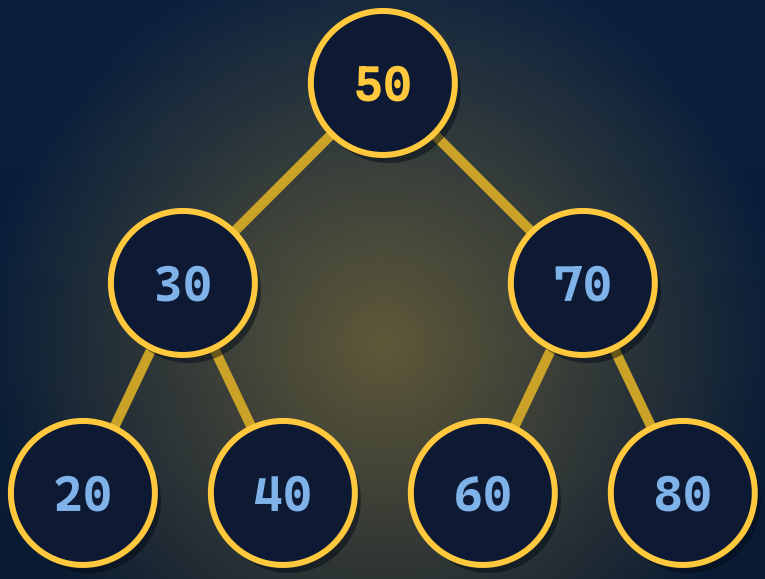
369 TESLA | KALVIYOGI

UNIT IV

DATA STRUCTURES & ALGORITHMS

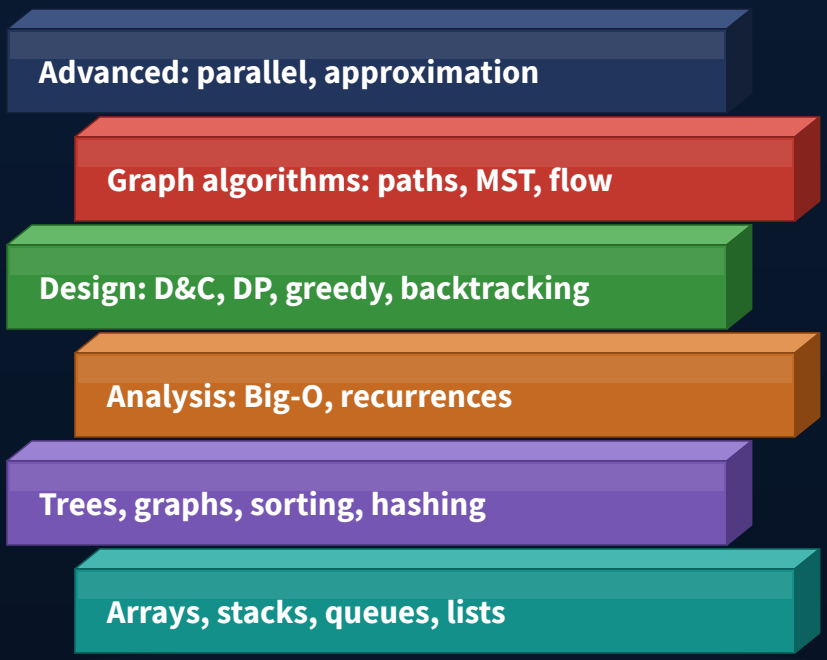
a brain-friendly guide

Head First style · puzzles & Q&A
Every trace computed
Step-by-step figures
TRB exam points



binary search tree

```
push(x): top ← top + 1
T(n) = 2T(n/2) + n → Θ(n log n)
h(k) = k mod m
dist[v] ← min(dist[v], dist[u] + w)
front = (front + 1) mod n
LCS[i][j] = LCS[i-1][j-1] + 1
inorder: L · N · R
balance = h(L) − h(R) ∈ {-1, 0, 1}
O(1) < O(log n) < O(n) < O(n log n)
Kruskal: sort edges, union-find
f(n) = O(g(n)) ⇔ f ≤ c·g
8-Queens: 92 solutions
```



VSKM Nagarajan

EDUNIVRA PVT LTD

369 TESLA | KALVIYOGI

UNIT IV

DATA STRUCTURES AND ALGORITHMS

A Brain-Friendly Visual Guide: Arrays · Stacks · Queues · Linked Lists · Trees · Graphs · Sorting · Hashing · Analysis · Design Techniques · Graph Algorithms · Advanced Algorithms

VSKM Nagarajan

EDUNIVRA PVT LTD

Data Structures and Algorithms — Unit IV
VSKM Nagarajan · 369 TESLA | KALVIYOGI

Published by EDUNIVRA PVT LTD.

This edition is a rewritten, redesigned version of the author's Unit IV notes. It follows the six syllabus areas of the source (fundamentals of data structures; trees, graphs, sorting, searching and hashing; performance analysis and recurrences; algorithm design techniques; graph algorithms; advanced algorithms) page by page, so nothing from the notes is left out.

This is a book of **notes**: algorithms are explained in numbered steps, figures and trace tables, not as program listings. Every trace, count and worked answer was computed and checked before it was printed. Technical statements were checked against the standard texts (Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*; Horowitz, Sahni and Rajasekaran, *Fundamentals of Computer Algorithms*; Tremblay and Sorenson, *An Introduction to Data Structures with Applications*). Where the source notes printed a wrong value or a wrong rule, the correct version is taught and a **Watch it!** box tells you what the error was, so that you are not trapped by it in the examination.

Contents

How to Use This Book 6

Fundamentals of Data Structures 9

Chapter 1 · What Data Structures Are 10

1.1 Data, Information and Structure 10
1.2 Why We Need Data Structures 11
1.3 Primitive and Non-Primitive Data Structures 11
1.4 Linear and Non-Linear Data Structures 12
1.5 Static and Dynamic Data Structures 12
1.6 Abstract Data Types 13
1.7 The List, Stack and Queue ADTs 14

Chapter 2 · Arrays and Sparse Matrices 16

2.1 What an Array Is 16
2.2 Declaring and Initializing Arrays 16
2.3 One-, Two- and Multi-Dimensional Arrays 17
2.4 Memory Representation and Address Calculation 17
2.5 Traversal, Insertion and Deletion 19
2.6 Searching in Arrays 19
2.7 Applications of Arrays 20
2.8 Matrices and Sparse Matrices 20
2.9 Sparse Matrix Representation: Triplet Form 21

Chapter 3 · Stacks 23

3.1 What a Stack Is 23
3.2 The LIFO Principle 24
3.3 Stack Operations 24
3.4 Overflow and Underflow 25
3.5 Array Implementation of a Stack 25
3.6 Linked List Implementation of a Stack 26
3.7 Applications of Stacks 27
3.8 Infix, Prefix and Postfix Notation 28
3.9 Converting Infix to Postfix 29
3.10 Evaluating a Postfix Expression 29
3.11 Checking Balanced Brackets 30

Chapter 4 · Queues and Priority Queues 32

4.1 What a Queue Is 32
4.2 Queue Operations 33
4.3 Front and Rear in an Array: the Simple Queue 33
4.4 Circular Queue 35
4.5 Linked Queue 36
4.6 Double-Ended Queue (Deque) 36
4.7 Priority Queue: Definition and Need 37
4.8 Ascending and Descending Priority Queues 38
4.9 Priority Queue Operations and Implementations 38
4.10 Applications of Queues and Priority Queues 40

Chapter 5 · Linked Lists 42

5.1 What a Linked List Is 42
5.2 Why Linked Lists Are Needed 43

5.3 Arrays and Linked Lists Compared 44
5.4 The Node and the Head Pointer 44
5.5 Singly Linked List 44
5.6 Doubly Linked List 45
5.7 Circular Linked List 45
5.8 Traversal 46
5.9 Insertion 46
5.10 Deletion 47
5.11 Searching 48
5.12 Applications of Linked Lists 48

Chapter 6 · Choosing and Applying Data Structures 51
6.1 Array and Linked List 51
6.2 Stack and Queue 52
6.3 Linear and Non-Linear Structures 52
6.4 Static and Dynamic Structures 53
6.5 The Major Structures Side by Side 53
6.6 Data Structures in Operating Systems 54
6.7 Data Structures in Database Systems 54
6.8 Data Structures in Compiler Design 55
6.9 Data Structures in Computer Networks 56
6.10 Data Structures in Artificial Intelligence 57

Trees 60

Chapter 7 · Trees and Graphs: Tree Terminology 61
7.1 Non-Linear Structures: Trees and Graphs 61
7.2 Why Trees and Graphs Matter 62
7.3 Definition of a Tree 62
7.4 Root, Parent, Child and the Other Relationships 63
7.5 Leaf Node, Internal Node and Degree 64
7.6 Level, Depth and Height 65
7.7 Ordered and Unordered Trees 65

Chapter 8 · Forests and Binary Trees 68
8.1 Forests 68
8.2 Converting a Forest into a Binary Tree 69
8.3 Applications of Forests 70
8.4 Binary Trees 70
8.5 Properties of Binary Trees 70
8.6 Full and Complete Binary Trees 71
8.7 Perfect, Balanced and Skewed Binary Trees 72
8.8 Array Representation 72
8.9 Linked Representation 73

BEFORE YOU BEGIN

How to Use This Book

"I can't believe they put that in an algorithms book!"

Who is this book for?

This book is for you if…

- you are preparing for TRB Computer Instructor, PG-TRB, TNPSC or a university paper on Data Structures and Algorithms;
- you can write a loop but have never asked why one sort takes a second and another a day;
- you would rather **understand** one example deeply than memorise ten definitions.

This book is probably not for you if…

- you want a dictionary of terms with no pictures;
- you are scared that a book which talks to you, asks you questions and makes you write in the margins cannot be serious. (It is serious. Every answer was checked.)

We know what you're thinking

"How can *this* be a serious algorithms book?" "What's with all the pictures?" "Can I really learn AVL rotations and the Master Theorem this way?" And we know what your **brain** is thinking. Your brain craves novelty. It is always scanning for something unusual. It quietly throws away the boring, repeated, textbook-shaped things, because they do not look important. So the twentieth definition that starts with "It is a method of…" slides straight out of your head.

The way to convince your brain that stacks, trees and recurrences are worth keeping is to make them **look important**: pictures with words **inside** them, a conversational voice, questions that make you stop, puzzles, a little surprise. That is what every page of this book tries to do.

Here's what we did

1 Pictures first
Words sit inside or beside the figure they describe, not on another page. Your brain links them better.

2 One story per chapter
Each chapter opens with a real situation (a pile of plates, a bus queue, a road map) and the theory explains it.

3 Every trace checked
Every stack, queue, tree and sorting trace, every shortest path and every count was run and checked before printing.

4 Mistakes shown
Every error of the source notes is turned into a **Watch it!** box: what was printed, what is right, why.

5 You do the work
Pencil exercises and matching puzzles make you **use** the idea right away. Solutions are at the chapter end.

6 Exam angle
Purple **Exam point** boxes say exactly what the TRB examiner asks and which trap options appear.

The elements you will meet



BRAIN POWER

A question for you to **think** about. There is no printed answer: the point is that your brain does the work. Spend one minute before you read on.

There are no Dumb Questions

Q: **Is it OK to ask very basic questions?**

A: Yes. These boxes collect the questions students are afraid to ask in class. Most of them turn up as MCQ traps.

✎ Sharpen your pencil

An exercise. Write your answers in the book. The worked solution is at the end of the chapter, so do not peek first.



WATCH IT! · COMMON MISTAKE

A wrong statement that students (and sometimes printed notes) make, with the correction and the reason.

Fireside Chats · two concepts, one fireplace

Tonight's guest: Stack	Tonight's guest: Queue
Two related ideas talk to each other about their differences. You will never mix them up again.	And the conversation is far easier to remember than a comparison table. (We give the table too.)

EXAM POINT

What the examiner asks, the one-line answer, and the trap option to avoid.

Bend your brain into submission

1. **Slow down.** The more you understand, the less you have to memorise. Stop at every Brain Power and really think.
2. **Do the exercises. Write your own notes.** Using a pencil while you learn is proven to help memory.
3. **Read the Dumb Questions.** They are not optional side-bars; they are part of the core content.
4. **Do not read everything in one place.** Stand up, stretch, move to another seat. Your brain links learning to the surroundings.
5. **Talk about it, out loud.** Explain a rotation or a recursion tree to a friend or to the wall. If you can explain it, you know it.
6. **Drink water and sleep.** A dehydrated brain thinks slowly, and a lot of the real learning happens after you close the book.
7. **Draw it.** Every tree, graph and trace table here is meant to be re-drawn by you, from memory, on a blank page.

PART I · UNIT IV SYLLABUS 1

Fundamentals of Data Structures

Chapter 1 What Data Structures Are

Chapter 2 Arrays and Sparse Matrices

Chapter 3 Stacks

Chapter 4 Queues and Priority Queues

Chapter 5 Linked Lists

Chapter 6 Choosing and Applying Data Structures

Data, structure and data structure · need and advantages · primitive and non-primitive, linear and non-linear, static and dynamic structures · abstract data types: List, Stack, Queue · arrays: declaration, one-, two- and multi-dimensional, address calculation, traversal, insertion, deletion, searching · sparse matrices and triplet form · stacks: LIFO, push, pop, peek, overflow, underflow, array and linked implementations, applications · queues: FIFO, simple, circular, deque, priority queue · linked lists: singly, doubly, circular; traversal, insertion, deletion, searching · comparisons and applications.

The same million records can take a second or a week to search. The difference is not the computer; it is how the data is arranged.

CHAPTER · UNIT IV · FUNDAMENTALS OF DATA STRUCTURES

1 What Data Structures Are

A canteen keeps plates in a pile, a ticket counter keeps people in a line, a school keeps files in folders. Each one chose a shape for its data. So does every program.

OPENING SITUATION

Your college has one lakh student records. The office wants three things: find a student by roll number instantly, add new admissions every June, and print everyone in order of marks. Kept as a heap of papers, each request means going through the whole heap. Kept as a sorted list, the search is quick but every admission means shifting thousands of entries. Kept in the right **data structure**, all three are fast. This chapter sets out the vocabulary for the whole unit: what data and structure mean, the ways data structures are classified, and the idea that holds the unit together, the **abstract data type**, which separates *what* a structure does from *how* it is built.

IN THIS CHAPTER

- 1.1 Data, Information and Structure
- 1.2 Why We Need Data Structures
- 1.3 Primitive and Non-Primitive Data Structures
- 1.4 Linear and Non-Linear Data Structures
- 1.5 Static and Dynamic Data Structures
- 1.6 Abstract Data Types
- 1.7 The List, Stack and Queue ADTs

YOU NEED

- Variables and data types in any programming language.
- The idea of memory as numbered cells.

1.1 Data, Information and Structure

Data is a collection of raw facts, values or symbols: numbers, text, images, sound (source p1). On its own, data is unprocessed and unorganized. **Information** is what you get after processing it: data made meaningful.

