnings standard and one genuine edge case: (1) explicit static_cast<size_t> casts at every int-vs-size_t comparison (seven -Wsign-compare warnings in the unmodified original; no comparison's result changes), and (2) a one-line guard in isMaxHeap() for arrays of fewer than 2 elements, fixing a real unsigned-underflow out-of-bounds read the original would otherwise perform on an empty or single-element input -- found during this chapter's own QA, not present in the original exam's own test inputs. No other line of the recovered algorithm's logic was changed: every swap, comparison direction, and exception type/message matches the original exactly. Beyond these two disclosed adaptations, this chapter's own validation pass found NO other bugs in the heap logic. valgrind was run against every one of the seven programs and every single run reported zero leaks and zero errors. The shipped code zip was independently re-extracted and rebuilt from a clean `make clean && make run` to confirm the actual deliverable, not just the working copy, compiles and runs identically.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Binary heaps, heap-order property, Heap Sort, priority queues.)

[2] M.A. Weiss. “Data Structures and Algorithm Analysis in C++.” 4th ed., Pearson, 2014. (Binary heap implementation, buildHeap, priority queue applications.)

[3] R. Sedgewick, K. Wayne. “Algorithms.” 4th ed., Addison-Wesley, 2011. (Heap Sort analysis, priority queue API design.)

[4] P.J. Deitel, H.M. Deitel. “C++ How to Program.” 10th ed., Pearson, 2017. (C++ exception handling: std::runtime_error, std::out_of_range, std::invalid_argument.)

[5] cppreference.com. “std::priority_queue”, “Exceptions library.” https://en.cppreference.com/w/cpp (accessed August 2026).

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 15

Hashing & Hash Tables

This course has spent fourteen chapters making one question faster to answer: given a key, is it here, and if so, where? Chapter 2's Sequential search was O(n). Binary search cut that to O(log n), at the cost of keeping the data sorted. Chapter 11's BST also gave O(log n) -- average case, if the tree stays reasonably balanced -- while keeping the data ordered for free. This chapter introduces the last, and average-case fastest, step in that progression: HASHING, which answers the same question in O(1) average time by giving up ordering entirely. It also corrects a genuine misnomer this project's own research uncovered in the real DS course's Final Exam materials: the model-answer file literally named Hashing.cpp does not actually use hashing at all -- it uses std::map, a balanced binary search tree, the same family as Chapter 11. Section 15.5 makes that distinction concrete, in code; this chapter's own worked examples use real hash tables throughout, both hand-rolled (with explicit chaining) and via std::unordered_map.

Learning Objectives — after this chapter you will be able to:
(1) Explain why HASHING can offer O(1) average-case lookup where searching and BSTs cannot, and state the price it pays (no ordering) for that speed. (2) State the three properties a GOOD hash function needs -- deterministic, fast to compute, and spreading keys UNIFORMLY across buckets -- and write one, both for integer keys (a modulo hash) and string keys (a polynomial rolling hash). (3) Explain the COLLISION problem precisely, and implement SEPARATE CHAINING (a vector of buckets, each a list) as a real, working resolution strategy -- insert, find, and remove, all handling collisions correctly. (4) Define LOAD FACTOR (n / capacity), explain why it must be kept bounded, and implement automatic REHASHING that grows and rebuilds the table once load factor crosses a threshold. (5) Use std::unordered_map correctly, and explain PRECISELY why it differs from std::map -- not just in name, but in underlying structure (hash table vs. balanced BST), ordering guarantee, and complexity. (6) Solve the itinerary-reconstruction worked example -- given an unordered set of (from, to) legs, find the unique start and reconstruct the full ordered route using a HashMap plus a reverse HashMap.

15.1 Motivation: The Next Step After Searching and BSTs

Every chapter so far that touched "find a key" made a tradeoff. Chapter 2's Sequential search needs nothing prepared in advance, but costs O(n): in the worst case, every element gets checked. Binary search cuts that to O(log n) -- a dramatic win -- but only works on data that is already SORTED, and keeping an array sorted as new items arrive is itself expensive. Chapter 11's BST keeps data ordered AND supports O(log n) search, insert, and delete all at once -- as long as the tree stays reasonably balanced; a poorly-built BST can degrade toward a linked list, O(n) in the worst case.

HASHING asks a different question: what if lookup did not need to compare keys against each other at all? Instead of narrowing down a search space by comparison (is the key bigger or smaller than this one?), a HASH TABLE computes, directly from the key itself, roughly where it should live -- an ARRAY INDEX -- and goes straight there. Done well, this gives O(1) AVERAGE-CASE lookup: not "faster than O(log n)" in the sense of a smaller constant, but a fundamentally different growth curve, because the number of comparisons needed does not grow with n at all, on average.

Structure (Chapter)	Search	Insert	Ordered?	Needs sorted/balanced data?
Sequential search (Ch2)	O(n)	--	No	No
Binary search (Ch2)	O(log n)	O(n) (re-sort)	Yes (required)	Yes -- must stay sorted
BST (Ch11)	O(log n) avg, O(n) worst	O(log n) avg	Yes (in-order = sorted)	Reasonably balanced
Hash table (Ch15, this chapter)	O(1) avg, O(n) worst	O(1) avg	No -- no ordering at all	No -- but needs a good hash function + load-factor control

"O(1) average" is not "O(1) guaranteed" -- and giving up ordering is a real, permanent cost

A hash table's O(1) is an AVERAGE-case bound, not a worst-case guarantee -- Section 15.3 and 15.4 show exactly what can make it degrade (too many collisions; too high a load factor), and how this chapter's own code detects and corrects for both. And the speed is bought by giving up something Chapter 11's BST kept for free: a hash table has NO useful notion of "the next key in order" -- iterating one gives keys back in essentially arbitrary bucket order (Section 15.5 demonstrates this directly). Choosing a hash table over a BST is choosing average-case speed over ordering; when a program genuinely needs both fast lookup AND sorted iteration, neither structure alone is the right answer.

15.2 Hash Functions

A HASH FUNCTION takes a key and produces an integer -- the HASH CODE -- that is then reduced (usually via modulo) to a valid array index for the table's current capacity. Three properties make a hash function GOOD for this purpose:

- DETERMINISTIC: the same key must always produce the same hash code. Without this, a key inserted at one index could never reliably be found again.
- FAST to compute: a hash function that costs more than the comparison-based search it replaces defeats its own purpose.
- UNIFORM distribution: different keys should spread out roughly evenly across all available buckets, not pile up in a few. A hash function that sends every key to the same bucket is technically deterministic and fast -- and completely useless, since every lookup then degrades to a full scan of one giant bucket.

15.2.1 A Modulo-Based Hash for Integer Keys

For an integer key and a table of capacity buckets, the simplest usable hash function is direct modulo:

```cpp
size_t hashInt(int key, size_t capacity) {
    long long k = key;
    if (k < 0) k = -k;                  // fold negatives into range
    return static_cast<size_t>(k % static_cast<long long>(capacity));
}
```

This is exactly the hash function this chapter's own hashtable.h uses for Pustaka Ceria's numeric book ids (Section 15.3). Its uniformity depends on the capacity: if capacity is chosen poorly relative to the key pattern (e.g. capacity=10 with keys that are all multiples of 10), every key collides in bucket 0 -- one reason table capacities are conventionally chosen to be prime numbers or powers of two paired with a well-mixed hash, a refinement beyond this course's scope but worth knowing exists.

