Java Data Structures

Complete Course — Zero to Advanced

JAVA 17+ | 16 DATA STRUCTURES | PHASE 1 → 4 | INTERVIEW READY

PHASE	LEVEL	TOPICS
1	**Foundation**	Memory, Big O, Generics, Java Basics
2	**Beginner**	Arrays, Strings, ArrayList, Stack, Queue, Deque
3	**Intermediate**	HashMap, LinkedList, Binary Tree, BST, Heap
4	**Advanced**	AVL Tree, Trie, Graph BFS/DFS, Segment Tree, Union-Find

How to use: Study each phase in order. Every topic covers: Concept → Visual → Complexity → Java Code → Dry Run → Edge Cases → Practice. Never skip phases — each one builds on the previous.

PHASE 1	Foundation — Before You Write a Single Data Structure

1.1 What is a Data Structure?

Definition: A data structure organises data in memory so that operations (access, search, insert, delete) are as fast as possible. Choosing the right one is often the difference between a solution that runs in 1 second vs 10 hours.

- **Array** — numbered shelf — fast O(1) access by index
- **Stack** — pile of plates — LIFO: last added, first removed
- **Queue** — cinema line — FIFO: first added, first removed
- **HashMap** — dictionary — O(1) key-to-value lookup
- **Tree** — org chart — hierarchical relationships
- **Graph** — road map — connections between any nodes

1.2 Memory: Stack vs Heap in Java

Java › Memory Model

```
// ■■ STACK MEMORY ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
// Stores: primitive variables and method call frames
// Managed: automatically freed when method returns
// Size: limited (~512 KB – 8 MB)
int x = 42; // x lives on the Stack
boolean flag = true; // flag lives on the Stack
// ■■ HEAP MEMORY ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
// Stores: all objects
// Managed: Garbage Collector frees unused objects
// Size: large (hundreds of MB or more)
int[] arr = new int[10]; // array object on Heap
String s = new String("hi"); // String object on Heap
// 'arr' and 's' are references — they live on the Stack
// and POINT to objects living on the Heap
// ■■ CRITICAL GOTCHA ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
int[] a = new int[]{1, 2, 3};
int[] b = a; // b and a point to the SAME array!
b[0] = 99; // also changes a[0] — shared memory
```

1.3 Big O Notation

Big O describes how runtime grows as input size *n* increases. Rule: drop constants and keep only the dominant term. **O(2n)** → **O(n)**, **O(n² + n)** → **O(n²)**.

Notation	Name	Example	At n = 1,000
`O(1)`	Constant	arr[0] / map.get(k)	1 step
`O(log n)`	Logarithmic	Binary Search, BST ops	~10 steps

O(n)	Linear	Single for-loop	1,000 steps
O(n log n)	Log-linear	Merge Sort, Heap Sort	~10,000 steps
O(n²)	Quadratic	Nested loops	1,000,000 steps
O(2■)	Exponential	Recursive subsets	Astronomical

Java › Big O Examples in Java

```
// O(1) — same time regardless of input size
int first = arr[0];
// O(n) — time grows linearly with n
for (int i = 0; i < n; i++) {
process(arr[i]);
}
// O(n²) — avoid for large n (n=10,000 = 100 million ops)
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
compare(arr[i], arr[j]);
}
}
// O(log n) — input halves each step
int lo = 0, hi = n - 1;
while (lo <= hi) {
    int mid = lo + (hi - lo) / 2; // avoids integer overflow
    if (arr[mid] == target) return mid;
    else if (arr[mid] < target) lo = mid + 1;
    else hi = mid - 1;
}
```

1.4 Java Generics & Comparable

Java › Generics — Write Once, Work for Any Type

```
// Without generics — you'd write separate IntBox, StringBox, etc.
class Box {
    private T value;
    public void set(T v) { this.value = v; }
    public T get() { return value; }
}
// Usage
Box intBox = new Box<>();
intBox.set(42);
int val = intBox.get(); // no cast needed
Box strBox = new Box<>();
strBox.set("hello");
// Comparable — allows Collections.sort() to work on your class
class Student implements Comparable {
    String name;
    int gpa;
@Override
    public int compareTo(Student other) {
        return Integer.compare(this.gpa, other.gpa); // ascending
}
}
List students = new ArrayList<>();
Collections.sort(students); // works because of Comparable
```

PHASE 2	Beginner Data Structures

2.1 Arrays

Analogy: A row of numbered mailboxes. Jump to any box instantly (O(1)) using its index number. But inserting a new box in the middle requires shifting all boxes after it — O(n).

