**Day1 :**

**Algorithms**

**1. What is an algorithm?**

An **algorithm** is a well-defined, step-by-step procedure designed to perform a specific task or solve a problem. It consists of a finite set of instructions that take an input, process it, and produce an output.

**Example: Finding the Largest Number in an Array**

1. Take an array of numbers as input.
2. Assume the first element is the largest.
3. Iterate through the array and compare each element with the assumed largest.
4. If a larger element is found, update the assumed largest.
5. Return the largest number.

**Characteristics of an Algorithm:**

* **Well-defined inputs and outputs:** Every algorithm takes input and produces output.
* **Definiteness:** Every step is precisely defined.
* **Finiteness:** The algorithm must terminate after a finite number of steps.
* **Effectiveness:** Each step must be simple enough to be executed in a finite amount of time.

**2. Why is algorithm analysis important?**

Algorithm analysis helps in **evaluating** and **comparing** algorithms to determine the most efficient one. It allows us to understand how an algorithm will perform as input size grows.

**Key Reasons:**

* **Efficiency:** Determines how fast or slow an algorithm executes.
* **Optimization:** Helps in choosing the best approach for large-scale problems.
* **Scalability:** Ensures the algorithm can handle large inputs effectively.

**Example: Searching an Element in an Array**

* **Linear Search:** O(n) time complexity (searching element one by one).
* **Binary Search:** O(log n) time complexity (divides the array in half each step). For large inputs, binary search is significantly faster.

**3. What are the key criteria for analyzing an algorithm?**

The **two primary criteria** are:

1. **Time Complexity:** How execution time increases as input size grows.
   * **Best Case, Average Case, Worst Case** complexities.
   * Measured using **Big-O Notation** (e.g., O(1), O(n), O(log n), O(n²)).
2. **Space Complexity:** The amount of memory required.
   * Includes **input space**, **auxiliary space**, and **stack space** (for recursion).

Example:

* **Bubble Sort:** O(n²) (inefficient for large inputs).
* **Merge Sort:** O(n log n) (better efficiency).

**4. What are the different approaches to developing algorithms?**

1. **Brute Force:** Try all possible solutions (e.g., checking all pairs in a list).
2. **Divide and Conquer:** Break problems into smaller subproblems (e.g., Merge Sort).
3. **Greedy Algorithm:** Make the best choice at each step (e.g., Kruskal’s Algorithm).
4. **Dynamic Programming:** Solve overlapping subproblems efficiently (e.g., Fibonacci).
5. **Backtracking:** Explore all possibilities but discard non-promising ones (e.g., N-Queens).

**5. What are the characteristics of a good algorithm?**

1. **Correctness:** Produces correct output for every valid input.
2. **Efficiency:** Uses minimal time and space.
3. **Finiteness:** Must complete execution in a finite number of steps.
4. **Generality:** Works for different inputs, not just one case.
5. **Simplicity & Clarity:** Should be easy to understand and implement.

**Data Structures**

**1. What are different types of data structures?**

* **Linear Data Structures:**
  + **Arrays:** Fixed-size contiguous memory storage.
  + **Linked Lists:** Dynamic memory with nodes connected via pointers.
  + **Stacks:** LIFO structure (Last In, First Out).
  + **Queues:** FIFO structure (First In, First Out).
* **Non-Linear Data Structures:**
  + **Trees:** Hierarchical structure with nodes (e.g., Binary Tree, BST).
  + **Graphs:** Nodes connected by edges (e.g., Social Networks).
* **Hashing:** Hash tables, hash maps (fast lookups).
* **Specialized Structures:** Tries, Heaps
* **2. What is the difference between an array and a linked list?**

| **Feature** | **Array** | **Linked List** |
| --- | --- | --- |
| **Memory Allocation** | Contiguous memory (fixed size) | Dynamic memory (flexible size) |
| **Insertion/Deletion** | Expensive (O(n) for shifting) | Efficient (O(1) at head) |
| **Access Time** | O(1) (direct access via index) | O(n) (traversal required) |
| **Extra Memory** | No extra space required | Requires additional space for pointers |

**3. How does a stack work? Provide a real-time example.**

A **stack** follows **LIFO (Last In, First Out)** principle.