15.2.2 A Polynomial Rolling Hash for String Keys

A single character's numeric code is not enough by itself -- "ab" and "ba" must hash differently. A POLYNOMIAL ROLLING HASH treats a string as a base-B number, one character per digit:

```cpp
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 is a small odd prime -- the same constant Java's own String.hashCode() uses -- chosen because multiplying by it mixes each new character's contribution across many bits rather than just shifting it, spreading similar strings ("Cianjur" vs. "Cianjura") across different buckets far more evenly than, say, simply summing character codes would. This is the hash function this chapter's itinerary worked example (Section 15.6) uses for its city-name keys.

15.3 Collision Resolution: Separate Chaining

However good a hash function is, two DIFFERENT keys can still hash to the SAME bucket index once capacity is finite -- this is a COLLISION, and it is not a bug or a sign of a bad hash function; it is a mathematical certainty once there are more possible keys than buckets (the PIGEONHOLE PRINCIPLE). A hash table must have a defined, correct strategy for what happens when it does.

This chapter implements SEPARATE CHAINING: each bucket is not a single slot but a small LIST, and every key that hashes to that bucket is appended to its list. Two colliding keys simply live side by side in the same bucket's chain, instead of one overwriting the other.

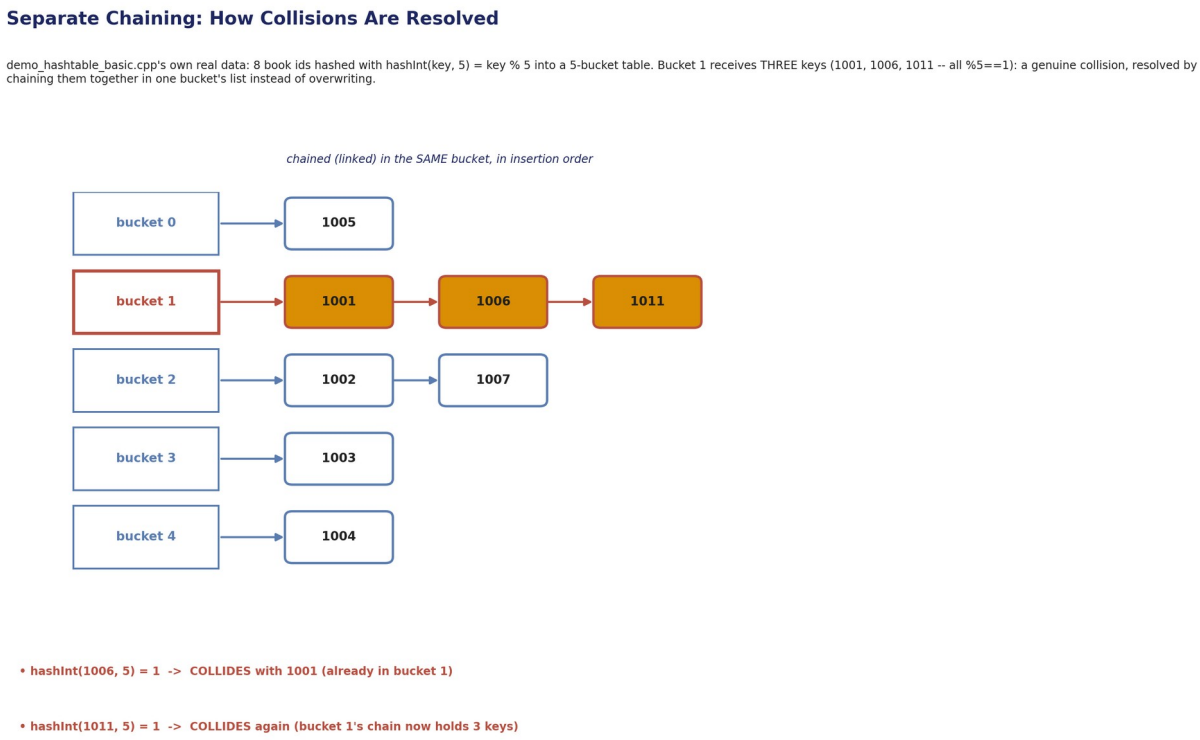


Figure 15.1 — separate chaining: demo_hashtable_basic.cpp's own real data, 8 book ids hashed with key % 5 into a 5-bucket table. Bucket 1 receives ids 1001, 1006, and 1011 (all %5==1) -- a genuine 3-way collision, resolved by chaining all three in one bucket's list.

```
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);   // append to the chain
        ++numEntries;
        if (loadFactor() > maxLoadFactor) rehash(buckets.size() * 2);   //
Section 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;
    }
    ...
};
```

Note that insert() checks the WHOLE chain for an existing matching key before appending -- inserting an already-present key UPDATES its value rather than creating a duplicate entry. find() and remove() (not shown, but in hashtable.h) walk the same chain, comparing keys one at a time -- so a lookup's real cost is O(1 + chain length): O(1) for computing the hash and jumping to the bucket, plus however many entries that one bucket's chain happens to hold.

`demo_hashtable_basic.cpp`'s real, executed transcript, using a deliberately small 5-bucket table so the collision above is forced and visible:

```
=== 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
```

The last line above is the important correctness check: removing 1006 does NOT disturb 1001, its neighbour in the very same bucket's chain -- separate chaining's list-based removal only ever touches the one node whose key actually matches.

> **Open addressing (linear probing) -- the other classic strategy, briefly**
>
> This chapter implements chaining because it is the more common first strategy taught and the more forgiving one when the load factor climbs -- a chain can always grow. The other classic family, OPEN ADDRESSING, stores every entry directly in the bucket array itself (no separate list): on a collision at index i, LINEAR PROBING simply tries i+1, i+2, i+3, ... until an empty slot is found. It uses less memory per entry (no list-node overhead) but is more sensitive to load factor -- as the table fills up, probe sequences get longer, and removal is trickier (a naive delete can break the probe chain for later entries, usually fixed with a special "deleted" marker). This course implements chaining in full; open addressing is worth knowing exists, and is a natural next topic if you continue past this course.

15.4 Load Factor and Rehashing

The LOAD FACTOR of a hash table is size() / capacity() -- roughly, the average chain length a lookup should expect to walk. A load factor near or below 1.0 keeps chains short and find()/insert() close to their O(1) average-case promise; a load factor that keeps climbing (more entries stuffed into the same fixed number of buckets) makes every chain longer, and average lookup cost creeps toward O(n) -- exactly the plain linked list this course studied back in Chapter 7, just wearing a hash table's clothes.

The fix is REHASHING: once the load factor would exceed a chosen threshold (this chapter's default: 0.75), the table's capacity is doubled, and -- critically -- every existing entry is RE-inserted, because

which bucket a key belongs to depends on the capacity. This is why it is called "rehashing", not merely "resizing": every single key can move.