Table 1.1 Data and information (source p1)

Data (raw facts)	Processing	Information (meaningful result)
95, 88, 76 (marks)	average	class average 86.3
32 °C, 34 °C, 35 °C (temperatures)	compare	the week is getting hotter
"Ravi", "20-06-2026"	link name to date	Ravi's date of joining

A **structure** is an organized arrangement of parts that work together, like the frame of a house, a school timetable or a bookshelf (source p2). In computing, a structure shows how data items are **arranged** and how they are **related** to one another.

Put the two together: a **data structure** is a way of organizing, storing and managing data in memory so that it can be used efficiently (source p3). It has three parts:

data structure = data + organization (arrangement and relationships) + operations allowed on it

★ **Data is raw facts; information is processed data; a data structure is data plus its organization plus its operations.**

1.2 Why We Need Data Structures

As data grows, the way it is arranged decides how fast a program runs and how much memory it wastes (source p4). Without a suitable structure, searches and updates become slow, memory is wasted and large data becomes hard to manage.

Worked 1.1 Finding one roll number among 10 lakh records

How the records are kept	Method	Comparisons in the worst case
unordered list	linear search: check one by one	1 000 000 (about 500 000 on average)
sorted array	binary search: halve the range each time	**20** ($\lfloor \log_2 1\,000\,000 \rfloor + 1$)
hash table	compute the slot from the key	about 1 on average

Same data, same computer: the structure alone changes the work from a million steps to twenty. Later chapters explain each method in full.

Table 1.2 Data structures you already know from daily life (source p5)

Structure	Real-life example	The behaviour it shows
array	attendance register	every student has a position; go straight to number 17
stack	plates in a canteen	the last plate placed is the first taken (LIFO)
queue	ticket counter line	the first person in line is served first (FIFO)
linked list	train coaches	each coach is coupled to the next; coaches can be added anywhere
tree	family tree, folders on a disk	one item branches into sub-items: a hierarchy
graph	road map, social network	many items connected in many ways: a network

Speed

Fewer steps for search, insert and delete; lower time complexity (source p6).

Memory

Space used where it is needed, not wasted on empty slots.

Clarity and scale

Cleaner, reusable code that keeps working as data and users grow.

★ **The right structure turns slow, wasteful programs into fast, scalable ones.**

1.3 Primitive and Non-Primitive Data Structures

The first split is by what one item can hold.

- **Primitive** data structures are the basic types built into the language, each holding a **single value** of fixed format and size: integer, float, character, boolean, and in languages such as C the pointer (source p7).
- **Non-primitive** data structures are built from primitive ones and hold **many values** and the relationships between them: arrays, linked lists, stacks, queues, trees, graphs (source p8).

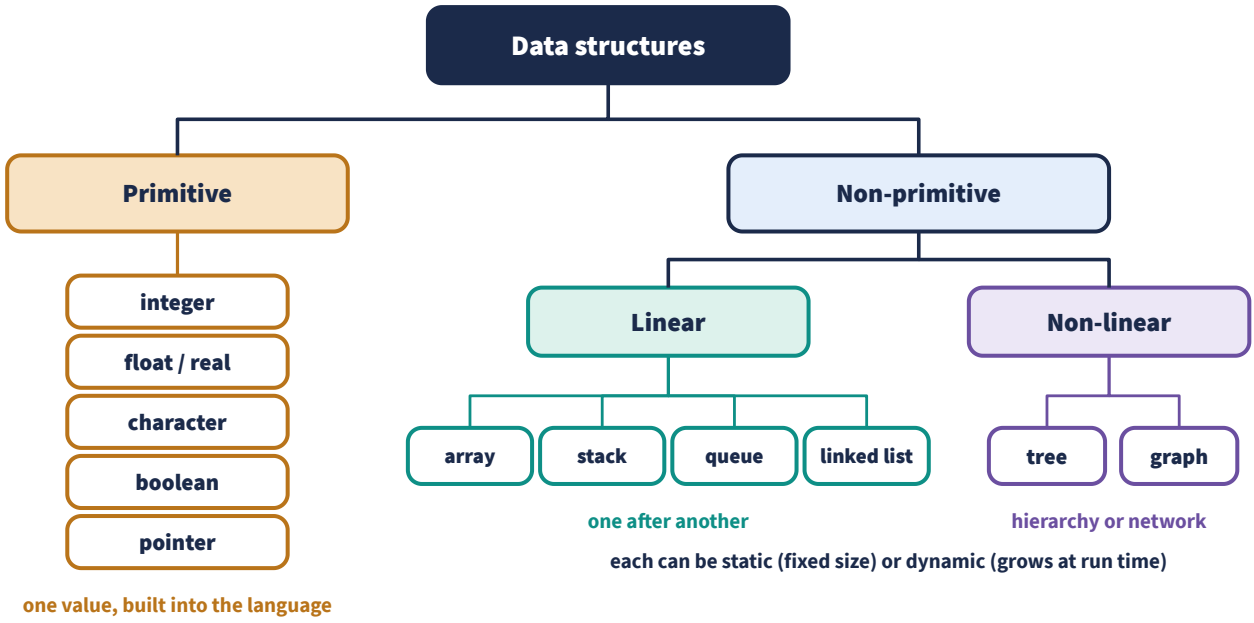


Figure 1.1 Classification of data structures. Non-primitive structures divide into linear and non-linear; any of them can be static or dynamic.

★ Primitive: one built-in value. Non-primitive: a collection with relationships.

1.4 Linear and Non-Linear Data Structures

In a **linear** data structure the elements form a sequence: every element except the first has one **predecessor**, and every element except the last has one **successor** (source p9). In the list 10, 20, 30, 40, 50 the element 30 has predecessor 20 and successor 40. A single pass from one end visits everything.

In a **non-linear** data structure an element can be related to several others: one-to-many in a **tree** (a parent has several children), many-to-many in a **graph** (a city joins many cities) (source p10). There is no single straight path through the data.

Table 1.3 Linear and non-linear structures (source p9–10)

Feature	Linear	Non-linear
arrangement	sequence, one after another	hierarchy or network
relationship	one-to-one (one predecessor, one successor)	one-to-many (tree) or many-to-many (graph)
traversal	one pass from start to end	needs a method (tree traversals, BFS, DFS)
implementation	simpler	more complex
examples	array, linked list, stack, queue	tree, graph

★ Linear: one after another. Non-linear: branches and networks.

1.5 Static and Dynamic Data Structures

A **static** data structure has a fixed size, decided when it is declared, for example int A[5] (source p11). A **dynamic** data structure grows and shrinks while the program runs, taking memory when an item is added and releasing it when an item is removed (source p12).

Table 1.4 Static and dynamic structures (source p11–12)

Feature	Static	Dynamic
size	fixed	grows and shrinks
memory allocated	before execution (compile time / declaration)	during execution (run time)
access	direct by index: fast	usually by following links
extra memory	none, but unused slots are wasted	a pointer (link) with every item
insert / delete in the middle	needs shifting	change a few links
examples	array, stack or queue built on an array	linked list, linked stack and queue, trees

A static array that is too big wastes memory; one that is too small overflows. Choose static when the amount of data is known in advance, dynamic when it changes often or cannot be predicted.

★ **Static: fixed size, fast index. Dynamic: flexible size, pays for links.**

1.6 Abstract Data Types

An **abstract data type (ADT)** is a logical model that states **what data** is stored and **what operations** are allowed on it, without saying how they are implemented (source p13). A user works through the operations only, the way a driver uses the steering wheel and pedals without knowing how the engine works (source p19).

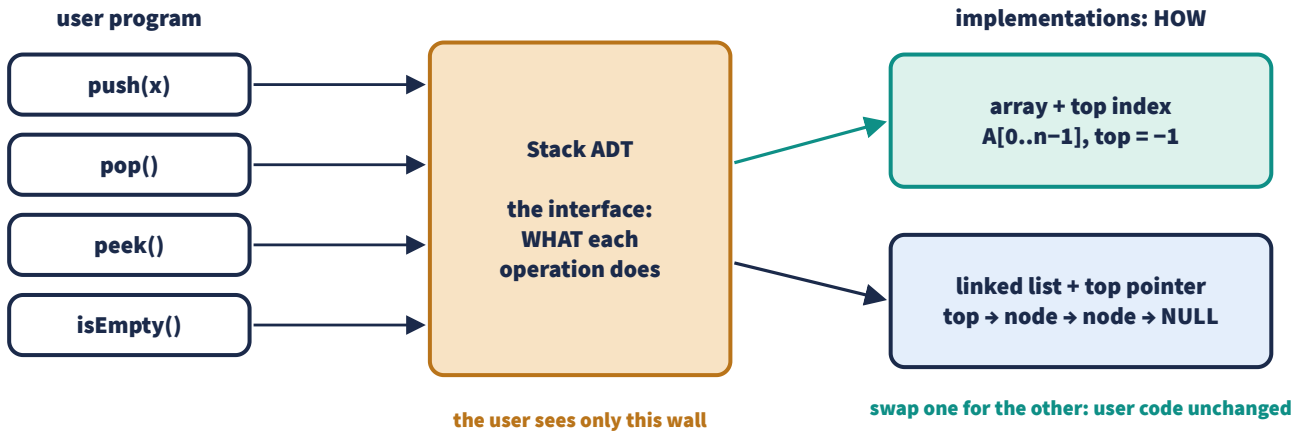


Figure 1.2 The Stack ADT as a wall between user and implementation. The same four operations can be built on an array or on a linked list.

Table 1.5 Common ADT operations (source p15)

Operation	Meaning
create	make a new, empty object of the ADT
insert / delete	add or remove an element
search	find an element
update	change an existing element
traverse	visit every element once, in order
isEmpty / size	is it empty? how many elements?

Why ADTs are needed (source p14, p19): they hide implementation details, give one standard interface, make code modular and reusable, and let the implementation change (array to linked list, say) without touching the programs that use it.

ADT (logical view)

- WHAT: data and operations
- independent of language
- example: Stack with push, pop, peek

Data structure (physical view)

- HOW: memory layout and algorithms
- written in a particular language
- example: array with a top index

★ ADT = what, not how: data plus allowed operations, implementation hidden.

1.7 The List, Stack and Queue ADTs

Three ADTs appear again and again in this unit (source p16–18).

Table 1.6 Three basic ADTs

ADT	Rule	Main operations	Access	Examples
List	ordered sequence; position matters; duplicates allowed	insert(pos, x), delete(pos), get(pos), set(pos, x), search(x), size, isEmpty	any position	to-do list, shopping list
Stack	LIFO: last in, first out	push(x), pop(), peek()/top(), isEmpty, size	top only	plates, undo, browser back button
Queue	FIFO: first in, first out	enqueue(x) at rear, dequeue() at front, front(), rear(), isEmpty, size	front and rear only	ticket line, printer queue, tokens

Worked 1.2 Stack: push 10, 20, 30, then pop

Operation	Stack (bottom → top)	Returns
`push 10`	*10*	—
`push 20`	*10 20*	—
`push 30`	*10 20 30*	—
`pop`	*10 20*	**30**

Worked 1.3 Queue: enqueue 10, 20, 30, then dequeue

Operation	Queue (front → rear)	Returns
`enqueue 10`	*10*	—
`enqueue 20`	*10 20*	—
`enqueue 30`	*10 20 30*	—
`dequeue`	*20 30*	**10**

Same three inputs, opposite outputs: the stack returns the newest item, the queue the oldest. Other common ADTs are the **Set** (unordered, no duplicates: add, remove, contains) and the **Map** or dictionary (key → value).



BRAIN POWER

A web browser's Back button and a printer both handle a sequence of items. Which one needs a stack and which a queue, and what would go wrong if you swapped them?

★ List: any position. Stack: top only, LIFO. Queue: rear in, front out, FIFO.

Who Does What?

ADT	•	• first in, first out
stack	•	• holds one built-in value
queue	•	• what, not how
primitive	•	• last in, first out

✎ Sharpen your pencil 1

1. Classify each as primitive / linear / non-linear: a bank token line; folders on a disk; the variable age; railway routes between cities. _____
2. On an empty stack: push A, push B, pop, push C, push D, pop, pop. What is popped, in order, and what is left? _____
3. Do the same operations on an empty queue (push = enqueue, pop = dequeue). _____
4. A sorted array holds 1000 roll numbers. At most how many comparisons does binary search need? _____

EXAM POINT

- Data structure = data + organization + operations.
- Primitive vs non-primitive; linear vs non-linear; static vs dynamic.
- ADT: logical model, operations without implementation.
- Stack LIFO (push/pop at top); queue FIFO (enqueue at rear, dequeue at front).

✓ Solution · Sharpen your pencil 1

1. Token line: **linear** (queue). Folders: **non-linear** (tree). age: **primitive**. Railway routes: **non-linear** (graph).
2. Popped **B, D, C**; left: **A**.
3. Dequeued **A, B, C**; left: **D**.
4. $\lfloor \log_2 1000 \rfloor + 1 = \mathbf{10}$.

Who does what: ADT → what, not how; stack → last in, first out; queue → first in, first out; primitive → holds one built-in value.

BULLET POINTS

- Arrangement decides speed.
- Primitive: one value.
- Linear: sequence; non-linear: hierarchy or network.
- Static: fixed; dynamic: grows.
- ADT hides the how.
- Brain Power: Back needs a stack (the most recent page first); the printer needs a queue (the earliest job first). Swapped, Back would jump to the first page you ever opened, and the printer would always print the newest job, possibly never reaching old ones.

Next: Chapter 2 starts with the simplest structure of all, the array: how it is laid out in memory, how an element's address is computed, and how a mostly-empty matrix is stored compactly.

CHAPTER · UNIT IV · FUNDAMENTALS OF DATA STRUCTURES

2 Arrays and Sparse Matrices

A row of numbered lockers: you can open locker 37 without touching the other 36. That single property makes the array the fastest structure there is, and also the most stubborn.

OPENING SITUATION

A teacher stores 60 students' marks in marks[60]. Reading any student's mark is instant: the computer works out the address with one multiplication. Then a new student joins in the middle of the roll list, and 30 entries must shuffle along to make room. A railway reservation chart has 1000 rows and 72 berths, but on a quiet day only 400 of the 72 000 cells are filled. Storing all 72 000 wastes memory. This chapter covers both sides of the array: why access is instant, what insertion and deletion cost, how 2-D arrays are laid out (row-major and column-major), and how a **sparse matrix** is stored as triplets.

IN THIS CHAPTER

- 2.1 What an Array Is
- 2.2 Declaring and Initializing Arrays
- 2.3 One-, Two- and Multi-Dimensional Arrays
- 2.4 Memory Representation and Address Calculation
- 2.5 Traversal, Insertion and Deletion
- 2.6 Searching in Arrays
- 2.7 Applications of Arrays
- 2.8 Matrices and Sparse Matrices
- 2.9 Sparse Matrix Representation: Triplet Form

YOU NEED

- Static structures and the List ADT (Chapter 1).
- Memory addresses and the size of a data type in bytes.

2.1 What an Array Is

An **array** is a collection of elements of the **same data type** stored in **contiguous** (adjacent) memory locations and accessed by an **index** (source p20). In C-family languages the index starts at 0, so an array of n elements has indices 0 to n − 1.

Homogeneous
Every element has the same type, so every element has the same size.

Contiguous
Elements sit side by side in memory; no gaps, no links.

Indexed
Any element is reached directly, in constant time, by its position.