Java › Arrays — 1D, 2D, and Dynamic Resizing

```
// ■■ 1D Array — fixed size, indices 0 to length-1 ■■■■■■■■■■■
int[] scores = {90, 85, 70, 95, 60};
int first = scores[0]; // 90 — O(1) access
int last = scores[scores.length - 1]; // 60
scores[2] = 75; // update — O(1)
// Iterate
for (int i = 0; i < scores.length; i++) {
System.out.println("Index " + i + " = " + scores[i]);
}
// ■■ 2D Array (matrix) ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
int[][] matrix = new int[3][4]; // 3 rows, 4 columns
matrix[1][2] = 99; // row 1, col 2
for (int row = 0; row < matrix.length; row++) {
    for (int col = 0; col < matrix[row].length; col++) {
System.out.print(matrix[row][col] + " ");
}
System.out.println();
}
// ■■ Dynamic Resizing (what ArrayList does internally) ■■■■■■■■
int[] arr = {1, 2, 3};
int[] bigger = new int[arr.length * 2]; // double the capacity
System.arraycopy(arr, 0, bigger, 0, arr.length);
arr = bigger; // arr now references the larger array
// ■■ Java ArrayList — preferred for dynamic sizing ■■■■■■■■■■■■
ArrayList list = new ArrayList<>();
list.add(10); // append — O(1) amortised
list.add(0, 5); // insert at index 0 — O(n)
list.get(1); // access — O(1)
list.remove(0); // remove by index — O(n)
list.size(); // number of elements
```

Operation	Array	ArrayList
Access by index	O(1)	O(1)
Search	O(n)	O(n)
Insert at end	O(1)	O(1) amortised
Insert at middle	O(n)	O(n)
Delete	O(n)	O(n)

- **Edge case:** arr[-1] or arr[arr.length] → ArrayIndexOutOfBoundsException
- **Edge case:** if arr is null → arr.length throws NullPointerException
- **Use Array** when size is known and fixed (slightly faster, less memory)
- **Use ArrayList** when size changes at runtime

2.2 Stack

> **Analogy:** A stack of plates. You can only add (push) or remove (pop) from the TOP. Last plate placed = first plate taken — **LIFO: Last In, First Out**.

Java › Stack — Custom Implementation Using Array

```
class Stack {
    private Object[] data;
    private int top = -1; // -1 means empty
    private int capacity;
    public Stack(int capacity) {
this.capacity = capacity;
this.data = new Object[capacity];
}
    // Add element to top — O(1)
    public void push(T item) {
        if (top == capacity - 1) {
            throw new RuntimeException("Stack overflow");
}
data[++top] = item; // increment top FIRST, then assign
}
    // Remove and return top element — O(1)
@SuppressWarnings("unchecked")
    public T pop() {
        if (isEmpty()) {
            throw new RuntimeException("Stack underflow");
}
        return (T) data[top--]; // return top, then decrement
}
    // Look at top without removing — O(1)
@SuppressWarnings("unchecked")
    public T peek() {
        if (isEmpty()) throw new RuntimeException("Stack is empty");
        return (T) data[top];
}
    public boolean isEmpty() { return top == -1; }
    public int size() { return top + 1; }
}
```

Java › Stack — Java Built-in (Recommended)

```
// Always use Deque instead of the legacy Stack class
Deque stack = new ArrayDeque<>();
stack.push(10); // top → 10
stack.push(20); // top → 20
stack.push(30); // top → 30
stack.peek(); // 30 — O(1), no removal
stack.pop(); // 30 — O(1), removes top
stack.pop(); // 20
stack.isEmpty(); // false (10 still there)
```

Dry Run — push(10), push(20), push(30), pop()

Java › Dry Run — Stack Operations

```
// Initial state: data=[ _, _, _ ] top = -1
push(10) → data=[10, _, _] top = 0
push(20) → data=[10, 20, _] top = 1
push(30) → data=[10, 20, 30] top = 2
pop() → returns 30 top = 1 data=[10, 20, _]
peek() → returns 20 top = 1 (no change)
```

Real uses: *Undo/Redo, browser back button, expression evaluation, DFS, call stack.*

2.3 Queue & Deque

Queue: A cinema ticket line — **FIFO: First In, First Out**. **Deque** (double-ended queue) allows adding or removing from BOTH ends. It acts as both a Stack and a Queue.