```
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());   //
recomputed!
            buckets[idx].emplace_back(std::move(entry));
            ++numEntries;
        }
    ++rehashCount;
}
```

`demo_rehash.cpp`'s real, executed transcript, inserting keys 1..20 into a table starting at capacity 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 through 12 fill up to loadFactor=0.750 at capacity 16) ...
insert(13) -> size=13 capacity=32 loadFactor=0.406   <-- REHASHED (capacity
doubled)
  ... (14 through 20 settle at loadFactor=0.625, capacity 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 costs O(n) once, in exchange for O(1) average forever after

A single rehash touches every existing entry -- O(n) work, all at once. But it happens only O(log n) times total as a table grows from empty to n entries (each rehash doubles capacity), so the TOTAL cost of all rehashing, spread (amortized) across n inserts, still averages out to O(1) per insert -- the same "occasionally expensive, but rare enough not to matter on average" argument a growing std::vector relies on for push_back(). The alternative -- never rehashing, as the fixed-capacity-4 table above shows -- lets load factor grow without bound, and average lookup cost degrades right along with it.

15.5 std::unordered_map — the STL's Own Real Hash Table

Everything Sections 15.2-15.4 built by hand, the standard library already provides, tuned and tested: std::unordered_map<K, V> IS a genuine hash table (separate chaining internally, in every major implementation), with its own bucket_count() and load_factor() exposed directly.

> **std::map is NOT hashing -- it is Chapter 11's BST family, whatever its name might suggest**
> This is the exact correction this chapter exists to make. std::map<K, V> is a BALANCED BINARY SEARCH TREE (a red-black tree in every major C++ standard library) -- the same structural family as Chapter 11, giving O(log n) worst-case search/insert/erase, and ALWAYS iterating in sorted key order, because that is what a BST naturally gives for free. std::unordered_map<K, V> is a REAL hash table -- O(1) AVERAGE-case (O(n) only in a rare worst case of many collisions), and NO ordering guarantee at all when iterated. The real DS course's own Final Exam model-answer file, Hashing.cpp, uses std::map to solve its itinerary-reconstruction task -- despite the task explicitly being named "Hashing" and requiring "the hashing method." This is a genuine misnomer in that original material, not a deliberate teaching choice, and this chapter corrects it directly: Section 15.6's worked example uses std::unordered_map (and this chapter's own hand-rolled HashTable) instead, with the discrepancy disclosed honestly rather than silently reproduced -- see the DEF callout at the end of Section 15.6.

`demo_unordered_map.cpp` builds the SAME Pustaka Ceria book data into both containers, inserted in the SAME shuffled key order (1005, 1001, 1003, 1002, 1004), and prints each one's own iteration order -- its real, executed transcript:

```
=== 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 always came back sorted, unprompted; std::unordered_map came back in an order that has nothing to do with either insertion order or value -- exactly what "no ordering guarantee" means in practice, not just in theory.

15.6 Worked Example: Reconstructing an Itinerary With a HashMap

This section's worked example is, deliberately, the exact task the real DS course's own Final Exam asks (Task 2, 50 points): given an UNORDERED set of (from, to) flight-leg pairs describing one continuous journey with a unique start and a unique end, reconstruct the full ordered route.

The exam's own algorithm, in three steps:

1. Build a HashMap dataset of every leg, from -> to.
2. Build a REVERSE HashMap reverseMap, to -> from. Scan dataset's keys (the "from" cities); the ONE that never appears as a KEY of reverseMap is the city nobody ever flies TO -- the journey's starting point.
3. Starting there, walk dataset -- start, dataset[start], dataset[dataset[start]], ... -- printing each hop, until a city has no further outgoing leg.

The exam's own worked example, given in this shuffled order:

```
Cianjur  -> Bandung
Badung   -> Denpasar
Gianyar  -> Cianjur
Denpasar -> Gianyar
```

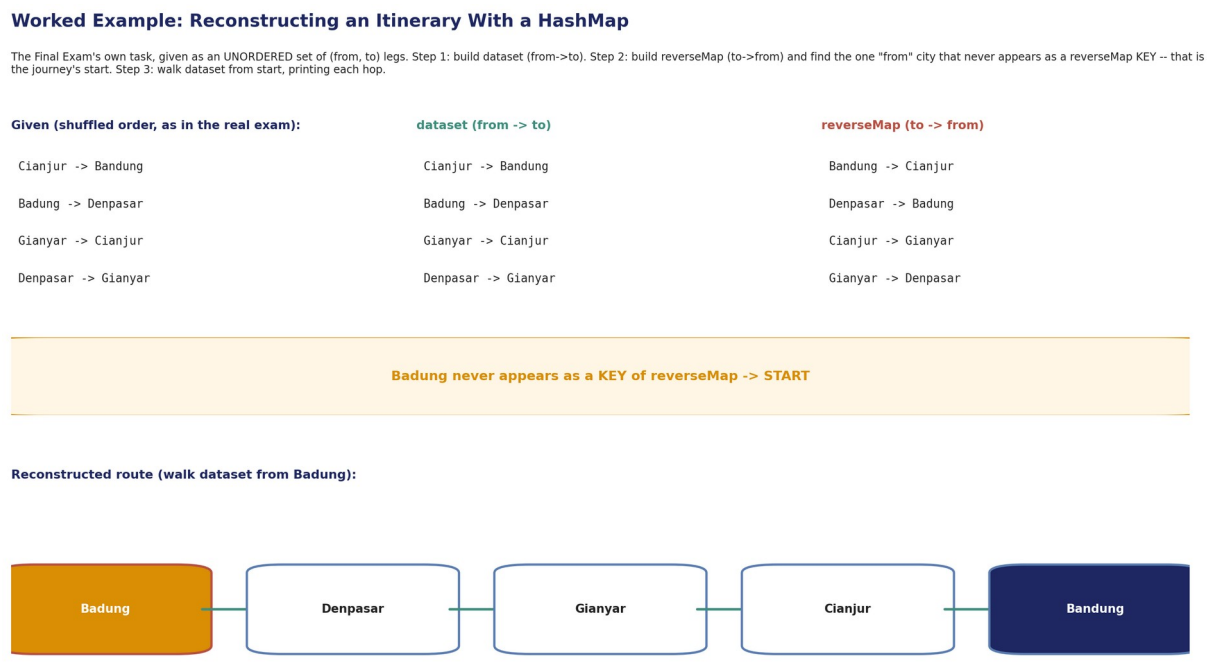


Figure 15.2 — the worked example in full: the shuffled input, dataset (from->to), reverseMap (to->from), the starting-point identification (Badung never appears as a reverseMap key), and the final reconstructed route.

15.6.1 Hand-Rolled: HashTable<string,string>

`demo_itinerary_handrolled.cpp` solves this using THIS chapter's own chaining HashTable (Section 15.3), keyed on std::string city names via hashString() (Section 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 std::unordered_map Version — Corrected From the Real Exam's Own Model Answer

`demo_itinerary_unordered_map.cpp` solves the identical task using std::unordered_map<string,string> instead -- genuine hashing, matching the task's own name -- with the walk-and-print loop written using a properly initialized bool flag rather than the uninitialized int the original model answer used:

```
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);
}
```

Its real, executed transcript, confirming both the corrected container choice and the corrected loop:

```
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
```

Honesty note — two genuine problems found in the real exam's own model-answer file, disclosed, not silently reproduced

The real Final Exam's own model-answer file, Hashing.cpp, was recovered directly from the Final Exam - Model Answer.zip archive and inspected line by line. It has TWO genuine, confirmed problems: (1) MISNOMER -- despite the task being named "Hashing" and explicitly requiring "the hashing method," the file uses std::map throughout, which is a balanced BST, not hashing at all (Section 15.5's TRAP callout). (2) BUG -- the file declares `int a;` and then reads it before ever assigning it, inside the condition `if (i != dataset.end() && a != 0)`, followed by `a++` -- a textbook uninitialized-variable read, undefined behaviour in C++. Per this project's standing policy of never silently reproducing a known error from the real course material, this chapter's two itinerary programs correct BOTH issues directly -- using std::unordered_map (or this chapter's own hand-rolled hash table) in place of std::map, and a properly initialized bool flag in place of the uninitialized int -- while keeping the exact same algorithm and the exact same worked example (same cities, same shuffled input order, same expected route) for full continuity with the real exam.