The price is a **fixed size**, set when the array is created: too large wastes memory, too small overflows.

★ **Array: one name, many values of one type, side by side, reached by index.**

2.2 Declaring and Initializing Arrays

A **declaration** states the type, the name and the size, and reserves memory: dataType arrayName[size]; (source p21). The size must be a positive integer. **Initialization** gives starting values, at declaration or later (source p22).

Table 2.1 Declarations and what they create

Declaration	Elements	Memory (int and float 4 bytes, char 1)	Contents
`int marks[5];`	marks[0] ... marks[4]	20 bytes	undefined until assigned (local array in C)
`float price[10];`	price[0] ... price[9]	40 bytes	undefined
`char grade[3];`	grade[0] ... grade[2]	3 bytes	undefined
`int marks[5] = {10, 20, 30, 40, 50};`	full initialization	20 bytes	10 20 30 40 50
`int a[5] = {3, 6};`	partial initialization	20 bytes	3 6 0 0 0 (the rest become 0)

★ **Declaration = type + name + size; initialization fills from index 0, and a partial list zero-fills the rest.**

2.3 One-, Two- and Multi-Dimensional Arrays

A **one-dimensional** array is a single row reached with one index: nums[3] is the fourth element (source p23). A **two-dimensional** array arranges elements in rows and columns like a table and needs two indices, [row][column]: int marks[3][4] has 3 rows and 4 columns, 12 elements (source p24). A **multi-dimensional** array has three or more indices: int data[2][3][4] is 2 layers of 3 × 4 tables, 24 elements (source p25).

Table 2.2 Dimensions at a glance (source p23–25)

Kind	Example	Elements	Picture	Typical use
1-D	`int nums[5]`	5	a row	marks, prices, daily readings
2-D	`int marks[3][4]`	3 × 4 = 12	a table / matrix	marks per subject, timetables, matrices
3-D	`int data[2][3][4]`	2 × 3 × 4 = 24	a stack of tables / a cube	images (height × width × colour), game levels

★ **One index per dimension; total elements = product of the sizes.**

2.4 Memory Representation and Address Calculation

Because the elements are contiguous and equal in size, the address of any element follows from the address of the first, the **base address** B (source p26).

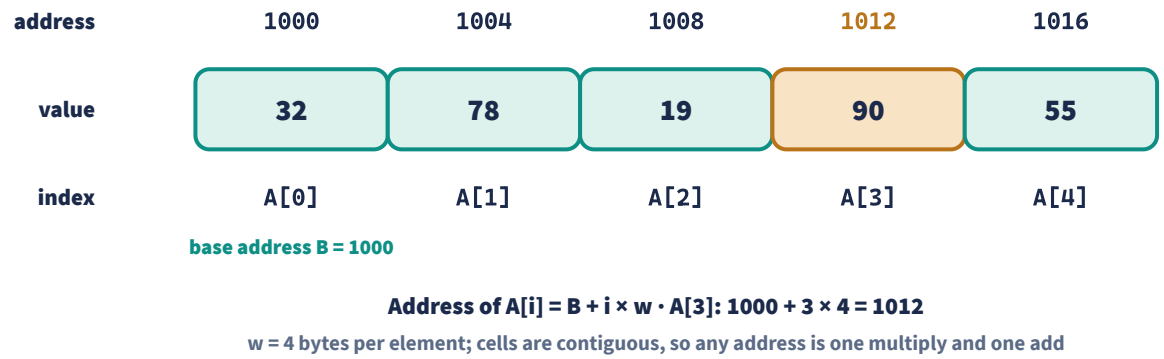


Figure 2.1 A five-element int array at base address 1000. Each step of the index adds w = 4 to the address.

$$\text{Address of } A[i] = B + w \times (i - LB) \quad \cdot \quad LB = \text{lower bound of the index (0 in C, so simply } B + w{\cdot}i)$$

Memory itself is one long row, so a 2-D array must be laid out row after row or column after column.

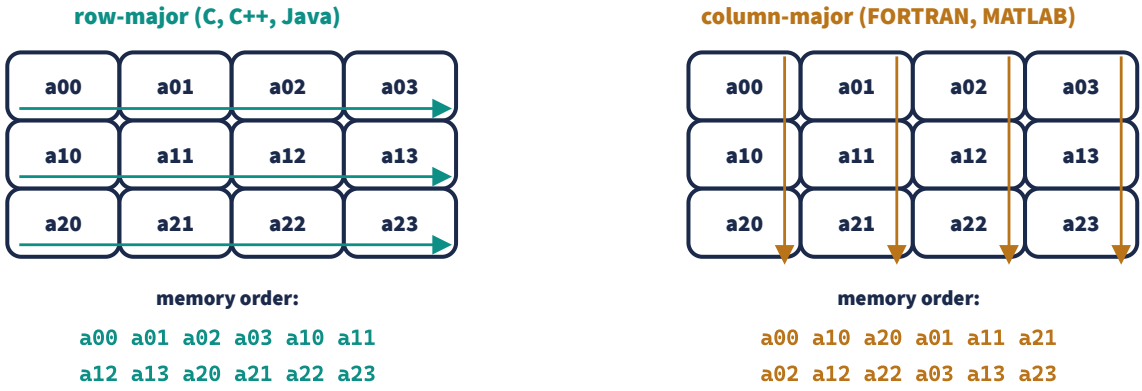


Figure 2.2 The same 3 × 4 matrix stored in row-major and in column-major order.

$$\text{row-major: } A[i][j] = B + w \times [\,(i - LR) \times n + (j - LC)\,] \quad \cdot \quad \text{column-major: } A[i][j] = B + w \times [\,(j - LC) \times m + (i - LR)\,]$$

Here m is the number of rows, n the number of columns, and LR, LC the lower bounds of the row and column index.

Worked 2.1 Address calculations

Array	Element	Working	Address
int A[0..9][0..19], B = 1000, w = 4, row-major	A[3][5]	1000 + 4 × (3 × 20 + 5)	**1260**
same, column-major	A[3][5]	1000 + 4 × (5 × 10 + 3)	**1212**
A[1..10][1..15], B = 2000, w = 2, row-major	A[4][6]	2000 + 2 × ((4−1) × 15 + (6−1))	**2100**
same, column-major	A[4][6]	2000 + 2 × ((6−1) × 10 + (4−1))	**2106**
int data[2][3][4], row-major	data[1][2][3]	B + 4 × (1 × 12 + 2 × 4 + 3)	B + **92** (the last of 96 bytes)

★ Address = base + size × (elements before it); row-major counts whole rows first, column-major whole columns.

2.5 Traversal, Insertion and Deletion

Traversal visits every element once, from index 0 to n − 1, to print, sum, count or update (source p27). For 10 20 30 40 50 a traversal that adds as it goes gives 150.

Insertion at position k needs a free slot and a gap: every element from k to the end moves one place right, starting from the last (source p28).

1. If n equals the capacity, report **overflow** and stop.
2. For i = n − 1 down to k: A[i + 1] ← A[i] (shift right, last element first).
3. A[k] ← x; n ← n + 1.

Deletion at position k closes the gap: every element after k moves one place left, starting from k + 1 (source p29).

1. If n = 0, report **underflow** and stop.
2. x ← A[k]; for i = k to n − 2: A[i] ← A[i + 1] (shift left).
3. n ← n − 1.

Worked 2.2 Insert 30 at index 2, then delete it again (source p28–29)

Step	Array	Elements moved
start (n = 4)	`10 20 40 50`	—
shift 50, then 40, right	`10 20 __ 40 50`	n − k = 4 − 2 = 2
place 30 (n = 5)	`10 20 30 40 50`	—
delete index 2: shift 40, then 50, left (n = 4)	`10 20 40 50`	n − 1 − k = 5 − 1 − 2 = 2

Table 2.3 Cost of array operations (n elements)

Operation	Best case	Worst case	Order
access A[i]	1 step	1 step	O(1)
traversal	n	n	O(n)
insert at k	at the end: 0 moves	at the front: n moves	O(n)
delete at k	the last: 0 moves	the first: n − 1 moves	O(n)

★ **Access is O(1); insertion and deletion are O(n) because of shifting.**

2.6 Searching in Arrays

Linear search compares the target with each element from the start until it is found or the array ends; it works on any array (source p30). **Binary search** needs a **sorted** array: compare with the middle element and keep only the half that can contain the target.

Worked 2.3 Find 40 in 10 20 30 40 50

Method	Steps	Comparisons
linear	10 ≠ 40, 20 ≠ 40, 30 ≠ 40, 40 = 40 (index 3)	4
binary	low 0, high 4, mid 2: 30 < 40 → low = 3; low 3, high 4, mid 3: 40 = 40 (index 3)	2

Table 2.4 Linear and binary search (source p30)

Aspect	Linear search	Binary search
requirement	none: sorted or unsorted	sorted array
method	check one by one	compare with the middle, halve the range
worst-case comparisons	n	$\lfloor \log_2 n \rfloor + 1$
time	O(n)	O(log n)
best for	small or unsorted data	large sorted data

★ **Linear: any array, O(n). Binary: sorted array, O(log n).**

2.7 Applications of Arrays

Arrays hold any fixed-size list of similar values: marks, monthly sales, roll numbers, temperatures, inventory counts (source p31). In computing they represent matrices and tables, store image pixels, are the working space of searching and sorting algorithms, form the tables of dynamic programming (Chapter 28), and are used to **implement** other structures: stacks, queues, heaps and hash tables.

★ **Arrays: the base for tables, images, sorting, DP tables and many other structures.**

2.8 Matrices and Sparse Matrices

A **matrix** is a rectangular arrangement of elements in m rows and n columns; m × n is its **order** (source p32). Row matrix 1 × n, column matrix m × 1, square matrix n × n, rectangular m ≠ n. In a program a matrix is a 2-D array.

> **WATCH THE MODEL**
>
> Source p32 shows A = [3 6 9; 2 5 8; 1 4 7] and says the element in row 2, column 3 is A[2][3] = 8. That is **mathematical** notation, counting from 1. In C, Java or Python, which count from 0, the same element is **A[1][2]**; A[2][3] would be outside a 3 × 3 array.

A **sparse matrix** is one in which most elements are zero (source p33). A matrix with mostly non-zero values is called **dense** (a "normal" matrix in the source, p35). Storing every zero wastes memory, and processing them wastes time (source p34).

Table 2.5 Dense and sparse matrices (source p35)

Aspect	Dense (normal)	Sparse
zeros	few or none	most elements
best storage	ordinary 2-D array: m × n cells	non-zero entries only, with their positions
processing	every cell	only the non-zero entries

★ **Sparse = mostly zeros; store only what is not zero.**

2.9 Sparse Matrix Representation: Triplet Form

The usual representation is the **triplet** (3-tuple) form: one row (row, column, value) for each non-zero element, in row order, under a **header row** that gives the number of rows, the number of columns and the number of non-zero elements (source p36–37).

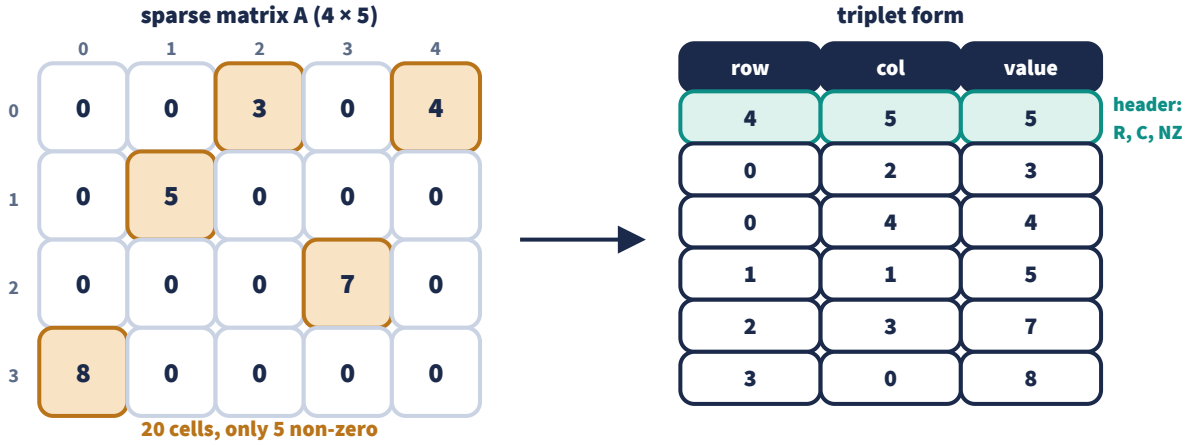


Figure 2.3 The source's 4 × 5 example in triplet form, 0-based indices. The header is 4, 5, 5: four rows, five columns, five non-zero values.

WATCH IT! · COMMON MISTAKE 2.1 — COUNT THE NON-ZEROS (SOURCE P37)

The source's header row reads 4, 5, **4** and the caption says "4 non-zero elements", yet the matrix holds 3, 4, 5, 7 and 8, and the source's own table lists five triplets. The header must be **4, 5, 5**. The header count is what a program uses to know how many triplets to read, so a wrong count loses data.

Worked 2.4 Does the triplet form save memory? (4-byte integers)

Matrix	Full 2-D array	Triplets (t + 1 rows × 3)	Saving
4 × 5, t = 5	20 cells = 80 bytes	18 cells = 72 bytes	small
100 × 100, t = 50	10 000 cells	153 cells	98 %

Triplets win only while $3(t + 1) < m \times n$, that is, while fewer than about a third of the cells are non-zero.

Worked 2.5 Transpose in triplet form: swap row and column, then re-sort by the new row

Original (row, col, value)	Swapped	Transpose, sorted (header 5, 4, 5)
`(0,2,3) (0,4,4) (1,1,5) (2,3,7) (3,0,8)`	`(2,0,3) (4,0,4) (1,1,5) (3,2,7) (0,3,8)`	`(0,3,8) (1,1,5) (2,0,3) (3,2,7) (4,0,4)`

Sparse matrices are used for graph adjacency matrices, image processing, scientific and finite-element computation, machine learning, recommendation systems and document–term matrices in text processing (source p38). Other compact forms are linked representations (one node per non-zero) and the compressed sparse row (CSR) format used by numerical libraries.

BRAIN POWER

A 4 × 5 matrix is stored in triplet form. Up to how many non-zero elements does that still use less memory than the plain 2-D array?

★ **Triplet: header (rows, cols, count) + one (row, col, value) per non-zero, in row order.**

Who Does What?

base address	•	• rows stored one after another
row-major	•	• (row, column, value) per non-zero
binary search	•	• address of the first element
triplet	•	• needs a sorted array

✎ Sharpen your pencil 2

1. int A[50] starts at 2000 (w = 4). Find the address of A[17]. _____
2. float M[5][8] starts at 3000 (w = 4). Find M[3][6] in row-major and in column-major order. _____
3. In 5 15 25 35 45 55 (n = 6): how many elements move to insert 30 at index 3? To delete index 1 from the original? _____
4. Write the triplet form of [0 0 6; 0 0 0; 9 0 0]. _____

EXAM POINT

- A[i] = B + w(i − LB); row-major B + w[(i−LR)n + (j−LC)]; column-major B + w[(j−LC)m + (i−LR)].
- Access O(1); insertion and deletion O(n) (shifting).
- Binary search needs sorted data: $\lfloor \log_2 n \rfloor + 1$ comparisons at most.
- Sparse matrix: triplet header = rows, columns, number of non-zeros.