Java › Queue & Deque

```
// ■■ Queue (FIFO) ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
Queue queue = new LinkedList<>();
queue.offer("Alice"); // enqueue — O(1)
queue.offer("Bob");
queue.offer("Charlie");
queue.poll(); // "Alice" — dequeue O(1) — returns null if empty
queue.peek(); // "Bob" — view front, no removal
queue.size(); // 2
// ■■ Deque — use as Stack OR Queue ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
Deque dq = new ArrayDeque<>();
dq.addFirst(1); // front: [1]
dq.addLast(2); // front: [1, 2]
dq.addLast(3); // front: [1, 2, 3]
dq.pollFirst(); // removes 1 → [2, 3]
dq.pollLast(); // removes 3 → [2]
dq.peekFirst(); // 2, no removal
// ■■ Prefer offer/poll over add/remove ■■■■■■■■■■■■■■■■■■■■■■■■■■
// offer() returns false on failure (capacity exceeded)
// add() throws an exception on failure — avoid in most cases
```

Real uses: *BFS traversal, task scheduling, sliding window problems, print queue.*

3.1 HashMap & HashSet

Analogy: A dictionary. You look up a word (key) and instantly get its definition (value). Internally a **hash function** maps each key to an array bucket index. Collisions (two keys → same index) are resolved by chaining (LinkedList at each bucket).

Java › HashMap — Key-Value Lookup in O(1)

```
// ■■ Basic Operations ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
HashMap scores = new HashMap<>();
scores.put("Alice", 95);
scores.put("Bob", 82);
scores.put("Carol", 91);
scores.get("Bob"); // 82 — O(1)
scores.getOrDefault("Dave", 0); // 0 — safe fallback
scores.containsKey("Alice"); // true — O(1)
scores.remove("Carol"); // removes entry
scores.size(); // 2
// ■■ Iterate all entries ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
for (Map.Entry entry : scores.entrySet()) {
System.out.println(entry.getKey() + " → " + entry.getValue());
}
// ■■ Count character frequencies (classic pattern) ■■■■■■■■■■■■
String text = "hello world";
Map freq = new HashMap<>();
for (char c : text.toCharArray()) {
freq.put(c, freq.getOrDefault(c, 0) + 1);
}
// freq: {h=1, e=1, l=3, o=2, ' '=1, w=1, r=1, d=1}
```

Java › HashSet — Unique Elements, O(1) Membership

```
// HashSet is a HashMap where we only care about keys, not values
HashSet visited = new HashSet<>();
visited.add(1);
visited.add(2);
visited.add(1); // duplicate — ignored silently
visited.contains(2); // true — O(1)
visited.contains(5); // false — O(1)
visited.remove(1); // removes 1
visited.size(); // 1
// ■■ Find duplicates in an array ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
int[] arr = {1, 3, 5, 3, 7, 1};
Set seen = new HashSet<>();
for (int x : arr) {
    if (!seen.add(x)) { // add() returns false for duplicates
System.out.println("Duplicate: " + x);
}
}
```

Operation	Average	Worst (all collide)
put	O(1)	O(n)
get	O(1)	O(n)
remove	O(1)	O(n)
containsKey	O(1)	O(n)

- **HashMap** — allows one null key, unordered, O(1) average
- **TreeMap** — sorted by key, O(log n) for all ops, no null keys
- **LinkedHashMap** — insertion-order preserved, great for LRU Cache

---

3.2 LinkedList — Built from Scratch

Analogy: A treasure hunt. Each clue (Node) contains data and tells you where the next clue is. Unlike arrays, there is no random access — you must walk node by node from the head.