**Operations:**

* **Push:** Insert an element.
* **Pop:** Remove the top element.
* **Peek:** Get the top element.

**Real-world Example:** **Undo feature in text editors**

* Each action (typing, deleting) is pushed onto a stack.
* When "Undo" is pressed, the last action is popped.

**4. What are the operations on a queue? Explain different types of queues.**

**Operations:**

* **Enqueue:** Insert element at the rear.
* **Dequeue:** Remove element from the front.
* **Peek:** Get the front element.

**Types of Queues:**

1. **Simple Queue:** Follows FIFO.
2. **Circular Queue:** Connects rear to front, avoiding wasted space.
3. **Priority Queue:** Dequeues elements based on priority.
4. **Deque (Double-Ended Queue):** Insertions & deletions allowed from both ends.

**5. What is a graph? Explain different types of graphs.**

A **graph** is a set of nodes (**vertices**) connected by **edges**.

**Types of Graphs:**

1. **Directed Graph:** Edges have directions.
2. **Undirected Graph:** Edges don’t have direction.
3. **Weighted Graph:** Each edge has a weight/cost.
4. **Unweighted Graph:** All edges have the same cost.
5. **Cyclic Graph:** Contains cycles.
6. **Acyclic Graph:** Does not contain cycles.

**Recursion in Java - Theoretical Explanation**

**1. What is Recursion, and How Does it Work in Java?**

Recursion is a technique in programming where a method calls itself to solve a problem. It continues to call itself with modified parameters until a stopping condition, called the **base case**, is met. Each recursive call creates a new stack frame in memory, and once the base case is reached, the function returns values back through the call stack.

**2. Why is Recursion Used in Java Programming?**

Recursion is used because it:

* **Simplifies complex problems** that can be broken down into smaller subproblems.
* **Is useful in divide-and-conquer algorithms**, such as sorting and searching techniques.
* **Is essential for data structures** like trees and graphs, which are naturally recursive in structure.
* **Helps in backtracking problems**, such as generating permutations or solving mazes.

**3. Advantages and Disadvantages of Recursion in Java**

**Advantages:**

* **Improves code readability** by reducing repetitive logic.
* **Best suited for hierarchical and tree-based structures.**
* **Reduces code complexity** for problems that naturally fit recursive patterns.

**Disadvantages:**

* **Consumes more memory** due to multiple function calls stored in the call stack.
* **Can cause StackOverflowError** if recursion depth is too high.
* **Slower execution** compared to iteration due to the overhead of function calls.

**4. Difference Between Recursion and Iteration in Java**

| **Feature** | **Recursion** | **Iteration** |
| --- | --- | --- |
| **Definition** | A function calls itself repeatedly | Uses loops like for, while |
| **Memory Usage** | Uses stack memory for each function call | Uses constant memory (loop variables) |
| **Performance** | Slower due to function call overhead | Faster as no function calls are required |
| **Best Used For** | Problems that are naturally recursive (tree traversal, Fibonacci, backtracking) | Problems with known and fixed iterations (loop-based calculations) |
| **Risk** | StackOverflowError if no base case is defined | Infinite loop if condition is not correctly set |

**5. What Are Base Cases in Recursion, and Why Are They Important?**

A **base case** is a condition in a recursive function that stops further recursive calls. It prevents infinite recursion and ensures that the function eventually terminates.

Without a base case, the function would keep calling itself indefinitely, leading to a **StackOverflowError**. The base case should be chosen carefully to correctly solve the problem and return results back up the call stack.