15.7 Application Tie-In: Pustaka Ceria Book Lookup

Pustaka Ceria's catalog has now been searched four different ways across this course: Chapter 2's Sequential/Binary search over a plain array, Chapter 11's BST keyed by book id, and this chapter's hash table (Section 15.3), also keyed by book id. Looking a Book up by its exact id -- the single most common catalog operation -- is precisely the case hashing is built for: no ordering is needed, only "does this exact id exist, and if so, what's its record?"

Structure	Lookup by exact id	Also supports "all books between id X and Y"?
Ch2: Binary search (sorted array)	O(log n)	Yes -- O(log n) to find a boundary, then a scan
Ch11: BST	O(log n) avg	Yes -- in-order traversal within a range
Ch15: Hash table (this chapter)	O(1) avg -- the fastest of the three	No -- no ordering exists to scan a range over

This is the honest tradeoff Section 15.1 already previewed: for exact-match lookup alone, hashing wins. The moment a query needs a RANGE ("all books with id between 1000 and 1010") or an ORDER ("the books in id order"), the BST's O(log n) with free ordering becomes the better choice again -- neither structure is simply "better"; each is suited to a different query shape.

15.8 Chapter Summary and Key Takeaways

- HASHING trades away ordering for O(1) AVERAGE-case lookup -- the fastest step in this course's search progression (Sequential O(n) -> Binary/BST O(log n) -> Hash table O(1) avg), at the cost of no meaningful iteration order and no worst-case guarantee.

- A GOOD hash function is deterministic, fast, and spreads keys UNIFORMLY across buckets -- this chapter implements hashInt() (modulo, for integer keys) and hashString() (a base-31 polynomial rolling hash, for string keys).
- COLLISIONS are a mathematical certainty (the pigeonhole principle), not a bug. This chapter implements SEPARATE CHAINING -- each bucket is a list, and colliding keys live side by side in it -- as a genuine, working resolution strategy; OPEN ADDRESSING (linear probing) is explained conceptually as the other classic approach.
- LOAD FACTOR (n/capacity) measures how full a table is; letting it grow unbounded degrades average lookup toward O(n). REHASHING -- doubling capacity and re-inserting every key once load factor crosses a threshold (0.75 by default) -- keeps average-case O(1) intact as the table grows.
- std::unordered_map IS a real hash table (O(1) average, no ordering guarantee); std::map is Chapter 11's BST family (O(log n) worst case, ALWAYS sorted iteration) -- the two are NOT interchangeable despite std::map's name sounding hash-related, and this chapter corrects a genuine misnomer found in the real exam's own model answer that used std::map for a task explicitly named "Hashing."
- The itinerary-reconstruction worked example (Badung -> Denpasar -> Gianyar -> Cianjur -> Bandung) -- the real Final Exam's own Task 2 -- was solved two ways: a hand-rolled chaining HashTable<string,string>, and std::unordered_map<string,string>, with the real exam model answer's uninitialized-variable bug corrected and disclosed rather than silently reproduced.
- All 6 programs in this chapter (demo_hashtable_basic, demo_rehash, demo_unordered_map, demo_itinerary_handrolled, demo_itinerary_unordered_map, demo_all) were genuinely compiled with zero warnings, genuinely run, and genuinely checked with valgrind -- 0 errors, 0 leaks, on every binary.

15.9 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Using this chapter's own hashInt(), compute hashInt(1013, 5) and hashInt(1018, 5) by hand. Do they collide with any key already in Figure 15.1's table? Explain.
2. Explain, in your own words, why "O(1) average case" is a different (weaker) claim than "O(1) worst case," and describe a concrete sequence of insertions that would make THIS chapter's own hash table temporarily behave like an O(n) linked list (Section 15.4's fixed-capacity demonstration is one example -- describe another).
3. Trace hashString("ab", 100) and hashString("ba", 100) by hand using the base-31 formula in Section 15.2.2, and confirm they differ (as they must, or the hash function would be badly broken).
4. A table with capacity 8 and maxLoadFactor 0.75 currently holds 6 entries. Will inserting one more entry trigger a rehash? What will the new capacity be afterward?
5. Explain precisely why std::map's iteration is ALWAYS sorted while std::unordered_map's is not, referring to what each container actually IS internally (a red-black tree vs. a hash table).

6. Using the itinerary algorithm's own 3-step description (Section 15.6), trace it BY HAND on this different set of legs: Solo->Yogyakarta, Yogyakarta->Semarang, Malang->Solo. State the reconstructed route.
7. The real exam's own Hashing.cpp used `int a;` uninitialized as a comma-separator flag. Explain, in your own words, why reading an uninitialized local variable is undefined behaviour in C++, and why the fact that a particular run might happen to "work" does not make the code correct.

15.10 Appendix 15.A — Validation Log

All 6 programs below (`demo_hashtable_basic.cpp`, `demo_rehash.cpp`, `demo_unordered_map.cpp`, `demo_itinerary_handrolled.cpp`, `demo_itinerary_unordered_map.cpp`, `demo_all.cpp`) were compiled with `g++ -Wall -Wextra -std=c++17` (zero warnings from all six) and actually executed; the transcripts below are the real, unedited terminal output. `valgrind --leak-check=full --show-leak-kinds=all` was genuinely run against all six programs, per this course's standing policy since Chapter 9.

15.A.1 demo_all.cpp (full transcript)

```
$ 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 (all six programs)