Java › Node Class (Building Block)

```
class Node {
T data;
Node next;
Node(T data) {
this.data = data;
this.next = null;
}
}
```

Java › Singly LinkedList — addFirst, addLast, delete

```
class SinglyLinkedList {
Node head;
    int size;
    // Add to front — O(1)
    public void addFirst(T data) {
Node newNode = new Node<>(data);
newNode.next = head;
head = newNode;
size++;
}
    // Add to end — O(n)
    public void addLast(T data) {
Node newNode = new Node<>(data);
        if (head == null) { head = newNode; size++; return; }
Node current = head;
        while (current.next != null) current = current.next;
current.next = newNode;
size++;
}
    // Delete by value — O(n)
    public boolean delete(T data) {
        if (head == null) return false;
        if (head.data.equals(data)) { head = head.next; size--; return true; }
Node current = head;
        while (current.next != null) {
            if (current.next.data.equals(data)) {
current.next = current.next.next;
size--; return true;
}
current = current.next;
}
        return false;
}
}
```

Java › Singly LinkedList — reverse() in-place

```
// Reverse in-place — O(n) time, O(1) space
public void reverse() {
Node prev = null;
Node current = head;
Node next = null;
    while (current != null) {
next = current.next; // 1. save next node
current.next = prev; // 2. reverse the pointer
prev = current; // 3. advance prev
current = next; // 4. advance current
}
head = prev; // prev is now the new head
}
```

Dry Run — Reverse: head → [1] → [2] → [3] → null

Java › Dry Run — Reverse LinkedList

```
// Before: head → [1] → [2] → [3] → null
// prev=null, current=[1], next=?
// Step 1: next=[2] [1].next=null prev=[1] current=[2]
// Step 2: next=[3] [2].next=[1] prev=[2] current=[3]
// Step 3: next=null [3].next=[2] prev=[3] current=null
// After: head = prev = [3]
// Result: head → [3] → [2] → [1] → null
```

3.3 Binary Tree & Traversals

> **Analogy:** A company org chart. CEO is the root. Every node has at most 2 children (left and right). Leaf nodes have no children. Every node has exactly one parent (except root).

Java › TreeNode Class

```
// Tree structure:
// 1 <- root
// / \
// 2 3
// / \
// 4 5 <- leaves
class TreeNode {
    int val;
TreeNode left;
TreeNode right;
TreeNode(int val) { this.val = val; }
}
```

Java › Binary Tree — Inorder, Preorder, Postorder

```
// INORDER: Left -> Root -> Right => 4 2 5 1 3
void inorder(TreeNode node) {
    if (node == null) return;
inorder(node.left);
System.out.print(node.val + " ");
inorder(node.right);
}
// PREORDER: Root -> Left -> Right => 1 2 4 5 3
void preorder(TreeNode node) {
    if (node == null) return;
System.out.print(node.val + " ");
preorder(node.left);
preorder(node.right);
}
// POSTORDER: Left -> Right -> Root => 4 5 2 3 1
void postorder(TreeNode node) {
    if (node == null) return;
postorder(node.left);
postorder(node.right);
System.out.print(node.val + " ");
}
```

Java › Binary Tree — Level Order (BFS) + Height

```
// LEVEL ORDER — row by row using Queue => 1 2 3 4 5
void levelOrder(TreeNode root) {
    if (root == null) return;
Queue q = new LinkedList<>();
q.offer(root);
    while (!q.isEmpty()) {
TreeNode node = q.poll();
System.out.print(node.val + " ");
        if (node.left != null) q.offer(node.left);
        if (node.right != null) q.offer(node.right);
}
}
// HEIGHT — O(n), uses recursion
int height(TreeNode node) {
    if (node == null) return 0;
    return 1 + Math.max(height(node.left), height(node.right));
}
```

3.4 Binary Search Tree (BST)

BST Rule: For every node: all values in the **left subtree** are smaller, all in the **right subtree** are larger. This halves the search space at each step — O(log n) for balanced trees.