✓ Solution · Sharpen your pencil 2

1. 2000 + 4 × 17 = **2068**.
2. Row-major 3000 + 4 × (3 × 8 + 6) = **3120**; column-major 3000 + 4 × (6 × 5 + 3) = **3132**.
3. Insert: 6 − 3 = **3** moves (35, 45, 55) giving 5 15 25 30 35 45 55. Delete index 1: 6 − 1 − 1 = **4** moves giving 5 25 35 45 55.
4. Header **3 3 2**; then **(0, 2, 6)** and **(2, 0, 9)**.

Who does what: base address → address of the first element; row-major → rows stored one after another; binary search → needs a sorted array; triplet → (row, column, value) per non-zero.

BULLET POINTS

- Same type, contiguous, indexed, fixed size.
- Address = B + w × offset.
- Row-major in C; column-major in FORTRAN.
- Shifting makes insert/delete O(n).
- Sparse: keep only non-zeros.
- Brain Power: 3(t + 1) < 20 gives t ≤ 5. With 5 non-zeros it is 72 bytes against 80; with 6 it is 84, already worse.

Next: Chapter 3 restricts an array to one end. Allowing insertion and deletion only at the top turns it into a stack, the structure behind function calls, undo and expression evaluation.

CHAPTER · UNIT IV · FUNDAMENTALS OF DATA STRUCTURES

3 Stacks

Ctrl + Z, the browser's Back button, and every function call your program makes all use the same rule: the last thing in is the first thing out.

OPENING SITUATION

You type "Hello", make it bold, then change the colour. You press Ctrl + Z three times. The colour goes first, then the bold, then the text: exactly the reverse of the order you did them. The editor never searches its history. It keeps the actions in a pile and always takes the one on top. That pile is a **stack**. This chapter defines the stack and its five operations, shows both ways of building one (array and linked list), and then works through the jobs stacks do best: function calls, expression conversion and evaluation, and bracket checking.

IN THIS CHAPTER

- 3.1 What a Stack Is
- 3.2 The LIFO Principle
- 3.3 Stack Operations
- 3.4 Overflow and Underflow
- 3.5 Array Implementation of a Stack
- 3.6 Linked List Implementation of a Stack
- 3.7 Applications of Stacks
- 3.8 Infix, Prefix and Postfix Notation
- 3.9 Converting Infix to Postfix
- 3.10 Evaluating a Postfix Expression
- 3.11 Checking Balanced Brackets

YOU NEED

- The Stack ADT and linear structures (Chapter 1).
- Array indexing and fixed capacity (Chapter 2).

3.1 What a Stack Is

A **stack** is a linear data structure in which insertion and deletion happen at **one end only**, called the **top** (source p39). The other end, the **bottom**, is never touched directly. Its parts are the **top** (where all action happens), the **bottom**, the **elements**, and the **size** or capacity, the largest number of elements it can hold.

A stack is **linear** because its elements sit in one sequence, one above another: every element except the bottom has exactly one element below it, and every element except the top has exactly one above it. There is no branching, unlike a tree or graph (source p40).

One door
Everything enters and leaves through the top.

Order kept
Elements below the top stay in the order they arrived.

Restricted access
Only the top element can be seen or removed; no index, no search.

★ Stack: a linear list with a single access end, the top.

3.2 The LIFO Principle

LIFO means **Last In, First Out**: the most recently inserted element is the first to be removed (source p41). Push 10, 20, 30; the first pop gives 30, the next 20, the last 10. The removal order is always the reverse of the insertion order. A pile of plates in a canteen works the same way: the newest plate goes on top and is taken first.

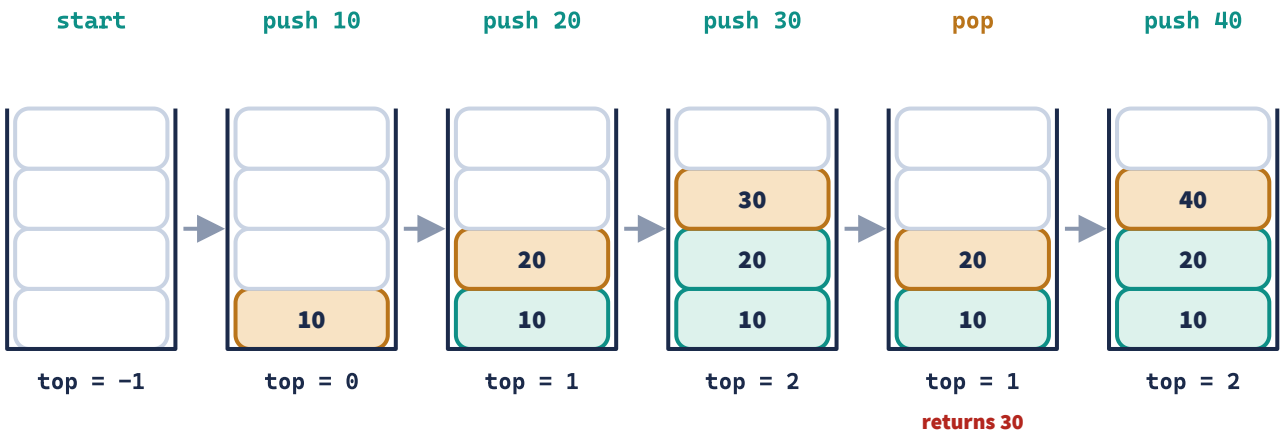


Figure 3.1 A stack of capacity 4 through five operations. The highlighted cell is the top; the top index moves up on push and down on pop.

★ LIFO: removal order is the reverse of insertion order.

3.3 Stack Operations

Five operations manage a stack, and every one works only on the top (source p42).

Table 3.1 The stack operations (source p42–45)

Operation	What it does	Changes the stack?	Error case	Time
push(x)	inserts x on the top; x becomes the new top	yes, size + 1	overflow if full	O(1)
pop()	removes and returns the top element	yes, size − 1	underflow if empty	O(1)
peek() / top()	returns the top element without removing it	no	underflow if empty	O(1)
isEmpty()	true if the stack has no elements	no	—	O(1)
isFull()	true if the stack has reached its capacity	no	—	O(1)

Push, step by step (source p43):

1. Check whether the stack is full. If it is, report **overflow** and stop.
2. Move the top one place up.
3. Store the new value at the top. It is now the top element.

Pop, step by step (source p44):

1. Check whether the stack is empty. If it is, report **underflow** and stop.
2. Read the top element.
3. Move the top one place down; the element below becomes the new top. Return the value read.

Peek checks for empty and then reads the top value, leaving the stack unchanged (source p45). It is used to look before deciding whether to pop, as the infix-to-postfix conversion in Section 3.9 does at every operator.

★ **Push adds, pop removes, peek looks; all at the top, all O(1).**

3.4 Overflow and Underflow

Overflow happens when a push is attempted on a **full** stack. **Underflow** happens when a pop or peek is attempted on an **empty** stack (source p46). Overflow is a limit of fixed-size storage; underflow is a logic error, asking for something that is not there. The top pointer detects both, so every push must check "full?" first and every pop or peek must check "empty?" first.

Overflow

Stack full, push attempted. Array stack: top = MAX − 1. Linked stack: no memory left for a new node. Plate rack already full and one more plate arrives.

Underflow

Stack empty, pop or peek attempted. Array stack: top = −1. Linked stack: TOP = NULL. Reaching for a plate on an empty rack.

★ **Full + push = overflow; empty + pop/peek = underflow.**

3.5 Array Implementation of a Stack

The simplest stack is an array S[0 ... MAX − 1] plus one integer, **top**, holding the index of the top element (source p47). Initially top = −1, meaning empty.

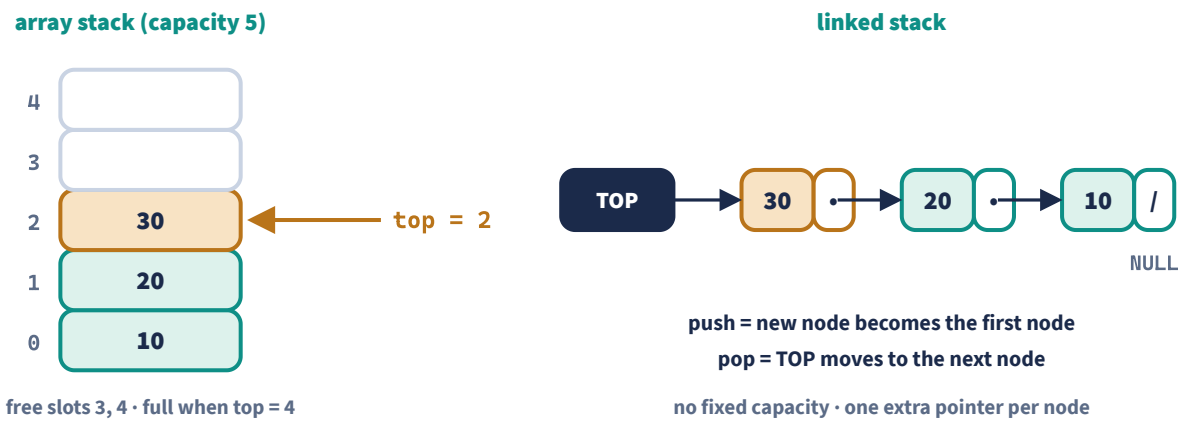


Figure 3.2 The same stack, 10 then 20 then 30, stored in an array with a top index and as a linked list with a TOP pointer.

Table 3.2 The array stack rules

Item	Rule
empty	top = −1
full	top = MAX − 1
push x	if full → overflow; else top ← top + 1, S[top] ← x
pop	if empty → underflow; else x ← S[top], top ← top − 1, return x
peek	if empty → underflow; else return S[top]
number of elements	top + 1

Worked 3.1 An array stack with MAX = 3, traced operation by operation

Operation	Check	top after	S[0] S[1] S[2] (live part)	Result
`push 10`	top −1 ≠ 2	0	`10`	—
`push 20`	top 0 ≠ 2	1	`10 20`	—
`push 30`	top 1 ≠ 2	2	`10 20 30`	stack now full
`push 40`	top = 2 = MAX − 1	2	`10 20 30`	**overflow**
`pop`	top 2 ≠ −1	1	`10 20`	30
`peek`	top 1 ≠ −1	1	`10 20`	20
`pop`	top 1 ≠ −1	0	`10`	20
`pop`	top 0 ≠ −1	−1	(empty)	10
`pop`	top = −1	−1	(empty)	**underflow**

> **WATCH THE MODEL**
>
> Pop does not erase anything. After the first pop above, the value 30 is still physically in S[2]; it is simply above the top, so it no longer belongs to the stack, and the next push overwrites it. The stack is defined by top, not by what the array cells happen to hold.

The array stack is simple and fast, with direct indexing and no pointer memory, but its capacity is fixed, so it can overflow while the rest of memory is free (source p47).

★ **Array stack = array + top index; top = −1 empty, top = MAX − 1 full.**

3.6 Linked List Implementation of a Stack

A linked stack stores each element in a **node** with a data field and a next pointer. A **TOP** pointer points to the first node, which is the top of the stack; the last node's next pointer is NULL (source p48). Push and pop both work at the head of the list, so neither needs to walk the list.

1. **Push x:** create a new node N; N.data ← x; N.next ← TOP; TOP ← N. If no node can be created, that is overflow.
2. **Pop:** if TOP = NULL, underflow. Otherwise x ← TOP.data; T ← TOP; TOP ← TOP.next; free T; return x.
3. **Peek:** if TOP = NULL, underflow; otherwise return TOP.data.

Worked 3.2 Push 40, then pop, on the linked stack 30 → 20 → 10 (source p48)

Step	List from TOP
start	`TOP → 30 → 20 → 10 → NULL`
push 40: N.next ← TOP (points to 30), TOP ← N	`TOP → 40 → 30 → 20 → 10 → NULL`
pop: x ← 40, TOP ← TOP.next, free old node	`TOP → 30 → 20 → 10 → NULL, returns 40`

WATCH IT! · COMMON MISTAKE 3.1 — LINK FIRST, THEN MOVE TOP

In push, the order N.next ← TOP and then TOP ← N matters. Moving TOP to the new node first makes N.next point to N itself and loses the whole old stack. In pop, save the old node before moving TOP, or it can never be freed.

Table 3.3 Array stack and linked stack

Aspect	Array stack	Linked stack
size	fixed capacity MAX	grows and shrinks as needed
overflow	when top = MAX − 1	only when memory runs out
memory per element	data only	data + one pointer
memory layout	contiguous	nodes anywhere in memory
push / pop / peek	O(1)	O(1)
best when	maximum size is known	size varies widely

★ **Linked stack = nodes + TOP pointer; push and pop at the head.**

3.7 Applications of Stacks

A stack is used wherever the most recent unfinished item must be handled first (source p49).

Table 3.4 Where stacks are used (source p49)

Area	What is pushed	What a pop means
function calls and recursion	a stack frame per call: parameters, local variables, return address	the call returns to its caller
undo in an editor (Ctrl + Z)	each action as it is done	undo the latest action (a second stack holds redo)
browser Back button	each page left behind	go back to the previous page
expression conversion and evaluation	operators or operands	apply the most recent operator (Sections 3.8–3.10)
syntax parsing, bracket matching	each opening bracket	a closing bracket matches the latest opener (Section 3.11)
backtracking, depth-first search	each choice point or vertex	return to the latest choice and try the next option
reversing a sequence	each item in order	items come out reversed: S T A C K → K C A T S

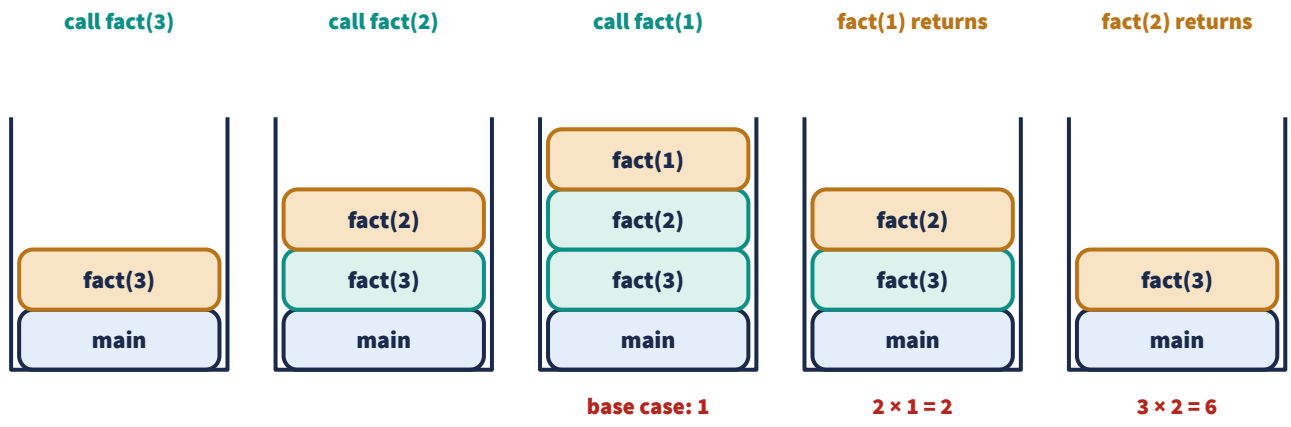


Figure 3.3 The call stack while fact(3) runs, with fact(n) = n × fact(n − 1) and fact(1) = 1. Calls push frames; returns pop them and pass the value down.