```
$ 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: same result on every one -- 0 errors from 0
contexts,
 all heap blocks freed, on every one of the six programs)
```

Honesty note

This chapter's HashTable<K,V> (hashtable.h) and both hash functions (hashInt, hashString) are authored fresh for this course, since no lecture or exam skeleton ever provided one -- the plan file's own research confirmed a hashing lecture slot never existed anywhere in the raw material. The itinerary worked example's DATA (the route Badung -> Denpasar -> Gianyar -> Cianjur -> Bandung, and the exact shuffled input order) is reproduced faithfully from the real Final Exam's own printed task description and its recovered model-answer file, Hashing.cpp -- confirmed directly from the recovered archive, not reconstructed from memory. Two genuine, disclosed corrections were made to that file's approach (see Section 15.6's DEF callout): std::map replaced with std::unordered_map/this chapter's own HashTable (correcting the "Hashing via BST" misnomer), and the uninitialized `int a` replaced with a properly initialized bool flag (correcting a genuine undefined-behaviour bug). No other aspect of the original task or its expected output was altered. Beyond these two disclosed, deliberate corrections, this chapter's own validation pass found no other bugs. valgrind was run against every one of the six programs and every single run reported zero leaks and zero errors. The shipped code zip was independently re-extracted into a clean directory and rebuilt from scratch (`make clean && make run`) to confirm the actual deliverable, not just the working copy, compiles and runs identically.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Hash tables, hash functions, chaining, open addressing, universal hashing.)

[2] M.A. Weiss. “Data Structures and Algorithm Analysis in C++.” 4th ed., Pearson, 2014. (Hash table implementation, separate chaining, load factor, rehashing.)

[3] R. Sedgewick, K. Wayne. “Algorithms.” 4th ed., Addison-Wesley, 2011. (Hashing, collision resolution strategies, symbol table applications.)

[4] cppreference.com. “std::unordered_map”, “std::map”, “std::hash.” https://en.cppreference.com/w/cpp (accessed August 2026).

[5] EF234201 Data Structure, Final Exam (2023/2024 semester 2), Task 2 ("Hashing", 50 points) and its model-answer file Hashing.cpp — source of this chapter's itinerary worked example and the misnomer/bug this chapter corrects.

Profitina

profitina.com

COURSE TEXTBOOK · DATA STRUCTURE (EF234201)

Data Structure

Arrays, Lists, Stacks, Queues, Trees, Graphs, Heaps, and Hashing in C/C++

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 16 · EXERCISE CHAPTER · FINAL CHAPTER OF THE COURSE

Final Exam

No new material here — an individual, open, take-home coding exam covering Chapter 14 (Heaps & Priority Queues) and Chapter 15 (Hashing & Hash Tables), the two topics that close out this course. This chapter ships no code/ subfolder — the two submissions below (plus your written report) are yours to write and submit as your own work. This is also the last chapter of the entire 16-chapter Data Structure course.

Learning Objectives — after working through this chapter you will be able to:
(1) Complete Chapter 14's own MaxHeap skeleton by writing five specific member functions — extractMax, increaseKey, getMin, buildMaxHeap, isMaxHeap — each worth 10 of the exam's 50 heap points. (2) Solve the itinerary-reconstruction problem using a REAL hash table (Chapter 15's unordered_map, not a std::map masquerading as one) plus a reverse map, correctly identifying the trip's unique starting point and chaining from->to to print the whole route. (3) Write a short technical report — method, source code, running result and analysis, conclusion — documenting your Task 2 solution, understanding that this report carries more weight here than in any earlier assessment in this course. (4) Recognize that every subtask in this exam is a direct, disclosed application of something Chapter 14 or Chapter 15 already taught and demonstrated working — there is no new theory in this chapter, only integration and independent execution.

16.1 Recap: Chapters 14-15 in Brief

Chapter 14 built a complete array-based MaxHeap<int> from the ground up: heapifyUp/heapifyDown (the two private primitives every other operation rests on), insert, deleteRoot, getMax, heapifyArray (an O(n) bottom-up build), heapSort (tying back to Chapter 3's sorting survey), and printHeap. On top of that base, Chapter 14 also implemented — and genuinely compiled, ran, and valgrind-verified — five additional functions that this exam now asks YOU to write yourself: extractMax (remove-and-return with proper exception handling), increaseKey (raise a key, then restore heap order upward), getMin (find the minimum without removing it, exploiting the fact that a max-heap's minimum always sits at a leaf), buildMaxHeap (functionally identical to heapifyArray, built directly from an unsorted array), and isMaxHeap (a validity check on an arbitrary array, touching nothing).

Chapter 15 corrected a genuine misnomer in this course's own history: the real Final Exam's model-answer file was named Hashing.cpp, yet it solved its itinerary-reconstruction problem with std::map — a balanced binary search tree, not a hash table. Chapter 15 taught real hashing instead: hash functions, collision resolution (chaining and open addressing), load factor and rehashing, and std::unordered_map, applying all of it to the SAME itinerary-reconstruction worked example this exam reuses — a set of (from, to) legs reconstructed into one ordered route via a HashMap plus a reverse HashMap that finds the unique starting point.

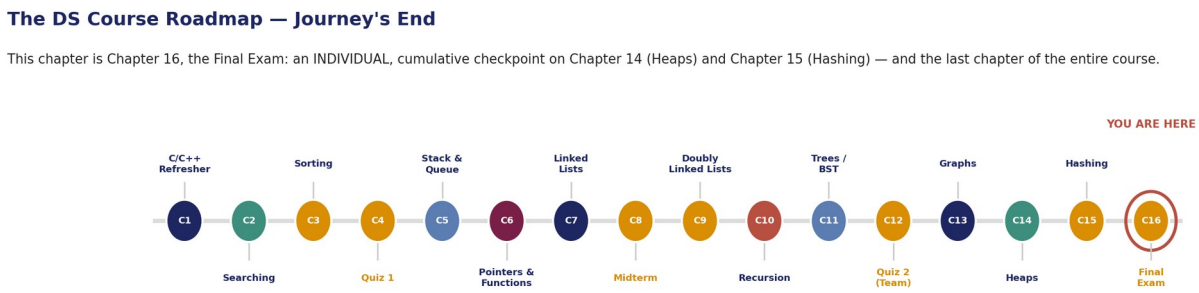


Figure 16.1 — This chapter sits at Chapter 16, the final slot in the sixteen-chapter DS roadmap: the course's last checkpoint, closing out Chapter 14 (Heaps) and Chapter 15 (Hashing).

Nothing below is new — this exam tests integration, not new theory

Every one of the six subtasks in Section 16.3 and 16.4 is a direct, named application of something Chapter 14 or Chapter 15 already taught, demonstrated working, and genuinely tested. Task 1's five functions are the exact five Chapter 14 wrote and verified — you are being asked to reproduce that same, already-proven design yourself, not invent a new one. Task 2 is the exact same worked example Chapter 15 solved twice (once with a hand-rolled hash table, once with std::unordered_map) — you are being asked to solve it a third time, on your own, and then explain your solution in writing.

16.2 How to Use This Exercise Chapter

What this chapter is — and is not

This is a practice chapter, not a new-content chapter — the last one in this course. The Final Exam, like the real DS assessment it is modeled on, is an INDIVIDUAL, open, take-home coding exam — you work alone, exactly like Quiz 1 (Chapter 4) and the Midterm (Chapter 8); team submissions of up to three students were a Quiz 2 (Chapter 12) exception only, not the course's general rule. Sections 16.3 and 16.4 below walk through the exam's two tasks, each with guidance callouts (THINK, TRAP, INSIGHT) rather than a full worked solution. Like every earlier exercise chapter, this chapter ships NO code/ subfolder — the programs you write for the Final Exam, and the report you write about them, are your own individual, independent work.

Before you start

Reread Chapter 14's heap.h and Chapter 15's itinerary demos (both `demo_itinerary_handrolled.cpp` and `demo_itinerary_unordered_map.cpp`) before opening a blank file. Every function you need to write below is a close cousin of something in one of those two files — your job is recognizing which cousin, not deriving an algorithm from first principles.

16.3 Task 1 — Complete the MaxHeap Skeleton (50 points, 5 × 10)

You are given the skeleton below — Chapter 14's own heap.h, with its base operations (insert, deleteRoot, getMax, heapifyArray, heapSort, printHeap, clear, and the private heapifyUp/heapifyDown) already complete, exactly as Chapter 14 demonstrated them working. Five member functions are left as TODO stubs. Complete each one so that the resulting class passes the same kind of heap-order and correctness checks Chapter 14's own automated harness ran.

Listing 16.1 — Supplied skeleton (heap.h)

Everything marked GIVEN below is complete and unchanged from Chapter 14 — do not need to modify it. Everything under the FINAL EXAM banner marked // TODO -- write your code here. is yours to write; the function SIGNATURES themselves (name, parameter types, return type) must not change, since your file will be compiled and tested against exactly these five signatures. The listing is split across four blocks below purely for page layout — it is one continuous heap.h file.