Java › BST — Insert and Search

```
// BST rule: left < node < right
// 8
// / \
// 3 10
// / \ \
// 1 6 14
class BST {
TreeNode root;
    // Insert — O(log n) avg, O(n) worst
TreeNode insert(TreeNode node, int val) {
        if (node == null) return new TreeNode(val);
        if (val < node.val) node.left = insert(node.left, val);
        else if (val > node.val) node.right = insert(node.right, val);
        return node;
}
    // Search — O(log n) avg
    boolean search(TreeNode node, int val) {
        if (node == null) return false;
        if (val == node.val) return true;
        return (val < node.val)
? search(node.left, val)
: search(node.right, val);
}
TreeNode findMin(TreeNode node) {
        while (node.left != null) node = node.left;
        return node;
}
}
```

Java › BST — Delete (3 cases)

```
TreeNode delete(TreeNode node, int val) {
    if (node == null) return null;
    if (val < node.val) node.left = delete(node.left, val);
    else if (val > node.val) node.right = delete(node.right, val);
    else {
        // Case 1: leaf — just remove
        if (node.left == null && node.right == null) return null;
        // Case 2: one child — return the surviving child
        if (node.left == null) return node.right;
        if (node.right == null) return node.left;
        // Case 3: two children
        // Replace value with inorder successor, then delete it
TreeNode successor = findMin(node.right);
node.val = successor.val;
node.right = delete(node.right, successor.val);
}
    return node;
}
```

Operation	Balanced BST	Skewed BST (worst case)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Worst case occurs when inserting already-sorted data e.g. 1,2,3,4,5. Use AVL Tree to prevent this.

3.5 Heap & Priority Queue

> **Analogy:** An ER triage system. Patients are treated by severity, not by arrival time. A heap always keeps the most critical element (min or max) at the top. Internally stored as an array — no pointers needed.

Java › Min-Heap and Max-Heap using PriorityQueue

```
// ■■ Min-Heap (default in Java) ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
// Property: parent <= both children
// Root is always the MINIMUM element
//
// Array layout: index i → parent = (i-1)/2
// left = 2*i + 1
// right = 2*i + 2
PriorityQueue minHeap = new PriorityQueue<>();
minHeap.offer(5);
minHeap.offer(1);
minHeap.offer(3);
minHeap.peek(); // 1 — view min, no removal — O(1)
minHeap.poll(); // 1 — remove min — O(log n)
minHeap.poll(); // 3
minHeap.poll(); // 5
// ■■ Max-Heap ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
// Root is always the MAXIMUM element
PriorityQueue maxHeap =
    new PriorityQueue<>(Collections.reverseOrder());
maxHeap.offer(5);
maxHeap.offer(1);
maxHeap.offer(3);
maxHeap.poll(); // 5 — always removes maximum
// ■■ Heap with custom objects ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
// Sort tasks by priority (lower number = higher priority)
PriorityQueue taskQueue =
    new PriorityQueue<>((a, b) -> a[1] - b[1]);
taskQueue.offer(new int[]{1, 5}); // {taskId, priority}
taskQueue.offer(new int[]{2, 2});
taskQueue.offer(new int[]{3, 8});
// Processes: task 2 (priority 2), task 1 (priority 5), task 3 (priority 8)
while (!taskQueue.isEmpty()) {
int[] task = taskQueue.poll();
System.out.println("Task " + task[0] + " (priority " + task[1] + ")");
}
```

Operation	Time	Notes
`offer`	O(log n)	Insert + sift up
`poll`	O(log n)	Remove root + sift down
`peek`	O(1)	Just read root
`heapify (all)`	O(n)	Build heap from existing array

4.1 AVL Tree — Self-Balancing BST

Problem with plain BST: Inserting sorted data [1,2,3,4,5] degrades a BST into a linked list — all operations become O(n). An **AVL Tree** self-balances after every insert/delete using rotations, guaranteeing O(log n) for all operations in all cases.

Java › AVL Node Class

```
// Balance Factor (BF) = height(left) - height(right)
// AVL rule: BF must be -1, 0, or +1 at EVERY node
// If |BF| > 1 => rotate to restore balance
class AVLNode {
    int val;
    int height;
AVLNode left;
AVLNode right;
AVLNode(int val) { this.val = val; this.height = 1; }
}
class AVLTree {
    private int height(AVLNode n) { return (n == null) ? 0 : n.height; }
    private int balanceFactor(AVLNode n) {
        return (n == null) ? 0 : height(n.left) - height(n.right);
}
    private void updateHeight(AVLNode n) {
n.height = 1 + Math.max(height(n.left), height(n.right));
}
}
```