Recursion that never reaches its base case keeps pushing frames until the call stack's memory is used up. That runtime error is called **stack overflow**, the same overflow as Section 3.4.

★ Wherever the latest unfinished item comes first, a stack is used.

3.8 Infix, Prefix and Postfix Notation

An arithmetic expression can be written in three ways, depending on where the operator goes relative to its two operands.

Table 3.5 The three notations

Notation	Operator position	A + B	(A + B) × C	A + B × C
infix	between the operands	`A + B`	`(A + B) * C`	`A + B * C`
prefix (Polish)	before the operands	`+ A B`	`* + A B C`	`+ A * B C`
postfix (reverse Polish)	after the operands	`A B +`	`A B + C *`	`A B C * +`

Infix needs precedence rules and brackets to be unambiguous. Prefix and postfix need **neither**: the order of the symbols alone fixes the order of evaluation. That is why compilers convert infix to postfix and evaluate postfix with a stack (source p94).

Table 3.6 Operator precedence and associativity used in conversion

Operator	Precedence	Associativity
`^`	highest	right to left: A ^ B ^ C = A ^ (B ^ C)
`* / %`	middle	left to right
`+ −`	lowest	left to right: A − B − C = (A − B) − C

★ Postfix puts the operator after its operands and needs no brackets.

3.9 Converting Infix to Postfix

Scan the infix expression left to right. Operands go straight to the output; operators wait on a stack until no operator of higher priority can still come before them.

1. **Operand:** append it to the output.
2. **"(":** push it.
3. **")":** pop operators to the output until "(" is reached; pop the "(" and discard both brackets.
4. **Operator op:** while the top of the stack is an operator of higher precedence, or of equal precedence and op is left-associative, pop it to the output. Then push op.
5. **End of input:** pop every remaining operator to the output.

Worked 3.3 Convert A + B * C − D / E

Symbol	Action	Stack (bottom → top)	Output
A	output		A
+	stack empty: push	+	A
B	output	+	A B
*	* is higher than +: push	+ *	A B
C	output	+ *	A B C
−	pop * (higher), pop + (equal, left-assoc.), push −	−	A B C * +
D	output	−	A B C * + D
/	/ is higher than −: push	− /	A B C * + D
E	output	− /	A B C * + D E
end	pop all		**A B C * + D E / −**

Worked 3.4 Convert (A + B) * (C − D): brackets override precedence

Symbol	Action	Stack	Output
(A + B	push (, output A, push +, output B	(+	A B
)	pop + to output, discard (		A B +
*	push	*	A B +
(C − D	push (, output C, push −, output D	* (−	A B + C D
)	pop − to output, discard (	*	A B + C D −
end	pop *		**A B + C D − ***

The source's compiler-design page converts (3 + 4) * 2 − 5 to 3 4 + 2 * 5 − (source p94); the same steps give that result.

★ **Operands out at once; operators wait on the stack until nothing stronger is pending.**

3.10 Evaluating a Postfix Expression

Scan the postfix expression left to right with one stack of values.

1. **Operand:** push its value.

2. **Operator:** pop the **right** operand first, then the **left** operand; compute left op right; push the result.
3. **End:** the single value left on the stack is the answer.

Worked 3.5 **Evaluate 6 2 3 + − 3 8 2 / + ***

Symbol	Action	Stack (bottom → top)
`6 2 3`	push each	`6 2 3`
`+`	pop 3, pop 2, 2 + 3 = 5, push	`6 5`
`−`	pop 5, pop 6, 6 − 5 = 1, push	`1`
`3 8 2`	push each	`1 3 8 2`
`/`	pop 2, pop 8, 8 / 2 = 4, push	`1 3 4`
`+`	pop 4, pop 3, 3 + 4 = 7, push	`1 7`
`*`	pop 7, pop 1, 1 × 7 = 7, push	`7`
end	answer	**7**

WATCH IT! · COMMON MISTAKE 3.2 — THE FIRST POP IS THE RIGHT OPERAND

For 8 2 / the stack pops 2 first and 8 second, and the answer is 8 / 2 = 4, not 2 / 8. Addition and multiplication hide this slip because they are commutative; subtraction, division and powers expose it.

Prefix is evaluated the same way but scanned from **right to left**, and then the first value popped is the **left** operand.

★ **Postfix evaluation: push operands; at an operator pop right, pop left, push the result.**

3.11 Checking Balanced Brackets

Compilers and editors check that every opening bracket has a matching closing bracket in the right order. A stack does it in one pass.

1. Opening bracket ([{: push it.
2. Closing bracket)] }: if the stack is empty, or the popped opener is not of the same kind, the expression is **unbalanced**.
3. At the end the expression is balanced only if the stack is empty.

Worked 3.6 **Three bracket checks**

Input	Trace (stack after each symbol)	Verdict
`{ ( [ ] ) }`	`{ · {( · {([ · {( · { · empty`	balanced
`( [ ) ]`	`( · ([ · ")" meets "[" → mismatch`	unbalanced
`( ( )`	`( · (( · ( · end with "(" left`	unbalanced

BRAIN POWER

A browser has both a Back and a Forward button. How can two stacks provide both, and what must happen to the Forward stack when you click a new link?

★ **Push openers; each closer must match the latest opener; the stack must end empty.**

Who Does What?

LIFO	•	• pop or peek on an empty stack
overflow	•	• read the top without removing it
underflow	•	• last in, first out
peek	•	• push on a full stack

✎ Sharpen your pencil 3

1. On an empty stack: push 5, push 8, pop, push 3, push 9, pop, peek. List the values returned and the final stack. _____
2. Convert A * (B + C) / D to postfix. _____
3. Evaluate the postfix expression 5 3 + 8 2 − *. _____
4. An array stack has MAX = 6 and starts with top = −1. After 4 pushes and 1 pop, what is top, and how many elements are in the stack? _____

EXAM POINT

- Stack = LIFO; insertion and deletion at the top only; all operations O(1).
- Array stack: empty top = −1, full top = MAX − 1, size = top + 1.
- Infix → postfix uses an operator stack; postfix evaluation uses an operand stack.
- Uses: recursion and function calls, undo, Back button, bracket matching, DFS and backtracking.

✓ Solution · Sharpen your pencil 3

1. Pop returns **8**, the second pop returns **9**, peek returns **3**. Final stack, bottom to top: **5 3**.
2. **A B C + * D /**: * waits, the bracket gives B C +, / pops * (equal precedence, left-associative), D is output, end pops /.
3. 5 + 3 = 8; 8 − 2 = 6; 8 × 6 = **48**.
4. top = −1 + 4 − 1 = **2**; elements = top + 1 = **3**.

Who does what: LIFO → last in, first out; overflow → push on a full stack; underflow → pop or peek on an empty stack; peek → read the top without removing it.

BULLET POINTS

- One access end: the top.
- Push / pop / peek / isEmpty / isFull, all O(1).
- Check full before push, empty before pop.
- Array: fixed MAX. Linked: grows, costs a pointer.
- Call stack, postfix, brackets, undo, DFS.
- Brain Power: Back pops the current page onto the Forward stack; Forward pops it back. A new link pushes onto Back and empties the Forward stack.

Next: Chapter 4 opens the other end. A queue adds at the rear and removes at the front, first in first out, and it leads to circular queues, deques and priority queues.

CHAPTER · UNIT IV · FUNDAMENTALS OF DATA STRUCTURES

4 Queues and Priority Queues

The ticket counter serves whoever came first. The hospital emergency room does not. Both are queues; the second one has priorities.

OPENING SITUATION

Five students send files to one printer within a second. The printer cannot print them together, so the operating system lines them up and prints them in arrival order: a **queue**. Then the principal sends a one-page notice marked urgent, and it should jump the line: a **priority queue**. Inside the computer the same ideas run the CPU's ready list, keyboard buffers and network routers. This chapter builds the queue from its two pointers, finds the flaw in the simple array queue and fixes it with the circular queue, opens both ends with the deque, and ends with priority queues and how they are stored.

IN THIS CHAPTER

- 4.1 What a Queue Is
- 4.2 Queue Operations
- 4.3 Front and Rear in an Array: the Simple Queue
- 4.4 Circular Queue
- 4.5 Linked Queue
- 4.6 Double-Ended Queue (Deque)
- 4.7 Priority Queue: Definition and Need
- 4.8 Ascending and Descending Priority Queues
- 4.9 Priority Queue Operations and Implementations
- 4.10 Applications of Queues and Priority Queues

YOU NEED

- Stacks, overflow and underflow (Chapter 3).
- Array indices and the mod operator (Chapter 2).

4.1 What a Queue Is

A **queue** is a linear data structure that follows the **FIFO** principle, **First In, First Out**: insertion happens at one end, the **rear**, and deletion at the other end, the **front** (source p50). The first element inserted is the first removed, and the order of arrival is preserved (source p51). A bus queue or a ticket counter works this way: the person who came first is served first, and newcomers join at the end.

Stack (LIFO)
One end, the top, for both insertion and deletion. Removal order is the reverse of arrival. Used for undo, recursion, backtracking.

Queue (FIFO)
Two ends: insert at the rear, delete at the front. Removal order equals arrival order. Used for fair, orderly service: printing, CPU scheduling, buffers.

★ **Queue: insert at the rear, delete at the front, first in first out.**

4.2 Queue Operations

Table 4.1 The queue operations (source p52)

Operation	What it does	Error case	Time
enqueue(x)	inserts x at the rear	overflow if full	O(1)
dequeue()	removes and returns the front element	underflow if empty	O(1)
front() / peek()	returns the front element without removing it	underflow if empty	O(1)
isEmpty()	true when the queue has no elements	—	O(1)
isFull()	true when the queue has reached its capacity	—	O(1)

Enqueue (source p53): check whether the queue is full (if so, **queue overflow**); move REAR to the next position; store the new element at REAR. **Dequeue** (source p54): check whether the queue is empty (if so, **queue underflow**); take the element at FRONT; move FRONT to the next element. The remaining elements keep their relative order.

Worked 4.1 Operations on an empty queue (source p52)

Operation	Queue (front → rear)	Returned
`enqueue 10`	`10`	—
`enqueue 20`	`10 20`	—
`enqueue 30`	`10 20 30`	—
dequeue	`20 30`	10
peek	`20 30`	20

★ **Enqueue at the rear, dequeue at the front; both O(1).**

4.3 Front and Rear in an Array: the Simple Queue

An array queue Q[0 … N − 1] uses two indices: **FRONT**, the index of the first element (the deletion end), and **REAR**, the index of the last element (the insertion end). Both start at −1 (source p55). FRONT moves forward on dequeue, REAR moves forward on enqueue; when only one element is present, FRONT = REAR.

Table 4.2 Simple (linear) queue rules, with FRONT = REAR = −1 for empty

Item	Rule
empty	FRONT = −1
full	REAR = N − 1
enqueue x	if REAR = N − 1 → overflow; if FRONT = −1 then FRONT ← 0; REAR ← REAR + 1; Q[REAR] ← x
dequeue	if FRONT = −1 → underflow; x ← Q[FRONT]; if FRONT = REAR then FRONT ← REAR ← −1 else FRONT ← FRONT + 1
number of elements	REAR − FRONT + 1 (when not empty)

Worked 4.2 FRONT and REAR, step by step (source p55)

Operation	Queue	FRONT	REAR
start	empty	−1	−1
`enqueue 10`	`10`	0	0
`enqueue 20`	`10 20`	0	1
`enqueue 30`	`10 20 30`	0	2
`dequeue → 10`	`20 30`	1	2
`enqueue 40`	`20 30 40`	1	3

WATCH THE MODEL

Source p54 tests for an empty queue with FRONT > REAR. That is a second convention, in which FRONT simply moves past REAR after the last deletion instead of both being reset to −1. Both work; an exam question will say which one it uses, and the empty and full tests must come from the same convention.

The simple queue has a serious flaw (source p56). FRONT only moves forward, so the slots freed by dequeues at the start of the array are never used again.

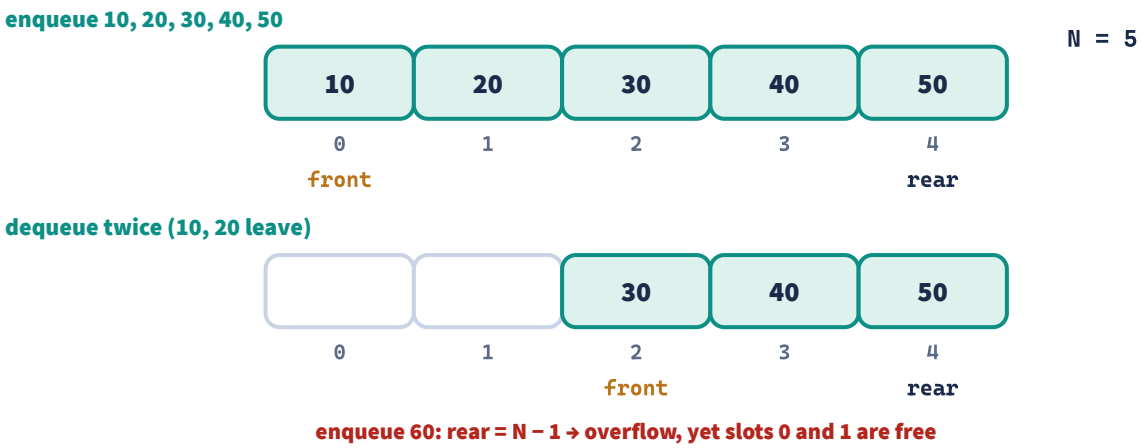


Figure 4.1 False overflow in a simple queue of size 5: after two dequeues the queue holds three elements, but REAR = N − 1, so the next enqueue fails.

This is **false overflow**: the queue reports full while space exists. Shifting all elements to the front after each dequeue fixes it but costs O(n) per dequeue. The circular queue fixes it in O(1).

★ Simple queue: FRONT and REAR only move forward, so freed front slots are wasted.