**6. Problems That Can Be Solved Using Recursion in Java**

**1. Mathematical Computations**

* Factorial Calculation
* Fibonacci Series
* Greatest Common Divisor (GCD)
* Power of a Number

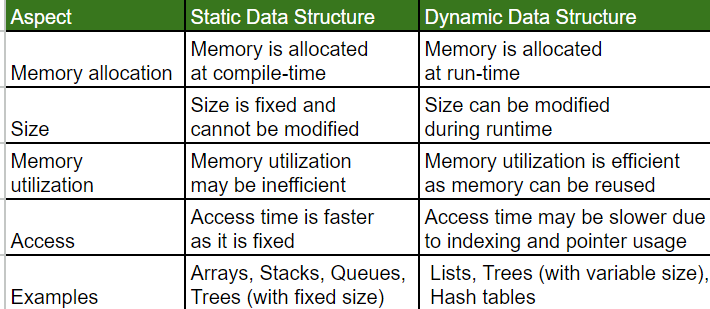
**2. Data Structure Operations**

* Tree Traversals (Preorder, Inorder, Postorder)
* Graph Traversal (Depth-First Search - DFS)
* Linked List Operations (Reverse, Merge)

**3. Algorithmic Problems**

* Merge Sort
* Quick Sort
* Binary Search
* Tower of Hanoi
* Backtracking problems (Sudoku Solver, N-Queens, Maze Solver)

**Difference between static Data Structure and Dynam**ic D**ata Structure**



**Day 2 :-**

**What is Recursion?**

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. Instead of using loops, recursion repeatedly breaks down a problem into subproblems until a base condition is met.

**How Recursion Works?**

A recursive function typically has two main components:

1. **Base Case (Stopping Condition):**
   * This is the condition that stops the recursion.
   * Without a base case, the function will keep calling itself indefinitely, causing a stack overflow.
2. **Recursive Case (Self-Call):**
   * This is where the function calls itself with a smaller or simpler input.
   * Each call works on a smaller part of the problem until it reaches the base case.

**How It Works?**

Each call splits into two more calls until the base cases (0 and 1) are reached. Then, the function returns values back up the recursion tree.

Advantages of Recursion

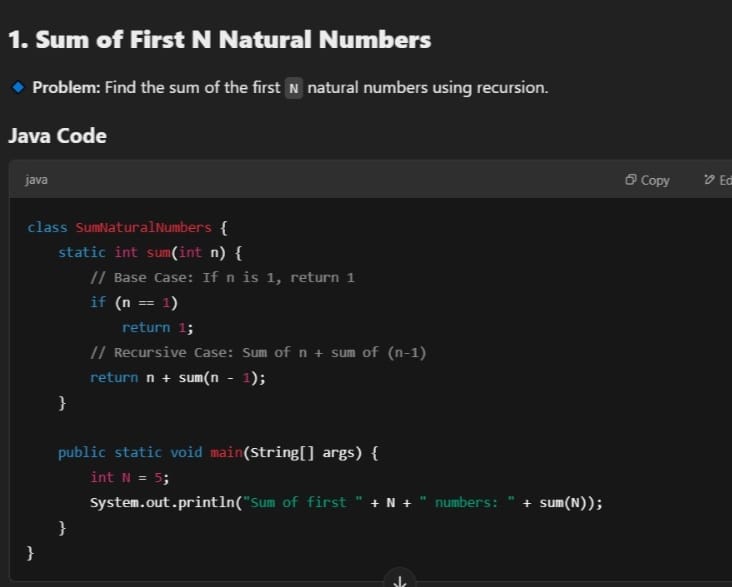
✅ Simplifies complex problems – Helps break down problems into smaller ones.  
✅ Easier to read and write – Some problems (like tree traversal) are easier with recursion.  
✅ Efficient for divide-and-conquer problems – Used in QuickSort, MergeSort, etc.