Java › AVL Tree — Right and Left Rotations

```
// Right Rotation — fixes Left-Left imbalance
// Before: z After: y
// / \ / \
// y T4 x z
// / \ / \
// x T3 T3 T4
private AVLNode rotateRight(AVLNode z) {
AVLNode y = z.left;
AVLNode T3 = y.right;
y.right = z; // rotation
z.left = T3;
updateHeight(z);
updateHeight(y);
    return y; // y is new subtree root
}
// Left Rotation — fixes Right-Right imbalance
private AVLNode rotateLeft(AVLNode z) {
AVLNode y = z.right;
AVLNode T2 = y.left;
y.left = z;
z.right = T2;
updateHeight(z);
updateHeight(y);
    return y;
}
```

Java › AVL Tree — Insert with Auto-Rebalancing

```
AVLNode insert(AVLNode node, int val) {
    // Step 1: Standard BST insert
    if (node == null) return new AVLNode(val);
    if (val < node.val) node.left = insert(node.left, val);
    else if (val > node.val) node.right = insert(node.right, val);
    else return node; // no duplicates
    // Step 2: Update height
updateHeight(node);
    // Step 3: Check balance and fix
    int bf = balanceFactor(node);
    // LL — single right rotation
    if (bf > 1 && val < node.left.val) return rotateRight(node);
    // RR — single left rotation
    if (bf < -1 && val > node.right.val) return rotateLeft(node);
    // LR — left-right double rotation
    if (bf > 1 && val > node.left.val) {
node.left = rotateLeft(node.left);
        return rotateRight(node);
}
    // RL — right-left double rotation
    if (bf < -1 && val < node.right.val) {
node.right = rotateRight(node.right);
        return rotateLeft(node);
}
    return node;
}
```

Operation	Plain BST (worst)	AVL Tree (always)
Search	O(n)	O(log n)
Insert	O(n)	O(log n)
Delete	O(n)	O(log n)

4.2 Trie (Prefix Tree)

Analogy: Phone contact autocomplete. Each node represents one character. Words sharing a prefix share the same root-to-node path. Search time depends on word length (L), not total number of words — O(L).

Java › TrieNode + Insert

```
// Visual: app, apple, apply
// root->[a]->[p]->[p*]->[l]->[e*]
// ->[y*] (* = word end)
class TrieNode {
TrieNode[] children = new TrieNode[26];
    boolean isEnd = false;
}
class Trie {
TrieNode root = new TrieNode();
    // Insert word — O(L)
    public void insert(String word) {
TrieNode current = root;
        for (char ch : word.toCharArray()) {
            int index = ch - 'a';
            if (current.children[index] == null)
current.children[index] = new TrieNode();
current = current.children[index];
}
current.isEnd = true;
}
}
```

Java › Trie — Search and StartsWith

```
// Search exact word — O(L)
public boolean search(String word) {
TrieNode current = root;
    for (char ch : word.toCharArray()) {
        int index = ch - 'a';
        if (current.children[index] == null) return false;
current = current.children[index];
}
    return current.isEnd; // must be a complete word
}
// Check prefix exists — O(L)
public boolean startsWith(String prefix) {
TrieNode current = root;
    for (char ch : prefix.toCharArray()) {
        int index = ch - 'a';
        if (current.children[index] == null) return false;
current = current.children[index];
}
    return true; // path exists — may or may not be a full word
}
// Usage
Trie trie = new Trie();
trie.insert("apple"); trie.insert("app"); trie.insert("apply");
trie.search("app"); // true
trie.search("ap"); // false — not inserted
trie.startsWith("app"); // true
trie.startsWith("xyz"); // false
```

4.3 Graph — BFS & DFS

> **Analogy:** Google Maps. Cities are **vertices**. Roads are **edges**. **BFS** (Queue) finds the shortest path measured in hops. **DFS** (Stack/recursion) explores as deep as possible before backtracking.