4.4 Circular Queue

A **circular queue** treats the array as a ring: after the last position, N − 1, comes position 0 again (source p57). Indices advance with modulo arithmetic, so freed slots at the start are reused.

$$\text{next}(i) = (i + 1) \bmod N \quad \cdot \quad \text{previous}(i) = (i - 1 + N) \bmod N$$

Table 4.3 Circular queue rules, with FRONT = REAR = −1 for empty

Item	Rule
empty	FRONT = −1
full	(REAR + 1) mod N = FRONT
enqueue x	if full → overflow; if empty then FRONT ← 0; REAR ← (REAR + 1) mod N; Q[REAR] ← x
dequeue	if empty → underflow; x ← Q[FRONT]; if FRONT = REAR then FRONT ← REAR ← −1 else FRONT ← (FRONT + 1) mod N
number of elements	(REAR − FRONT + N) mod N + 1 (when not empty)

Worked 4.3 A circular queue of size N = 5 that really wraps around

Operation	Q[0] Q[1] Q[2] Q[3] Q[4]	FRONT	REAR	Note
`enqueue 10, 20, 30`	`10 20 30 _ _`	0	2	
`dequeue, dequeue`	`_ _ 30 _ _`	2	2	returns 10, then 20
`enqueue 40, 50`	`_ _ 30 40 50`	2	4	a simple queue is now "full"
`enqueue 60`	`60 _ 30 40 50`	2	0	(4 + 1) mod 5 = 0: wrap-around
`enqueue 70`	`60 70 30 40 50`	2	1	(1 + 1) mod 5 = 2 = FRONT: full
`enqueue 80`	`60 70 30 40 50`	2	1	**overflow** (genuine)
`dequeue`	`60 70 _ 40 50`	3	1	returns 30

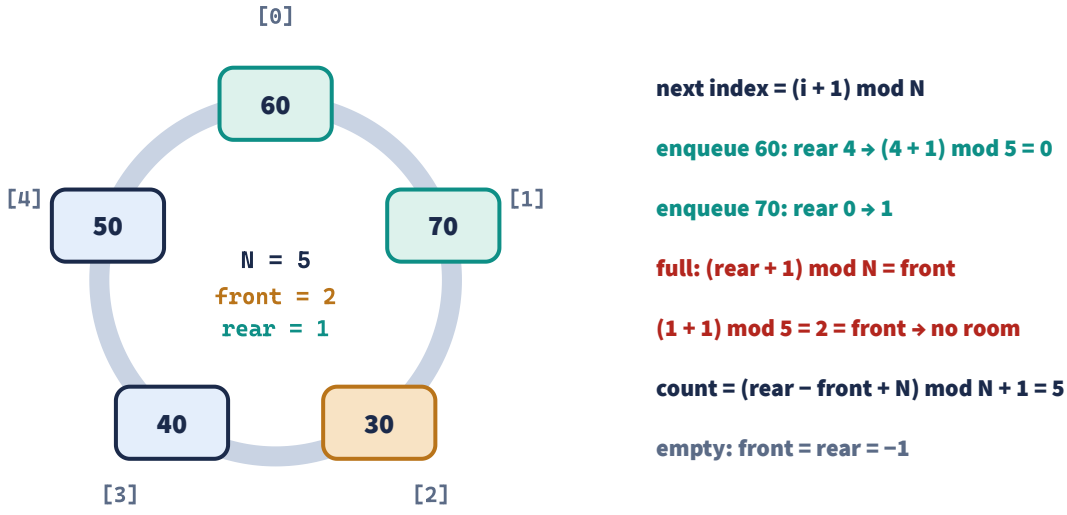


Figure 4.2 The full circular queue after enqueue 70: the queue runs from index 2 round to index 1, front to rear.

WATCH IT! · COMMON MISTAKE 4.1 — THE FULL AND EMPTY TESTS MUST AGREE (SOURCE P57)

Source p57 gives "next(REAR) = FRONT → full" and "FRONT = REAR → empty", and its example starts the empty queue at FRONT = 0, REAR = 4. With those numbers (4 + 1) mod 5 = 0 = FRONT, so its own full test calls the empty queue full. And FRONT = REAR is also true when the queue holds exactly one element. Its example labelled "wrap-around" never wraps either: after enqueue 40, 50 REAR is 4, and the wrap happens only at the next enqueue. Use one consistent convention: Table 4.3 above (empty FRONT = −1, full (REAR + 1) mod N = FRONT), or the textbook alternative that keeps one slot always empty, with empty FRONT = REAR and full (REAR + 1) mod N = FRONT, holding at most N − 1 elements.

★ **Circular queue: indices advance mod N, so freed slots are reused; full when (REAR + 1) mod N = FRONT.**

4.5 Linked Queue

A queue can also be a singly linked list with two pointers: FRONT to the first node and REAR to the last. Enqueue links a new node after REAR and moves REAR to it; dequeue removes the node at FRONT and moves FRONT to the next node. When the last node leaves, both become NULL. Both operations stay O(1), and overflow happens only when memory runs out, so there is no false overflow and no fixed capacity. Linked lists are the subject of Chapter 5.

Table 4.4 Three ways to store a queue

Aspect	Simple array queue	Circular array queue	Linked queue
capacity	N, but freed slots lost	N, all reused	limited only by memory
false overflow	yes	no	no
extra memory	none	none	one pointer per node
enqueue / dequeue	O(1)	O(1)	O(1)

★ **Linked queue: FRONT and REAR pointers, no fixed size, O(1) both ends.**

4.6 Double-Ended Queue (Deque)

A **deque** (double-ended queue, pronounced "deck") allows insertion and deletion at **both** the front and the rear (source p58). It has four basic operations: insertFront, insertRear, deleteFront and deleteRear. A deque generalizes both structures: using only one end makes it a stack, inserting at the rear and deleting at the front makes it a queue. It is built on a circular array or a doubly linked list.

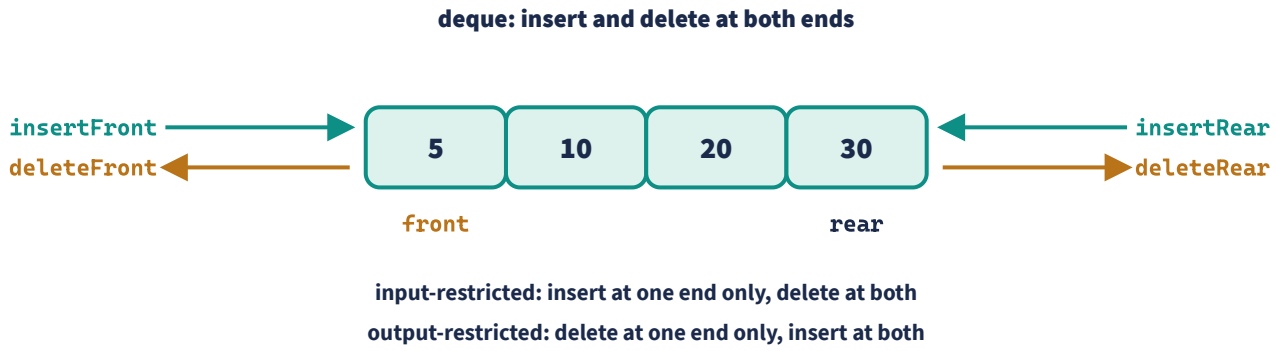


Figure 4.3 A deque with its four operations, and its two restricted variants.

Table 4.5 Deque variants (source p58)

Variant	Insertion	Deletion
input-restricted deque	at one end only (usually the rear)	at both ends
output-restricted deque	at both ends	at one end only (usually the front)

Worked 4.4 Deque operations (source p58, corrected)

Operation	Deque (front → rear)	Returned
`insertRear(10), insertRear(20)`	`10 20`	—
`insertFront(5)`	`5 10 20`	—
`deleteRear()`	`5 10`	20

WATCH IT! · COMMON MISTAKE 4.2 — INSERTFRONT ADDS AN ELEMENT (SOURCE P58)

The source's example shows the deque as 10 20 after insertFront(5) and as 10 after deleteRear(). The 5 has vanished. insertFront(5) gives 5 10 20, and deleteRear() then removes 20 and leaves **5 10**.

Deques are used for sliding-window problems, palindrome checking (compare and remove from both ends), and undo lists that drop their oldest entry.

★ **Deque: both ends open; input-restricted limits insertion, output-restricted limits deletion.**

4.7 Priority Queue: Definition and Need

A **priority queue** is a queue in which every element has an associated **priority**, and elements are served by priority, not only by arrival time (source p59, p60). A higher-priority element is served before a lower-priority one; elements of **equal priority are served in FIFO order**.

An ordinary FIFO queue treats every task as equally important. Real systems cannot: a critical patient, an interrupt from the disk, or a voice packet must not wait behind routine work (source p61, p68). The priority queue serves the most important item first, uses resources better and responds faster, while FIFO among equals keeps it fair.

Table 4.6 Ordinary queue and priority queue (source p61, p68)

Aspect	Ordinary queue	Priority queue
order of service	arrival time (FIFO)	priority; FIFO only among equal priorities
served first	oldest element	highest-priority element
insertion	at the rear	by priority (or anywhere, then searched at deletion)
cost with a heap	O(1) enqueue and dequeue	O(log n) enqueue and dequeue
typical use	print queue, ticket line, buffers	CPU scheduling, emergency handling, Dijkstra's algorithm

Worked 4.5 Mixed tasks served by priority (source p68)

Arrival order	T1 Low · T2 High · T3 Low · T4 High · T5 Medium
FIFO queue serves	`T1 T2 T3 T4 T5`
priority queue serves	`T2 T4 T5 T1 T3 (T2 before T4 and T1 before T3 by arrival)`

★ **Priority queue: priority first, arrival time second.**

4.8 Ascending and Descending Priority Queues

There are two kinds, depending on which end of the priority scale counts as most important (source p62, p69).

Table 4.7 The two types (source p62–64, p69–71)

Aspect	Ascending (min-priority) queue	Descending (max-priority) queue
served first	smallest priority value	largest priority value
meaning	lower number = more important	higher number = more important
removal order of 5, 2, 8, 1, 7	`1 2 5 7 8`	`8 7 5 2 1`
usual implementation	min-heap	max-heap
typical uses	shortest job first, Dijkstra's shortest path, earliest event in a simulation, Huffman coding	highest score or profit, most urgent task, best move in a game

Worked 4.6 Descending priority queue (source p64, p71)

Inserted	A(25) B(70) C(40) D(90) E(60)
dequeue order	`D(90) → B(70) → E(60) → C(40) → A(25)`
as an ascending queue	`A(25) → C(40) → E(60) → B(70) → D(90)`

★ **Ascending = smallest value first (min); descending = largest value first (max).**

4.9 Priority Queue Operations and Implementations

A priority queue supports enqueue(x, p) (insert x with priority p), dequeue() (remove and return the highest- or lowest-priority element, or report underflow if empty), peek(), isEmpty(), and changePriority(x, newP), which updates an element's priority and restores the order (source p65, p72).

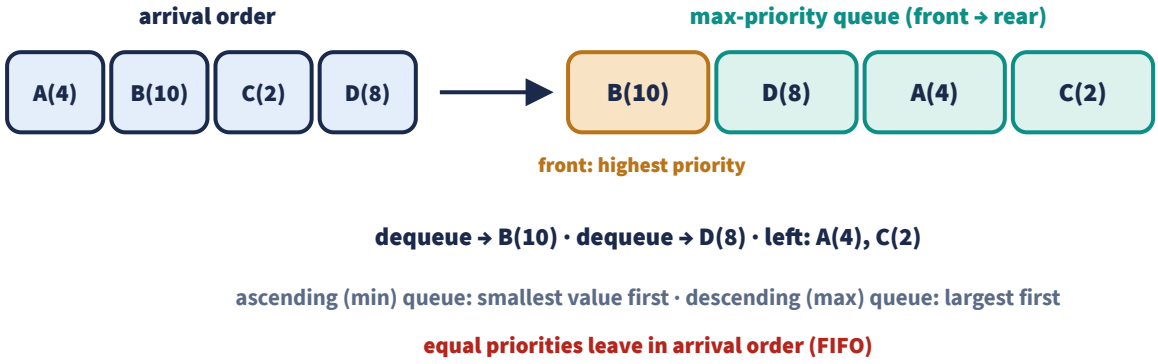


Figure 4.4 A max-priority queue kept in priority order: the front always holds the largest priority.

Worked 4.7 Max-priority queue: enqueue (A,4), (B,10), (C,2), (D,8), then dequeue twice (source p72)

Operation	Queue (front → rear)	Returned
`enqueue (A,4)`	`A(4)`	—
`enqueue (B,10)`	`B(10) A(4)`	—
`enqueue (C,2)`	`B(10) A(4) C(2)`	—
`enqueue (D,8)`	`B(10) D(8) A(4) C(2)`	—
dequeue	`D(8) A(4) C(2)`	B(10)
dequeue	`A(4) C(2)`	D(8)

Table 4.8 How a priority queue can be stored (n elements)

Storage	enqueue	dequeue (highest)	Idea
unsorted array or list	O(1)	O(n)	add at the end; search for the highest at removal
sorted array	O(n)	O(1)	shift to insert in place; highest kept at the end
sorted linked list	O(n)	O(1)	walk to the right place; highest kept at the head
binary heap	**O(log n)**	**O(log n)**	a complete binary tree in an array; peek O(1)

The heap is the standard choice because neither operation is ever slow. Heaps are built in the trees part of this book and used again in heap sort.

★ **Sorted storage makes dequeue cheap, unsorted makes enqueue cheap; a heap makes both O(log n).**

4.10 Applications of Queues and Priority Queues

Table 4.9 Where they are used (source p50, p56, p66, p73)

Structure	Applications
simple / circular queue	CPU ready queue (round-robin), printer spooling, keyboard and I/O buffers, network packet buffering, breadth-first search, customer-service systems
deque	sliding-window maximum, palindrome checking, work-stealing task schedulers
priority queue	priority CPU scheduling and interrupt handling, hospital triage, printing urgent jobs first, QoS packet scheduling, event-driven simulation, Dijkstra's and Prim's algorithms, Huffman coding, keeping the top k elements

Worked 4.8 Priority service examples (source p66, p73)

Situation	Rule	Service order
CPU queue P2(5), P1(1), P4(4), P3(2)	higher number = higher priority	P2 P4 P3 P1
print jobs J3(1), J1(4), J5(2), J2(5)	lower number = higher priority	J3 J5 J1 J2
documents A 5 MB, B 1 MB, C 2 MB, D 8 MB	smaller document first	B C A D



BRAIN POWER

Can a priority queue behave exactly like an ordinary queue, or like a stack? What priority would you give each element as it arrives?

★ **Queues give fair service; priority queues give urgent service.**

Who Does What?

FIFO	•	• insert and delete at both ends
circular queue	•	• served by priority, ties by arrival
deque	•	• first in, first out
priority queue	•	• indices advance mod N

✎ Sharpen your pencil 4

1. A simple array queue has N = 4. Enqueue 1, 2, 3; dequeue twice; enqueue 4; enqueue 5. What happens at the last step, and why? _____
2. A circular queue has N = 6, FRONT = 4, REAR = 1. How many elements does it hold, and at which index does the next enqueue go? _____
3. On an empty deque: insertRear(1), insertRear(2), insertFront(3), deleteRear(), insertFront(4). Write the deque front to rear. _____
4. A max-priority queue receives (P,3), (Q,7), (R,7), (S,1) in that order. Give the dequeue order. _____

EXAM POINT

- Queue = FIFO: enqueue at the rear, dequeue at the front, both O(1).
- Simple array queue wastes freed slots (false overflow); the circular queue fixes it with mod N.
- Circular full: (REAR + 1) mod N = FRONT; count = (REAR − FRONT + N) mod N + 1.
- Deque: both ends; input-restricted and output-restricted variants.
- Priority queue: highest priority first, FIFO among equals; heap gives O(log n).