Disadvantages of Recursion

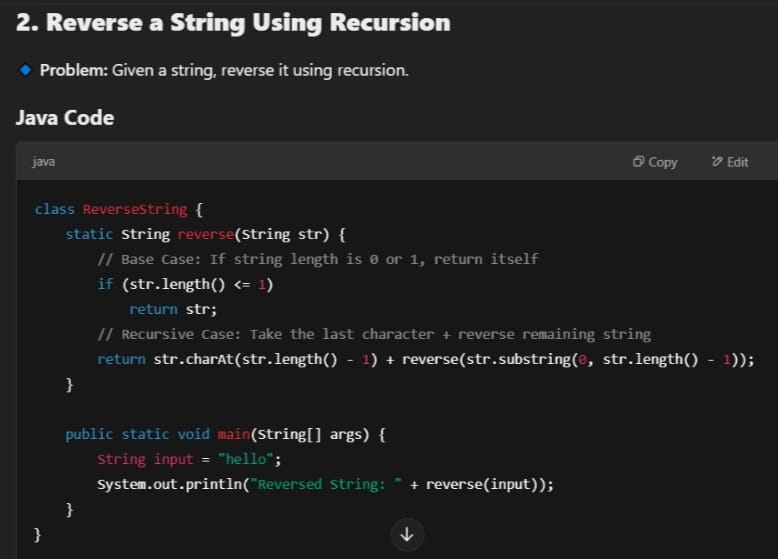
❌ High memory usage – Each function call is stored in memory (stack).  
❌ Slower for large inputs – Recursive calls create overhead compared to loops.  
❌ Risk of infinite recursion – If no proper base case, it results in a stack overflow.

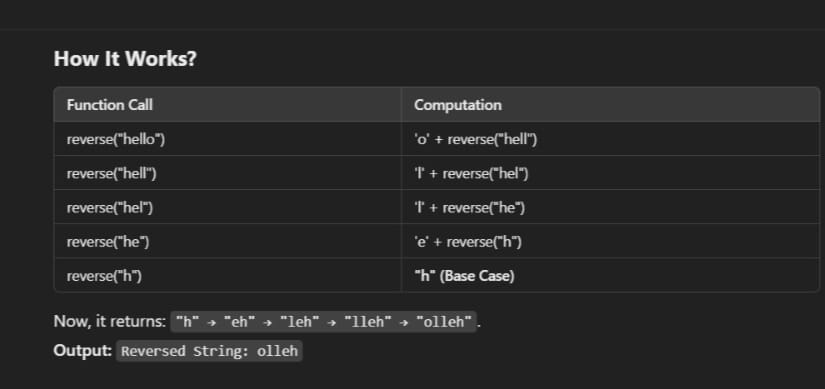
When to Use Recursion?

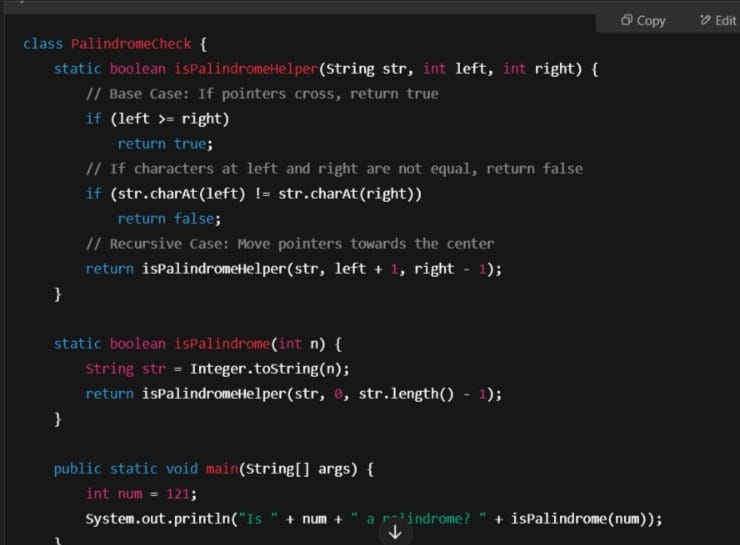
* When a problem can be divided into smaller subproblems (e.g., factorial, Fibonacci).
* When using tree or graph traversal (e.g., DFS, binary tree operations).
* When implementing divide-and-conquer algorithms (e.g., MergeSort, QuickSort).