```
// heap.h -- SUPPLIED SKELETON for the Final Exam, Task 1.
// This is Chapter 14's own MaxHeap class. Every function you have already
// seen working (insert, deleteRoot, getMax, heapifyArray, heapSort,
// printHeap, clear, plus the private heapifyUp/heapifyDown) is GIVEN,
// unchanged. Your job is to write the bodies of the five functions below
// marked TODO -- nothing else in this file should need to change.
#ifndef HEAP_H
#define HEAP_H

#include <iostream>
#include <stdexcept>
#include <vector>

class MaxHeap {
public:
    MaxHeap() = default;

    // --- GIVEN: insert -- append + heapifyUp ("percolate up") -----------
    void insert(int key) {
        heap.push_back(key);
        size_t index = heap.size() - 1;
        heapifyUp(index);
    }

    // --- GIVEN: deleteRoot -- remove the max, print-and-return on empty --
    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);
    }
```

```
    // --- GIVEN: getMax -- peek at the root, O(1) -------------------------
    int getMax() const {
        if (heap.empty()) {
            std::cout << "Heap is empty" << std::endl;
            return -1;
        }
        return heap[0];
    }

    // --- GIVEN: heapifyArray -- O(n) bottom-up build, REPLACES 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));
        }
    }

    // --- GIVEN: heapSort -- repeatedly take the max, 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;
    }

    // --- GIVEN: clear -- empty the heap entirely -------------------------
    void clear() { heap.clear(); }
```

The five functions you must write

Everything from here to the matching FINAL EXAM banner below is what you complete — the base class above already gives you every helper these five functions need (heapifyUp, heapifyDown, and the private heap vector).

```cpp
// ============================== FINAL EXAM ==============================

    // 1. ExtractMax: remove AND return the maximum element. [10 points]
    int extractMax() {
        // TODO -- write your code here.
    }

    // 2. IncreaseKey: raise the key at `index` to `newValue`, then restore
    //    the heap-order property. [10 points]
    void increaseKey(int index, int newValue) {
        // TODO -- write your code here.
    }

    // 3. GetMin: find the minimum element WITHOUT removing it. [10 points]
    int getMin() {
        // TODO -- write your code here.
    }

    // 4. BuildMaxHeap: build a Max-Heap from an unsorted array. [10 points]
    void buildMaxHeap(const std::vector<int> &arr) {
        // TODO -- write your code here.
    }

    // 5. IsMaxHeap: check whether an arbitrary array already satisfies the
    //    Max-Heap property, without touching or rebuilding it. [10 points]
    bool isMaxHeap(const std::vector<int> &arr) {
        // TODO -- write your code here.
    }

// ============================== FINAL EXAM ==============================
```

```cpp
private:
    std::vector<int> heap;

    void heapifyDown(size_t index) { /* GIVEN -- unchanged from Ch.14 */ }
    void heapifyUp(size_t index)   { /* GIVEN -- unchanged from Ch.14 */ }
};

#endif // HEAP_H
```

Where Each Final Exam Subtask Comes From

Task 1 hands you Chapter 14's own MaxHeap skeleton with five functions removed; Task 2 reuses Chapter 15's own itinerary worked example -- every row below traces back to a specific earlier chapter.

#	Final Exam Subtask	Traces back to	Pts
1	extractMax() -- remove and return the maximum	Ch.14 -- deleteRoot()'s exact removal shape, upgraded to throw on empty rather than print-and-return	10
2	increaseKey(index, newValue)	Ch.14 -- heapifyUp(): raise a key, then restore heap order upward from that index	10
3	getMin() -- find the minimum without removing it	Ch.14 -- Section 14.6's proof that a max-heap's minimum always sits at a leaf	10
4	buildMaxHeap(arr) -- build from an unsorted array	Ch.14 -- Section 14.4's O(n) bottom-up build, identical in shape to heapifyArray()	10
5	isMaxHeap(arr) -- validity check on an arbitrary array	Ch.14 -- the heap-order property, checked without touching or rebuilding anything	10
6	Hashing: reconstruct the itinerary + written report	Ch.15 -- HashMap<from,to> + reverse map to find the start, then chain from->to; unordered_map, not std::map	50

Total: 100 points

Figure 16.2 — Every subtask in this exam traces back to a specific Chapter 14 or Chapter 15 idea; none of them is new material.

16.3.1 Subtask 1 — extractMax() (10 pts)

Remove AND return the maximum element (the root) from the heap, restoring the heap-order property afterward. Unlike deleteRoot() (which the skeleton already gives you, printing a message and returning on an empty heap), extractMax() must communicate an empty-heap condition through a proper C++ exception rather than a printed message and a silent return.

Hint

The removal SHAPE is identical to deleteRoot(): move the last element into the root's slot, shrink the vector by one, then heapifyDown(0) to restore order — the only two differences are (a) you must capture and return the ORIGINAL root value before overwriting it, and (b) an empty heap must throw rather than print-and-return.

Save the value before you overwrite index 0

heap[0] = heap.back() overwrites the very value you need to return — read and store heap[0] into a local variable FIRST, then perform the removal, then return the saved value. Also remember to only call heapifyDown when the heap is non-empty AFTER the pop_back(): a heap that had exactly one element before extraction is empty afterward, and heapifyDown(0) on an empty vector reads out of bounds.

Why a thrown exception here, and not a printed message

deleteRoot()'s print-and-return behavior was the ORIGINAL skeleton's own design choice, predating this exam. extractMax() is the exam's deliberate upgrade: a caller who writes `int m = h.extractMax();` on an empty heap and gets back a printed message plus an unspecified int has no reliable way to detect the failure in code — a caller who wraps the same call in a try/catch for std::runtime_error does. This is a genuine, disclosed API-design improvement, not busywork.

16.3.2 Subtask 2 — increaseKey(int index, int newValue) (10 pts)

Raise the value stored at a given index to newValue, then restore the heap-order property by moving the (now larger) value upward as far as it needs to go.

Hint

This is heapifyUp(), applied starting from the index you were given rather than from the newly-inserted last index — the function you write is a thin wrapper: validate, assign, then call the SAME private heapifyUp(index) the GIVEN insert() already calls.

Two DIFFERENT failure modes need two DIFFERENT exceptions

An index that is negative or at/past heap.size() is an out-of-range access — a different kind of caller mistake than a newValue that is actually SMALLER than the current value at that index, which the function's own name ("increase") promises never happens. Distinguish these with two distinct, clearly-named exception types rather than one generic error, and check the index bound BEFORE you index into the vector with it — reading heap[index] to compare against newValue when index is already out of range is itself undefined behavior.