✓ Solution · Sharpen your pencil 4

1. After 1, 2, 3 REAR = 2; two dequeues give FRONT = 2; enqueue 4 makes REAR = 3 = N − 1. Enqueue 5 reports **overflow**, a **false overflow**: slots 0 and 1 are free but a simple queue cannot reuse them.
2. (1 − 4 + 6) mod 6 + 1 = **4** elements (indices 4, 5, 0, 1). Next enqueue goes to (1 + 1) mod 6 = **index 2**.
3. 1 2 → 3 1 2 → 3 1 → **4 3 1**.
4. **Q, R, P, S**: Q and R share priority 7, and Q arrived first.

Who does what: FIFO → first in, first out; circular queue → indices advance mod N; deque → insert and delete at both ends; priority queue → served by priority, ties by arrival.

BULLET POINTS

- FRONT deletes, REAR inserts.
- Simple queue: false overflow.
- Circular: (i + 1) mod N, reuses slots.
- Deque: four operations, two restricted forms.
- Min-PQ smallest first, max-PQ largest first.
- Brain Power: yes. Give each element its arrival number. A min-priority queue then serves the earliest first, a queue; a max-priority queue serves the latest first, a stack.

Next: Chapter 5 drops the fixed array altogether. A linked list joins nodes with pointers, the structure behind linked stacks and queues, and grows one node at a time.

CHAPTER · UNIT IV · FUNDAMENTALS OF DATA STRUCTURES

5 Linked Lists

A treasure hunt: each clue tells you where the next clue is hidden. Nobody needs the clues to be next to each other; they only need each one to point to the next.

OPENING SITUATION

A music app holds your playlist of 500 songs. You drag a new song into position 3. If the playlist were an array, 497 songs would shift one place. The app instead keeps each song in a small **node** that remembers which song comes next, so placing a song between two others means changing two links and nothing else. That chain of nodes is a **linked list**. This chapter builds the node and the head pointer, compares the list with the array, covers the singly, doubly and circular forms, and traces traversal, insertion, deletion and search link by link.

IN THIS CHAPTER

- 5.1 What a Linked List Is
- 5.2 Why Linked Lists Are Needed
- 5.3 Arrays and Linked Lists Compared
- 5.4 The Node and the Head Pointer
- 5.5 Singly Linked List
- 5.6 Doubly Linked List
- 5.7 Circular Linked List
- 5.8 Traversal
- 5.9 Insertion
- 5.10 Deletion
- 5.11 Searching
- 5.12 Applications of Linked Lists

YOU NEED

- Arrays and the cost of shifting (Chapter 2).
- Linked stacks and queues (Sections 3.6 and 4.5).

5.1 What a Linked List Is

A **linked list** is a linear data structure whose elements are stored as **nodes** connected by **links** (source p74). Each node holds the data and the address of the next node. The nodes are **not** stored in contiguous memory: they lie wherever free memory was found, and the links alone give the order. The first node is reached through the **head**; the last node's link is **NULL**, meaning "no next node".

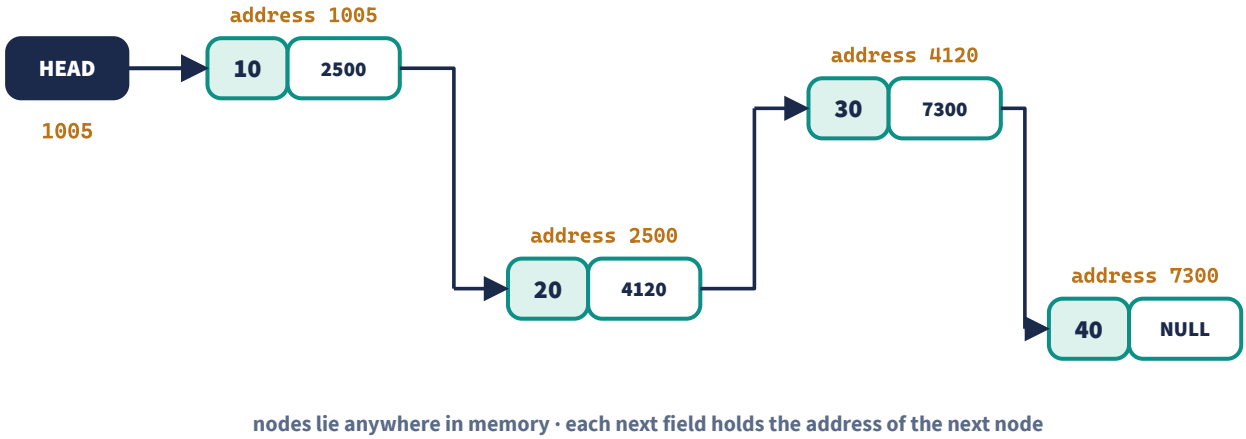


Figure 5.1 Four nodes at scattered addresses (source p74). Read in link order they form the list 10, 20, 30, 40.

Dynamic size

Nodes are created and freed at run time; the list grows and shrinks as needed.

Cheap updates

Insertion and deletion change a few links; nothing is shifted.

Sequential access

To reach the k-th node you must walk k links from the head.

★ **Linked list: nodes anywhere in memory, ordered by links, entered through the head.**

5.2 Why Linked Lists Are Needed

Arrays have a fixed size and make insertion or deletion in the middle costly (source p75). Inserting 25 between 20 and 30 in 10 20 30 40 50 shifts 30, 40 and 50 one place right, O(n). An array must also be sized in advance: too small and it overflows, too large and memory is wasted.

A linked list removes both problems. Memory is allocated one node at a time, so no size is fixed in advance and no single large block is needed. Inserting 25 needs only the links of the neighbouring nodes to change. Linked lists therefore suit data whose size is unknown or changes often: stacks and queues, adjacency lists of graphs, chaining in hash tables, playlists with next and previous, undo/redo, and file-system and scheduling structures (source p75).

> **WATCH THE MODEL**
> "Insertion in a linked list is O(1)" is true only when you already hold a pointer to the node after which you insert. Inserting at position k first needs a walk of k links, which is O(n). What the linked list saves is the shifting, not the searching.

★ **Linked lists trade random access for dynamic size and shift-free updates.**

5.3 Arrays and Linked Lists Compared

Table 5.1 Array and linked list (source p76)

Aspect	Array	Linked list
memory	contiguous block	non-contiguous nodes joined by pointers
size	fixed when created	grows and shrinks at run time
access to the k-th element	random access, O(1)	sequential access, O(n)
insertion / deletion	shifting needed, O(n)	change links, O(1) once the position is known
memory use	may waste unused slots; no extra per element	no unused slots, but one or two pointers per node
cache behaviour	good locality, fast in practice	poor locality, nodes scattered
implementation	simple	needs pointers and dynamic memory management
choose when	size is known and fast access matters	size varies and updates are frequent

★ **Array for speed of access; linked list for flexibility of size and updates.**

5.4 The Node and the Head Pointer

A **node** is the basic unit of a linked list (source p77). In a singly linked list it has two fields: the **data** field, which stores the information, and the **link** or next field, which stores the address of the next node. The last node's link is NULL. A doubly linked list node has three fields: prev, data and next.

The **head pointer** points to the first node and is the only entry into the list (source p78). Traversal always starts from it, and insertion or deletion at the beginning changes it. An empty list has **head = NULL**. Lose the head and every node becomes unreachable, a memory leak.

★ **Node = data + link; head = the door to the list; empty list: head = NULL.**

5.5 Singly Linked List

In a **singly linked list** each node has data and the address of the next node, the last link is NULL, and a single head pointer gives the first node (source p79). It can be traversed **forward only**.

Advantages

Simple; one pointer per node, so the least memory of the linked forms; size changes at run time; easy insertion and deletion; no size needed in advance.

Disadvantages

No backward traversal; no random access; extra memory for addresses; to delete a node you need its predecessor; if the head is lost the list is lost.

Uses: linked stacks, adjacency lists, playlists, symbol tables in compilers, and polynomials, where each node stores a coefficient and an exponent (Section 5.12).

★ **Singly linked: one link per node, forward only, ends at NULL.**

5.6 Doubly Linked List

In a **doubly linked list** each node has three parts: a **prev** link, the **data**, and a **next** link, so the list can be traversed forward and backward (source p80). The first node's prev and the last node's next are NULL; a **tail** pointer is often kept to the last node.

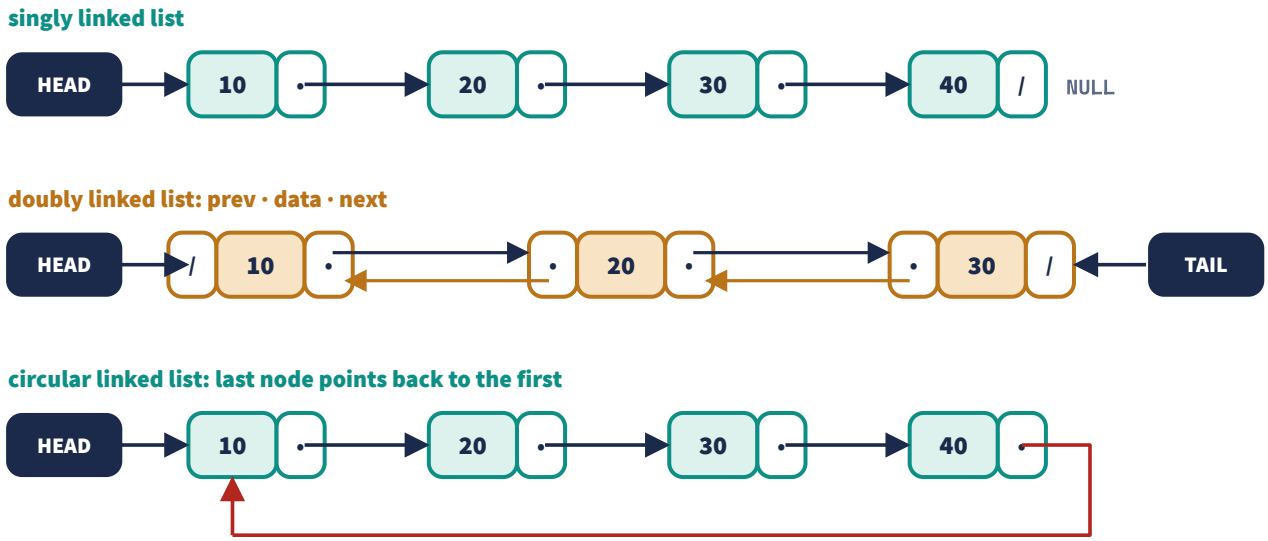


Figure 5.2 The three forms: singly (next only), doubly (prev and next), and circular (the last node links back to the first).

Table 5.2 Singly and doubly linked lists (source p80)

Feature	Singly linked	Doubly linked
links per node	one (next)	two (prev and next)
traversal	forward only	forward and backward
memory	less	more
delete a node you hold a pointer to	needs its predecessor: walk from the head, O(n)	prev is in the node: O(1)
code complexity	simple	moderate: four links to keep right

Uses: browser Back and Forward, undo/redo, image viewers and media players with previous and next, OS task lists, and the **LRU cache**, which moves a used item to the front and evicts from the back in O(1).

★ **Doubly linked: prev and next, both directions, O(1) delete of a known node.**

5.7 Circular Linked List

In a **circular linked list** the last node points back to the **first** node instead of NULL (source p81). There is no end: from any node you can reach every other node. A circular list can be singly or doubly linked; in a circular doubly linked list the first node's prev also points to the last node.

Table 5.3 Linear and circular singly linked lists (source p81)

Aspect	Singly linked	Circular linked
last node's link	NULL	the first node
traversal ends	at NULL	when the walk returns to the start node
sequence	linear	cyclic
risk	—	infinite loop if the stop test is missing

A circular list is often accessed through a pointer to its **last** node: last.next is then the first node, so both ends are reachable in O(1). Uses: round-robin CPU scheduling, multiplayer turns, circular buffers and repeating playlists.

★ **Circular: last joins first; stop when you come back to where you started.**

5.8 Traversal

Traversal visits every node once, in order (source p82).

1. Set a pointer T to head.
2. While T is not NULL: process T.data, then T ← T.next.
3. Stop when T becomes NULL. For a circular list, process the first node and stop when T returns to the head.

Worked 5.1 Traverse 10 → 20 → 30 → 40, counting nodes and summing data

T at	Visit	count	sum
node 1	10	1	10
node 2	20	2	30
node 3	30	3	60
node 4	40	4	100
NULL	stop	**4**	**100**

Every node is visited exactly once: O(n). There is no way to jump to a node directly.

★ **Traversal: start at head, follow next, stop at NULL; O(n).**

5.9 Insertion

Insertion adds a new node at the beginning, the end, or after a given node (source p83). Every case starts the same way: create a node N and store the data in it. Then only the links differ.

Worked 5.2 **The three insertion cases on 10 → 20 → 30 (source p83)**

Case	Link steps, in this order	Result	Time
insert 5 at the beginning	N.next ← head; head ← N	`5 10 20 30`	O(1)
insert 40 at the end	walk T to the last node (T.next = NULL); N.next ← NULL; T.next ← N	`10 20 30 40`	O(n); O(1) with a tail pointer
insert 25 after 20 (node P)	N.next ← P.next; P.next ← N	`10 20 25 30`	O(1) once P is found; finding P is O(n)

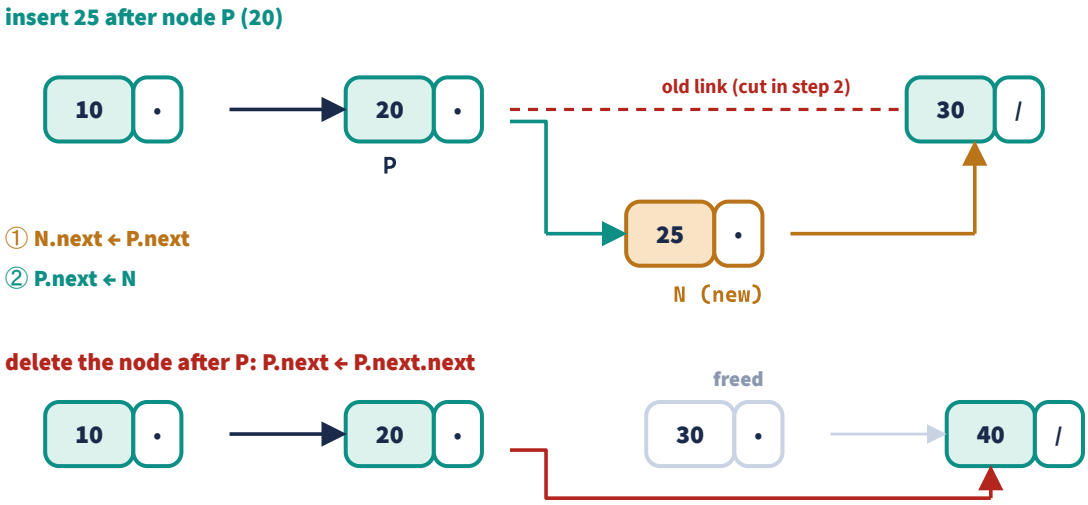


Figure 5.3 Inserting after P takes two link changes, in a fixed order. Deleting the node after P is one link change: P skips over it.