Java › Graph — Adjacency List + BFS

```
// Build adjacency list (space-efficient)
int V = 6;
List> graph = new ArrayList<>();
for (int i = 0; i < V; i++) graph.add(new ArrayList<>());
void addEdge(int u, int v) {
graph.get(u).add(v);
graph.get(v).add(u); // undirected
}
// BFS — level by level using Queue — O(V + E)
void bfs(int start) {
boolean[] visited = new boolean[V];
Queue queue = new LinkedList<>();
queue.offer(start);
visited[start] = true;
    while (!queue.isEmpty()) {
        int node = queue.poll();
System.out.print(node + " ");
        for (int nb : graph.get(node)) {
            if (!visited[nb]) {
visited[nb] = true;
queue.offer(nb);
}
}
}
}
```

Java › Graph — DFS + Cycle Detection

```
// DFS — go deep first using recursion — O(V + E)
void dfs(int node, boolean[] visited) {
visited[node] = true;
System.out.print(node + " ");
    for (int nb : graph.get(node)) {
        if (!visited[nb]) dfs(nb, visited);
}
}
// Cycle Detection in undirected graph via DFS
// A back edge (to non-parent visited node) = cycle
boolean hasCycle(int node, int parent, boolean[] visited) {
visited[node] = true;
    for (int nb : graph.get(node)) {
        if (!visited[nb]) {
            if (hasCycle(nb, node, visited)) return true;
} else if (nb != parent) {
            return true; // back edge found
}
}
    return false;
}
```

Algorithm	Time	Space	Best For
`BFS`	O(V + E)	O(V)	Shortest path (unweighted)
`DFS`	O(V + E)	O(V)	Cycle detect, topo sort
`Dijkstra`	O((V+E) log V)	O(V)	Shortest path (weighted, non-negative)
`Adjacency List`	—	O(V+E)	Sparse graphs (preferred)
`Adjacency Matrix`	—	O(V²)	Dense graphs, O(1) edge check

4.4 Union-Find / Disjoint Set

Analogy: Friend circles in a social network. "Are Alice and Bob in the same circle?" Union-Find answers connectivity queries in nearly O(1) using two key optimisations: **path compression** (flatten tree during find) and **union by rank** (attach smaller tree under taller one).

Java › Union-Find with Path Compression + Union by Rank

```
class UnionFind {
int[] parent;
int[] rank;
UnionFind(int n) {
parent = new int[n];
rank = new int[n];
        // Each element starts as its own root
        for (int i = 0; i < n; i++) {
parent[i] = i;
}
}
    // Find root — with PATH COMPRESSION
    // On the way back up, point every node directly to root
    int find(int x) {
        if (parent[x] != x) {
parent[x] = find(parent[x]); // flatten the tree
}
        return parent[x];
}
    // Union — by RANK (smaller tree goes under taller one)
    boolean union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX == rootY) return false; // already in same set
        if (rank[rootX] < rank[rootY]) parent[rootX] = rootY;
        else if (rank[rootX] > rank[rootY]) parent[rootY] = rootX;
        else {
parent[rootY] = rootX;
rank[rootX]++; // only increment on tie
}
        return true;
}
    boolean connected(int x, int y) {
        return find(x) == find(y);
}
}
// ■■ Usage: count connected components ■■■■■■■■■■■■■■■■■■■■■■■■
UnionFind uf = new UnionFind(6); // nodes 0..5
uf.union(0, 1); // component A: {0, 1}
uf.union(1, 2); // component A: {0, 1, 2}
uf.union(3, 4); // component B: {3, 4}
// node 5 is isolated
uf.connected(0, 2); // true — same component
uf.connected(0, 3); // false — different components
uf.connected(2, 5); // false — 5 is isolated
```

Operation	Time Complexity
find	O(α(n)) ≈ O(1) — inverse Ackermann, practically constant
union	O(α(n)) ≈ O(1)

`connected`	O(α(n)) ≈ O(1)

Practice Problems

[EASY] **Array** — Find the maximum element in an unsorted array.

Java › Solution — Array

```
int max = arr[0];
for (int x : arr) {
max = Math.max(max, x);
}
return max; // O(n) time, O(1) space
```

[EASY] **Stack** — Check if a string's brackets are balanced: '({[]})' → true, '({[})' → false.

Java › Solution — Stack

```
Deque stack = new ArrayDeque<>();
for (char c : s.toCharArray()) {
    if (c == '(' || c == '{' || c == '[') {
stack.push(c);
} else {
        if (stack.isEmpty()) return false;
        char top = stack.pop();
        if (c == ')' && top != '(') return false;
        if (c == '}' && top != '{') return false;
        if (c == ']' && top != '[') return false;
}
}
return stack.isEmpty(); // true only if all brackets matched
```

[EASY] **HashMap** — Count the frequency of each character in a string.