Only ever needs to move UPWARD, never down

Because the new value can only be LARGER than what was there before, it can only violate the heap property with its PARENT (parent too small), never with its children (children were already no larger than the old, smaller value, so they remain no larger than the new, larger one). This is exactly why heapifyUp — not heapifyDown — is the correct restoration step here.

16.3.3 Subtask 3 — getMin() (10 pts)

Retrieve the minimum element currently stored in the max-heap, WITHOUT removing it and without disturbing the heap's structure in any way.

Hint — where must the minimum live?

A max-heap only guarantees a parent is no smaller than its children — it says nothing about ordering AMONG siblings or across different subtrees, so the minimum could in principle be anywhere. But it can never be an INTERNAL node (any node with at least one child must be >= that child, so it cannot be the smallest value present) — the minimum must therefore be at a LEAF. In this array representation, every index from heap.size()/2 through heap.size()-1 is a leaf (no children exist for those indices) — scanning only that range, roughly half the array, is enough.

This scans the LEAVES, not the whole array — and it is still O(n), not O(log n)

Do not mistake this for an O(log n) operation the way getMax() is: even though you only scan roughly half the array, that is still linear in the heap's size, since roughly half of any array is still Theta(n). A common mistake is scanning ALL n elements instead of just the leaf range — this still produces a correct answer, just doing exactly twice the necessary work; a more subtle mistake is getting the leaf-range starting index off by one (it is heap.size()/2, not heap.size()/2 - 1 or heap.size()/2 + 1) and silently excluding or including one extra internal node.

An empty heap has no minimum to report

Just like extractMax(), getMin() should throw rather than return a fabricated sentinel value on an empty heap — an int return type has no way to represent "there is no such value" other than a value that could also, confusingly, be a legitimate heap entry.

16.3.4 Subtask 4 — buildMaxHeap(const std::vector<int>& arr) (10 pts)

Build a max-heap from an arbitrary, unsorted input array, replacing whatever this heap object currently holds.

Hint — this is heapifyArray(), by another name

Compare this function's required behavior against the GIVEN heapifyArray() in the skeleton above: both take an arbitrary array, both replace the object's own internal heap with it, and both restore the heap property via the standard O(n) bottom-up build — starting from the last non-leaf index (heap.size()/2 - 1) and calling heapifyDown at every index down to 0. If your buildMaxHeap and the given heapifyArray end up looking nearly identical, that is expected, not a sign you misread the task.

Signed/unsigned subtraction on an empty array

heap.size() is an UNSIGNED size_t; computing heap.size()/2 - 1 when heap.size() is 0 or 1 either produces 0 (safe, loop simply does not execute if you use a signed loop counter starting from n/2-1 cast appropriately) or, if done carelessly with an unsigned loop variable, underflows to a huge number and the loop runs far past the array's bounds. Chapter 14's own heap.h handles this by computing the loop bound as a signed int and only casting to size_t inside the loop body, after confirming the index is >= 0 — follow the same pattern.

Why the exam asks for this AND heapifyArray already exists

buildMaxHeap and heapifyArray are functionally the same algorithm, deliberately — Chapter 14 pointed this out directly (Section 14.4's side-by-side note) so this subtask should feel like recognition, not invention. The exam names it buildMaxHeap specifically because that is the vocabulary heap literature and other courses use most often for exactly this O(n) construction step, distinct from calling insert() n times (which would cost O(n log n) instead).

16.3.5 Subtask 5 — isMaxHeap(const std::vector<int>& arr) (10 pts)

Check whether an ARBITRARY array (not this object's own internal heap) already satisfies the max-heap property — every parent at index i must be >= both of its children at 2i+1 and 2i+2 — without modifying or rebuilding anything.

Hint — check every INTERNAL node, and only internal nodes

A leaf has no children to violate the property against, so only indices with at least one child need checking: i ranges from 0 up to (and including) (arr.size()-2)/2. For each such i, verify arr[i] >= arr[2*i+1], and, if a right child exists (2*i+2 < arr.size()), also arr[i] >= arr[2*i+2]. The moment any single check fails, the array is not a max-heap — return false immediately rather than continuing to scan.

arr.size() - 2 on an array of 0 or 1 elements underflows

arr.size() is unsigned; for an array with fewer than 2 elements, arr.size() - 2 wraps around to a huge value BEFORE the loop bound is even used, and the very next array access reads far out of bounds — this is a genuine bug Chapter 14's own QA process found in the recovered exam model answer, disclosed rather than silently reproduced (Appendix 14.A). Guard explicitly: an array of 0 or 1 elements trivially satisfies the max-heap property (there is no parent-child pair to violate), so return true immediately before evaluating any size-dependent index arithmetic.

This checks an array Chapter 14's buildMaxHeap could have produced, but need not have
isMaxHeap takes an arbitrary vector<int> as its argument — it does not need to be, and generally is not, this object's OWN heap. This is precisely why it is written as a check on the given arr parameter throughout, never referring to the class's private heap member at all; a correct implementation could be written as a free function, or (as Chapter 14 chose, disclosed as a course design decision) a static member function, without changing its behavior.

16.4 Task 2 — Hashing: Itinerary Reconstruction + Written Report (50 points)

You are given an unordered list of flight/travel legs, each recorded as a (from, to) pair. Using a hash-based approach (a real hash table — std::unordered_map, or a hand-rolled hash table like Chapter 15's own — not std::map), reconstruct and print the full itinerary in its correct travel order, starting from the trip's unique starting point.

The exact algorithm, in three steps
[1] Build a HashMap called dataset, where every entry has the form from -> to (one entry per leg). [2A] Build a SECOND HashMap, reverseMap, where every entry has the form to -> from — the reverse of every entry in dataset. [2B] Find the itinerary's starting point: traverse dataset's keys, and the one key that does NOT appear as a key of reverseMap is the trip's start (it is never anyone's destination, so it never appears as a "to"). [3] Starting from that point, repeatedly look up dataset[current] to get the next city, printing each from -> to hop, until there is no further entry to follow.

Worked example — the exact dataset this exam uses (also Chapter 15's own worked example, for continuity), given in this shuffled insertion order:

```
Cianjur  -> Bandung
Badung   -> Denpasar
Gianyar  -> Cianjur
Denpasar -> Gianyar
```

reverseMap built from the dataset above:

```
Bandung  -> Cianjur
Denpasar -> Badung
Cianjur  -> Gianyar
Gianyar  -> Denpasar
```

Badung never appears as a KEY of reverseMap (nobody's route ever arrives AT Badung) — Badung is the starting point. Chaining from there through dataset produces the expected output:

```
Badung -> Denpasar, Denpasar -> Gianyar, Gianyar -> Cianjur, Cianjur -> Bandung
```

Hint — reuse Chapter 15's exact shape, on your own

This is precisely the algorithm Chapter 15's Section 15.6 walked through twice — once with a hand-rolled chaining hash table keyed on hashString(), once with std::unordered_map<std::string,std::string>. You are not being asked to invent a new algorithm; you are being asked to implement this same three-step shape yourself, correctly, and explain it in your own words in your report.

Use a REAL hash table — this is the exact misnomer Chapter 15 corrected

The real exam's own historical model answer for this task used std::map<std::string,std::string> — a balanced binary search tree with O(log n) operations, not a hash table, despite the task being named "Hashing" and explicitly requiring "the hashing method." A submission that reaches for std::map here repeats that same disclosed misnomer; use std::unordered_map (average O(1) lookups via a real hash function) or a hand-rolled hash table like Chapter 15's own, and be ready to explain in your report WHY your container choice is genuinely hashing and not merely a same-named substitute.

Initialize every loop/flag variable you use to separate hops with a comma

A historical version of this exact program's model answer declared a bare `int a;` and used it, uninitialized, as a flag deciding whether to print a leading comma before each hop — reading an uninitialized local variable's value is undefined behavior in C++, and the resulting output was unreliable. Declare and INITIALIZE any such flag explicitly (a `bool firstHop = true;`, for instance) before your traversal loop begins, exactly as Chapter 15's own corrected demo does.

Why the reverse map is the clever part, not the traversal

Once the starting point is known, printing the route is a simple, repeated hash-map lookup — the genuinely clever step is FINDING that starting point in the first place without scanning the whole dataset by brute force for "which from-city is never a to-city": building reverseMap and checking membership turns an otherwise O(n^2) search (compare every from against every to) into two O(n)-ish passes over the data, each lookup running in average O(1) time thanks to real hashing.

16.4.1 The Written Report (required, part of these 50 points)

Report contents — required, and prominent in this exam's own instructions

Your report must cover: (1) the METHOD(S) you used to solve the problem — the hashing approach, the reverse-map trick, and why they are correct; (2) your SOURCE CODE, included or clearly referenced; (3) your RUNNING RESULT and an ANALYSIS of it (does the output match the expected itinerary? what would happen on a malformed input with no unique start, or with a cycle?); and (4) a CONCLUSION summarizing what the exercise demonstrated. This is the same four-part shape — method, code, result-and-analysis, conclusion — every technical report in a professional or research setting is expected to have.

Note: unlike prior assessments' report components, the Final Exam's report carries proportionally more weight

Quiz 1, the Midterm, and Quiz 2 asked for working, correct code as their entire deliverable, with no separately-graded written report component. The Final Exam is different by explicit design: it is your CAPSTONE documentation of the whole course's data-structure toolkit, and its written report is graded as a substantial, separately-weighted part of Task 2's 50 points (Section 16.5's rubric gives the exact split) — not a token afterthought. Treat the analysis and conclusion sections as seriously as the code itself: a correct program with a thin, one-paragraph report will not earn full marks on this task the way a working Quiz 1 or Midterm program alone would have.

What a strong analysis section actually contains

Beyond confirming the output matches Section 16.4's worked example, a strong analysis discusses: why std::unordered_map (or your hand-rolled table) is the RIGHT container here and what would go wrong (or merely become slower) with std::map instead; what your program does on an input with NO unique starting point (every city is someone's destination — a cycle with no start) or on an input with more than one plausible start; and the time complexity of your solution as a function of the number of legs, in Big-O terms, tying back to Chapter 2's formal introduction of that notation.

16.5 Final Exam Format and Grading

The Final Exam is an INDIVIDUAL, open, take-home coding exam — not a team assignment, exactly like Quiz 1 (Chapter 4) and the Midterm (Chapter 8). "Open" means the textbook, your own notes, and standard C++ reference material (such as cppreference.com) are all fair game while you work; it does NOT mean you may copy another student's code, another student's report, or submit work that is not genuinely your own — an open exam still tests whether YOU can apply what the course has taught, working independently, and most real versions of this exam require a signed, individual pledge of academic honesty submitted alongside your work.

DS's own point-per-subtask rubric, one more time

Like every earlier assessment in this course, the Final Exam is graded task by task, in whole points, not by a percentage-weighted Design/Implementation/Analysis/Conclusion template. Task 1's five subtasks are worth 10 points each (50 total); Task 2's 50 points split into 35 for a correct, genuinely-hashing implementation and 15 for the written report — a deliberately heavier report weight than any earlier assessment in this course carried, reflecting Section 16.4.1's note above.

Subtask	What it tests	Points
1a. extractMax()	Correct remove-and-return of the max, with a thrown exception on empty (not a printed message)	10
1b. increaseKey(index, newValue)	Correct bounds/direction validation, then heapifyUp restoring order	10

1c. getMin()	Correct leaf-range scan (not a full O(n) scan of the wrong kind, and not internal nodes), throws on empty	10
1d. buildMaxHeap(arr)	Correct O(n) bottom-up build from an arbitrary unsorted array	10
1e. isMaxHeap(arr)	Correct validity check on an arbitrary array, with the 0/1-element underflow guarded	10
2a. Hashing implementation	Correct HashMap + reverseMap algorithm using REAL hashing (unordered_map or a hand-rolled hash table, not std::map); correctly finds the unique start and prints the full ordered route	35
2b. Written report	Method, source code, running result & analysis, and conclusion — all four parts present and substantive	15
Total		100

Submit Task 1 as a single, individually compilable heap.h/.cpp (or a single .cpp with the class inline) demonstrating all five new functions working correctly against the worked example in Section 16.3, and Task 2 as a single .cpp file plus your written report — a program that does not compile earns no points for that subtask, regardless of how close its logic is to correct, so leave time to actually build and run both submissions before finalizing your report.

16.6 Chapter Summary — and the End of This Course

- The Final Exam introduces no new material — it is a cumulative checkpoint on Chapter 14 (Heaps & Priority Queues) and Chapter 15 (Hashing & Hash Tables), the two chapters immediately before it.
- It is an INDIVIDUAL, open, take-home coding exam — not a team assignment; team-of-up-to-3 submissions were a Quiz 2 exception only.
- Task 1 (50 pts, 5 x 10): complete Chapter 14's own MaxHeap skeleton by writing extractMax, increaseKey, getMin, buildMaxHeap, and isMaxHeap — the exact five functions Chapter 14 already implemented and verified.
- Task 2 (50 pts: 35 implementation + 15 report): reconstruct Chapter 15's itinerary worked example using a REAL hash table, plus a written report covering method, code, result-and-analysis, and conclusion — this report is weighted more heavily than any earlier assessment's report component in this course.
- Grading follows DS's own native point-per-subtask rubric — the same shape used in Chapters 4, 8, and 12, applied one final time.
- This chapter ships no code/ subfolder: both submissions, and the report, are each student's own individual, independent work.

This is the sixteenth and final chapter of Data Structure (EF234201). From arrays and searching in Chapter 1 through heaps and hashing in Chapters 14-15, every data structure and algorithm this book covered has led here — this exam asks you to show, one more time and entirely on your own, that the toolkit is genuinely yours. Congratulations on reaching the end of the course.

Appendix 16.A — Self-Check Checklist (Non-Graded)

Before submitting your Final Exam, run it through this checklist. It costs a few minutes and catches most of the mistakes graders see most often on a first submission of this exam.

- Do both programs compile cleanly with `g++ -Wall -Wextra -std=c++17`, with zero warnings?
- Task 1: does extractMax() save heap[0] into a local variable BEFORE overwriting it, and throw (not print) on an empty heap?
- Task 1: does increaseKey() check the index bound BEFORE indexing into the vector with it, and throw two DIFFERENT exception types for an out-of-range index versus a decreasing value?
- Task 1: does getMin() scan only the leaf range (heap.size()/2 through heap.size()-1), not the whole array and not just the internal nodes?
- Task 1: does isMaxHeap() return true immediately for an array of 0 or 1 elements, BEFORE evaluating arr.size()-2 anywhere?
- Task 2: have you used std::unordered_map (or a hand-rolled hash table) rather than std::map for both dataset and reverseMap?
- Task 2: is every flag variable used in your traversal loop explicitly initialized before the loop begins — no uninitialized int used as a comma-separator flag?
- Task 2: does your program correctly identify Badung as the starting point on the worked example, and print the exact expected route?
- Report: does it contain all four required parts — method, source code, running result & analysis, and conclusion — each substantive, not a single line?
- Is every file you are submitting, and every word of your report, genuinely your own individual work, written for this Final Exam — not copied from another student, a prior year's solution, or an AI tool your course's own academic-integrity policy does not permit?

References

[1] This exercise chapter's assessment format is modeled directly on this course's real Final Exam (EF234201 Data Structure): an individual, open, take-home coding exam of two tasks — Heap (50 points, 5 subtasks x 10) and Hashing (50 points, including a required written report) — graded task by task rather than by a percentage-weighted rubric. This format is reused here as-is; the task framing above (which recaps Chapters 14-15's actual content and replaces a known code-quality issue in the original exam's own model answer — the std::map-as-"Hashing" misnomer and an uninitialized-variable bug — with corrected, generic guidance rather than silently reproducing either) has been adapted for this textbook.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms.” 3rd ed., MIT Press, 2009. (Heaps and heapsort, Chapter 6; hash tables, Chapter 11.)

[3] cppreference.com. “std::unordered_map,” “std::vector,” “Exceptions.” https://en.cppreference.com/w/cpp (accessed August 2026).