WATCH IT! · COMMON MISTAKE 5.1 — SET THE NEW NODE'S LINK FIRST

Doing P.next ← N first destroys the only pointer to the rest of the list (30 and beyond); the following N.next ← P.next would then point N at itself. Always connect the new node to its successor **before** connecting its predecessor to it. The same rule gives N.next ← head before head ← N.

In a **doubly** linked list, inserting N between P and Q sets four links: N.prev ← P, N.next ← Q, P.next ← N, Q.prev ← N. In a **circular** singly list, inserting at the beginning must also update the last node's link to the new first node.

★ **Insert = create, link the new node forward, then link the old node to it.**

5.10 Deletion

Deletion removes a node and reconnects the list so it stays whole (source p84). First check for an empty list (underflow).

Worked 5.3 **The three deletion cases (source p84)**

Case	Steps	Before → after	Time
first node	T ← head; head ← head.next; free T	`10 20 30 → 20 30`	O(1)
last node	walk P to the second-last node; free P.next; P.next ← NULL	`10 20 30 → 10 20`	O(n)
a middle node (20)	find its predecessor P (10); T ← P.next; P.next ← T.next; free T	`10 20 30 40 → 10 30 40`	O(n), for the search

WATCH IT! · COMMON MISTAKE 5.2 — A TAIL POINTER DOES NOT MAKE "DELETE LAST" O(1)

In a singly linked list the tail pointer reaches the last node, but deletion needs the **second-last** node, whose link must become NULL, and nothing points back to it. Deleting the last node stays O(n). A doubly linked list fixes this: tail.prev is the second-last node.

In a doubly linked list a node X is removed with X.prev.next ← X.next and X.next.prev ← X.prev, after checking whether X is the first or the last node.

★ **Delete = find the predecessor, bypass the node, free it; update head if it was first.**

5.11 Searching

A linked list supports only **sequential search**: start at the head, compare each node's data with the key, and move on until it is found or NULL is reached (source p85).

Worked 5.4 Search the list 10 → 20 → 30 → 40 (source p85)

Key	Comparisons	Result
30	10 ≠ 30, 20 ≠ 30, 30 = 30	found at node 3 (3 comparisons)
50	10, 20, 30, 40 all ≠ 50, then NULL	not found (4 comparisons)

Best case O(1), key at the head; worst case O(n), key last or absent. Binary search does not help even if the list is sorted, because there is no O(1) way to reach the middle node.

★ **Search a linked list from head to NULL; always O(n) in the worst case.**

5.12 Applications of Linked Lists

Table 5.4 Where linked lists are used (source p86)

Use	How the list is used
stacks and queues	push and pop at the head; enqueue at the tail, dequeue at the head
graph adjacency lists	one list of neighbours per vertex
hash tables (chaining)	keys that collide in one bucket form a list
polynomials	one node per term: coefficient, exponent, next
sparse matrices	one node per non-zero element
dynamic memory management	free blocks kept in a free list
playlists, browser history, undo/redo	next and previous navigation (doubly linked)
round-robin scheduling	a circular list of processes

Worked 5.5 **Adding two polynomials stored as lists (terms in decreasing exponent)**

Step	P1 = $5x^3 + 4x^2 + 2$	P2 = $3x^2 + x + 7$	Result so far
exponent 3 vs 2: copy the larger	(5, 3)	—	$5x^3$
exponent 2 = 2: add coefficients	(4, 2)	(3, 2)	$5x^3 + 7x^2$
exponent 0 vs 1: copy the larger	—	(1, 1)	$5x^3 + 7x^2 + x$
exponent 0 = 0: add coefficients	(2, 0)	(7, 0)	$5x^3 + 7x^2 + x + 9$



BRAIN POWER

How can you find the middle node of a singly linked list in a single pass, without first counting the nodes?

★ **Linked lists power stacks, queues, graphs, hashing, polynomials and anything that grows unpredictably.**

Who Does What?

head = NULL	•	• last node links to the first
doubly linked	•	• the list is empty
circular	•	• visit every node from head to NULL
traversal	•	• prev, data and next in each node

✎ Sharpen your pencil 5

1. Start with HEAD → 12 → 7 → 19 → NULL. Insert 5 at the beginning, insert 30 at the end, then delete 7. Write the final list. _____
2. How many link fields are set when a node is inserted between two nodes of a doubly linked list? How many of them belong to existing nodes? _____
3. In 10 → 20 → 30 → 40, how many comparisons does an unsuccessful search for 50 make? _____
4. Store 1000 integers (4 bytes each, pointers 8 bytes, ignore padding) in an array, a singly linked list and a doubly linked list. How many bytes does each use? _____

EXAM POINT

- Node = data + next (singly); prev + data + next (doubly). Empty list: head = NULL.
- Access O(n); insert/delete at the head O(1); at the end O(n) without a tail pointer.
- Insert after P: N.next ← P.next, then P.next ← N.
- Circular list: last node points to the first; traversal stops on returning to the start.

✓ Solution · Sharpen your pencil 5

1. 5 12 7 19 → 5 12 7 19 30 → delete 7: **5 → 12 → 19 → 30 → NULL**.
2. **Four**: N.prev, N.next, P.next, Q.prev. **Two** of them (P.next and Q.prev) are in existing nodes.
3. **4** comparisons, then the pointer reaches NULL and the search reports not found.
4. Array 1000 × 4 = **4000** bytes; singly 1000 × (4 + 8) = **12 000**; doubly 1000 × (4 + 16) = **20 000**.

Who does what: head = NULL → the list is empty; doubly linked → prev, data and next in each node; circular → last node links to the first; traversal → visit every node from head to NULL.

BULLET POINTS

- Nodes anywhere, links give the order.
- No shifting, but no random access.
- Singly: forward only, least memory.
- Doubly: both ways, O(1) delete of a known node.
- Circular: no NULL, careful stop test.
- Brain Power: move a slow pointer one node and a fast pointer two nodes per step; when the fast one reaches NULL, the slow one is at the middle.

Next: Chapter 6 puts the structures of Part I side by side (array and list, stack and queue, linear and non-linear, static and dynamic) and follows them into operating systems, databases, compilers, networks and AI.

CHAPTER · UNIT IV · FUNDAMENTALS OF DATA STRUCTURES

6 Choosing and Applying Data Structures

There is no best data structure, only the best one for the operation you perform most often.

OPENING SITUATION

Open your laptop and five data structures are already at work. The scheduler holds ready processes in a queue. The memory manager keeps free blocks in a linked list. The file explorer shows a tree of folders. Your interrupted program waits on a stack. The browser looks up a website's address in a hash table. None of these choices is an accident; each structure was chosen for the one operation that system performs most. This chapter sets the structures of Part I side by side in four comparisons and a master table, and then follows them into operating systems, databases, compilers, networks and AI.

IN THIS CHAPTER

- 6.1 Array and Linked List
- 6.2 Stack and Queue
- 6.3 Linear and Non-Linear Structures
- 6.4 Static and Dynamic Structures
- 6.5 The Major Structures Side by Side
- 6.6 Data Structures in Operating Systems
- 6.7 Data Structures in Database Systems
- 6.8 Data Structures in Compiler Design
- 6.9 Data Structures in Computer Networks
- 6.10 Data Structures in Artificial Intelligence

YOU NEED

- Chapters 1–5: arrays, stacks, queues and linked lists.
- Postfix evaluation (Section 3.10).

6.1 Array and Linked List

The array keeps elements of one type in contiguous memory and reaches them by index; the linked list keeps nodes anywhere and joins them by links (source p87). Table 5.1 compared them in full. The choice reduces to one question: which operation dominates?

Choose an array when

you mostly **read by position**, the size is known and stable, memory overhead must be low, and cache speed matters.

Choose a linked list when

you mostly **insert and delete**, the size is unknown or changes often, and a large contiguous block is hard to get.

A useful picture from the source: an array is a row of adjacent rooms in one building; a linked list is houses in different streets, each with directions to the next.

★ **Array: fixed, fast access, slow updates. Linked list: dynamic, slow access, cheap updates.**

6.2 Stack and Queue

Table 6.1 Stack and queue (source p88)

Aspect	Stack	Queue
principle	LIFO, last in first out	FIFO, first in first out
insertion	push, at the TOP	enqueue, at the REAR
deletion	pop, from the TOP	dequeue, from the FRONT
accessible element	only the top	only the front
pointers	one: TOP	two: FRONT and REAR
effect on order	reverses it	preserves it
applications	call stack, expression evaluation, undo/redo, backtracking, DFS, parsing	CPU scheduling, printer spooling, I/O buffers, network packets, BFS, ticket queues

★ **Stack reverses order with one pointer; queue keeps order with two.**

6.3 Linear and Non-Linear Structures

A **linear** structure arranges elements in one sequence: each element except the first and last has exactly one predecessor and one successor, and one straight pass visits them all (source p89). A **non-linear** structure arranges them in a hierarchy or a network: an element may connect to many others, and there are many traversal orders.

Table 6.2 Linear and non-linear (source p89)

Aspect	Linear	Non-linear
arrangement	one sequence, a single level	hierarchy (tree) or network (graph)
relationships	one predecessor, one successor	many predecessors and/or successors
traversal	one straight path	many paths and orders (pre-, in-, post-order, BFS, DFS)
implementation	simpler	more complex; often recursive
examples	array, linked list, stack, queue	tree, graph, heap, trie
used for	lists, undo, scheduling	file systems, social networks, routing, database indexes

WATCH THE MODEL

Linear or non-linear describes the **logical** relationships, not the memory layout. A linked list is linear although its nodes are scattered; a binary heap is non-linear (a tree) although it is stored in a plain array.

★ **Linear = one road; non-linear = many roads and junctions.**

6.4 Static and Dynamic Structures

A **static** structure has a fixed size, decided when it is created, usually at compile time, and its memory does not change while the program runs. A **dynamic** structure grows and shrinks at run time, allocating and freeing memory through pointers (source p90).

Table 6.3 Static and dynamic (source p90)

Aspect	Static	Dynamic
memory allocated	once, at creation	at run time, as needed
size	fixed; overflow when exceeded	grows and shrinks
memory use	may waste pre-allocated space	no unused slots, but pointer overhead
access	fast, direct (O(1) for arrays)	slower, through pointers
risks	overflow, waste	memory leaks if nodes are not freed
examples	array, array stack, array queue	linked list, linked stack and queue, tree, graph

★ **Static for speed and simplicity; dynamic for flexibility.**

6.5 The Major Structures Side by Side

Source p91 compares the major structures by type, size, access and memory. Table 6.4 adds the costs that decide most exam questions; trees, heaps, hashing and graphs are built in full in later parts of this book.

Table 6.4 Operation costs (n elements; V vertices, E edges)

Structure	Type	Access	Search	Insert	Delete
array (unsorted)	linear, static	O(1)	O(n)	O(n); O(1) at the end	O(n)
sorted array	linear, static	O(1)	O(log n)	O(n)	O(n)
singly linked list	linear, dynamic	O(n)	O(n)	O(1) at the head	O(1) at the head
stack	linear	top only, O(1)	—	push O(1)	pop O(1)
queue	linear	front only, O(1)	—	enqueue O(1)	dequeue O(1)
binary search tree	non-linear	—	O(log n) average, O(n) worst	same	same
balanced tree (AVL, B-tree)	non-linear	—	O(log n)	O(log n)	O(log n)
binary heap	non-linear	max/min O(1)	O(n)	O(log n)	O(log n)
hash table	—	—	O(1) average, O(n) worst	same	same
graph, adjacency list	non-linear	—	traversal O(V + E)	add edge O(1)	O(degree)

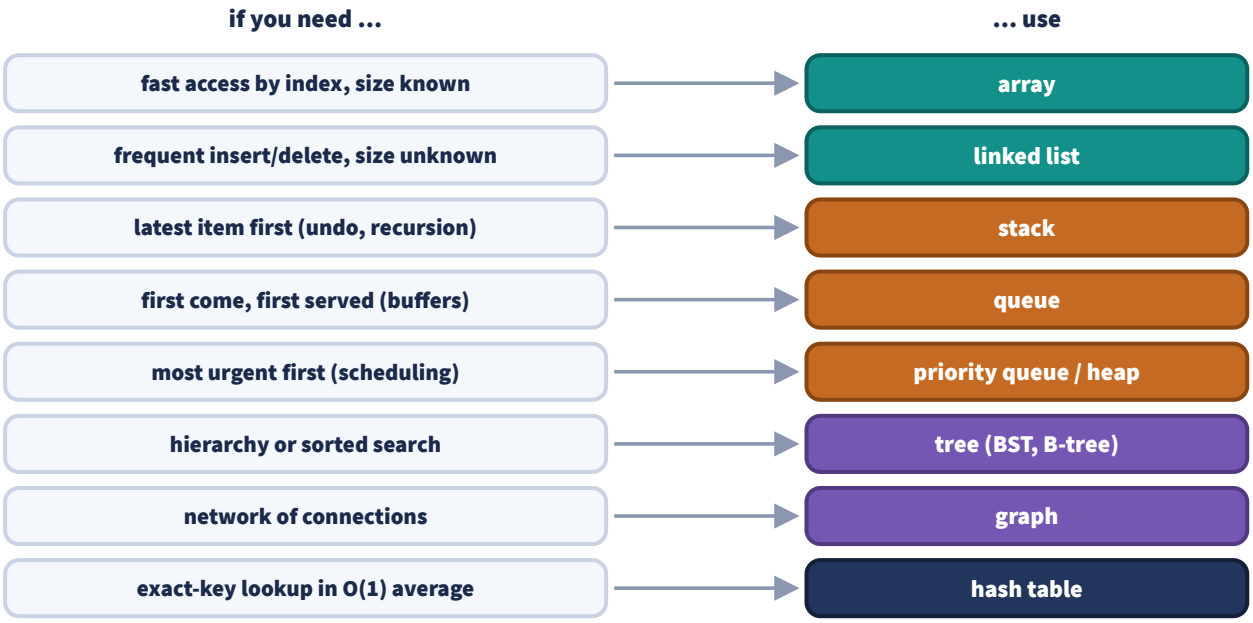


Figure 6.1 Start from the operation you need most, and the structure follows.

★ Pick the structure whose cheap operation is the one you do most.

6.6 Data Structures in Operating Systems

Table 6.5 Every OS task has a structure behind it (source p92)

OS task	Structure	How it is used
process scheduling	queues	ready queue, waiting queue and I/O queue; FIFO, round robin (circular queue) and priority scheduling (priority queue)
memory management	linked list	a free list of variable-sized blocks (e.g. 100 KB → 200 KB → 500 KB → 300 KB → NULL); adjacent free blocks are merged (coalesced) to reduce external fragmentation
file systems	tree	root directory at the top, folders as internal nodes, files as leaves; search, create, delete and traverse
device management	I/O queues	disk and printer requests join at the rear and are served from the front
interrupt handling	stack	CPU state is pushed when an interrupt arrives and popped on return; nested interrupts unwind in LIFO order

★ OS: queues schedule, lists track free memory, trees hold files, stacks handle interrupts.

6.7 Data Structures in Database Systems

Databases store, organize and retrieve data on disk, where every block read is expensive, so their structures aim to reduce I/O (source p93).