Java › Solution — HashMap

```
Map freq = new HashMap<>();
for (char c : s.toCharArray()) {
freq.put(c, freq.getOrDefault(c, 0) + 1);
}
return freq; // O(n) time and space
```

[MEDIUM] **LinkedList** — Reverse a singly linked list in-place in O(n) time, O(1) space.

Try solving this yourself first. Think: which data structure gives the right time complexity? Write the brute force first, then optimise.

[MEDIUM] **Binary Tree** — Find the maximum depth of a binary tree.

Try solving this yourself first. Think: which data structure gives the right time complexity? Write the brute force first, then optimise.

[MEDIUM] **HashMap** — Two Sum: given array and target, return indices of two numbers that add up to target.

Try solving this yourself first. Think: which data structure gives the right time complexity? Write the brute force first, then optimise.

[HARD] **Graph** — Detect a cycle in an undirected graph using DFS.

Try solving this yourself first. Think: which data structure gives the right time complexity? Write the brute force first, then optimise.

[HARD] **Heap** — Find the K-th largest element in an unsorted array. Target: O(n log k).

Try solving this yourself first. Think: which data structure gives the right time complexity? Write the brute force first, then optimise.

[HARD] **Trie** — Implement autocomplete: given a prefix, return all words in the Trie starting with it.

Try solving this yourself first. Think: which data structure gives the right time complexity? Write the brute force first, then optimise.

Quick Reference Cheat Sheet

When to Use Which Data Structure

Data Structure	Best For	Java Class
`Array`	Fixed-size, index access	int[], String[]
`ArrayList`	Dynamic array, random access	ArrayList
`LinkedList`	Frequent head insert / delete	LinkedList
`Stack (LIFO)`	Undo, DFS, expression eval	Deque (ArrayDeque)
`Queue (FIFO)`	BFS, scheduling, print queue	Queue (LinkedList)
`PriorityQueue`	Min/Max access, greedy problems	PriorityQueue
`HashMap`	O(1) key-value lookup	HashMap
`TreeMap`	Sorted keys, range queries	TreeMap
`HashSet`	Unique elements, fast membership	HashSet
`BST`	Sorted data, floor/ceiling queries	TreeSet
`Trie`	Prefix search, autocomplete	Custom
`Graph`	Connections, shortest paths	Custom (adj list)
`Union-Find`	Connectivity, Kruskal's MST	Custom

Big O Complexity at a Glance

Notation	Name	Example Operation	Speed
`O(1)`	Constant	arr[i], map.get(k), heap.peek()	Instant
`O(log n)`	Logarithmic	Binary search, BST/AVL ops	Very fast
`O(n)`	Linear	Single loop, LinkedList search	Fast
`O(n log n)`	Log-linear	Merge Sort, Heap Sort	Moderate
`O(n²)`	Quadratic	Bubble Sort, nested loops	Slow for large n
`O(2■)`	Exponential	Recursive subsets, naive DP	Avoid

Common Mistakes to Avoid

- **NullPointerException** — check head != null before traversal; check arr != null before arr.length
- **ArrayIndexOutOfBoundsException** — index must satisfy: 0 <= i < arr.length
- **ConcurrentModificationException** — never modify a collection while iterating with for-each

• **Integer overflow in mid** — use mid = lo + (hi - lo) / 2, NOT (lo + hi) / 2

• **Stack overflow in DFS** — for large graphs, use iterative DFS with an explicit Stack

• **Off-by-one in LinkedList** — stop when current.next == null (not current == null) to avoid NPE

• **Wrong HashMap key type** — use Integer, not int (autoboxed); know that equals() checks value, not reference

• **Modifying during stream** — collect results first, then apply changes to the original collection

Your next steps after completing this course: (1) Solve 100 LeetCode problems (40 Easy, 50 Medium, 10 Hard). (2) Learn the 5 core Sorting algorithms: Bubble, Selection, Insertion, Merge, Quick. (3) Study Dynamic Programming — it combines data structures with recursion. (4) Aim to code every structure from scratch in under 20 minutes without notes.