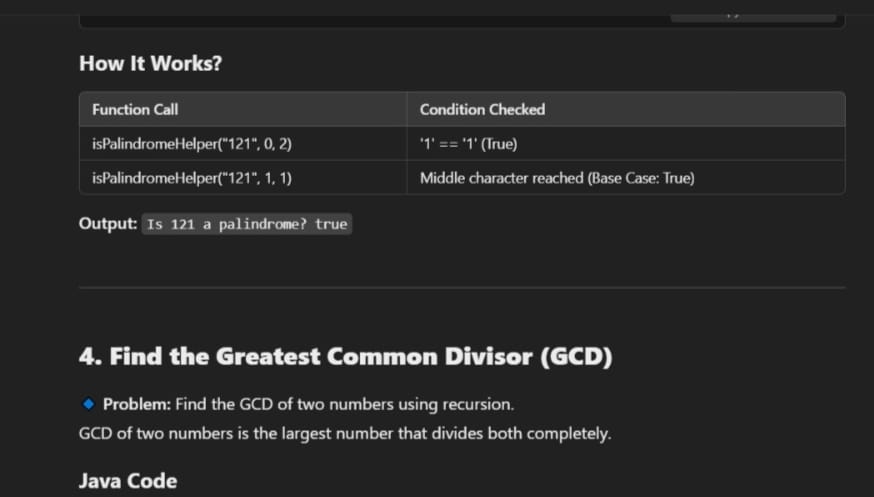
****

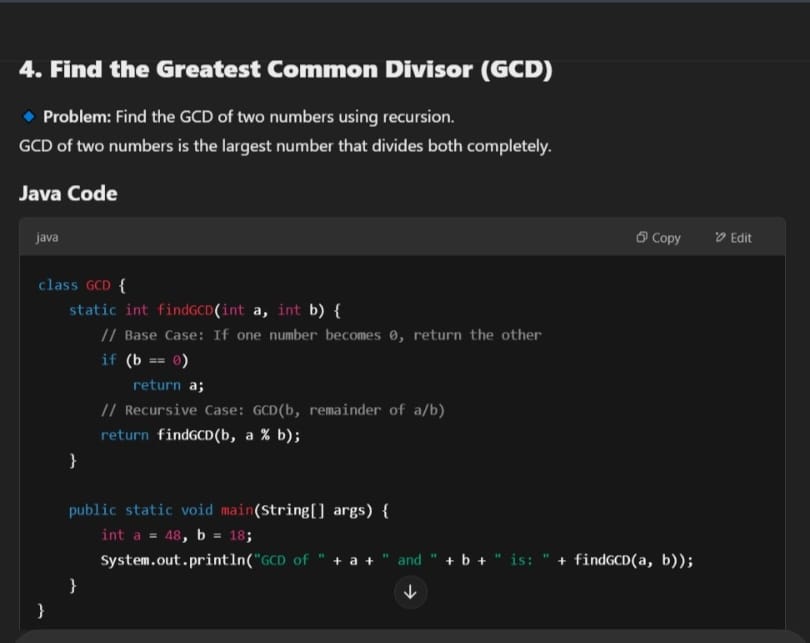
****

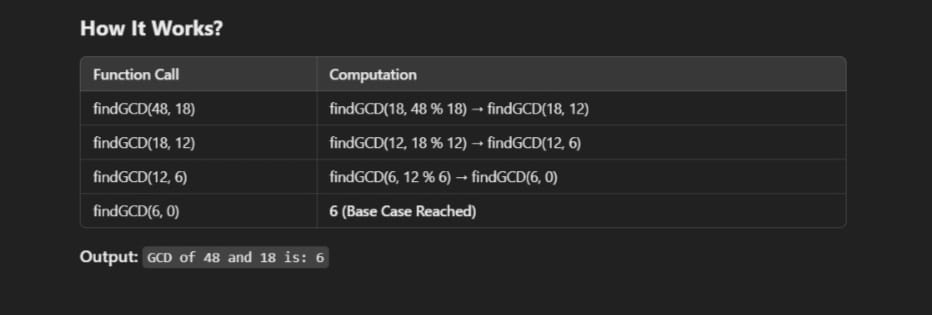
****

****

****

****

****

****

The Tower of Hanoi is a classic mathematical puzzle that consists of three rods (A, B, and C) and a set of disks of different sizes that can be stacked on top of each other. The objective of the puzzle is to move the entire stack of disks from rod A to rod C, subject to the following rules:

1. Only one disk can be moved at a time.
2. Each move involves taking the top disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No disk may be placed on top of a smaller disk.

**Java Example:**

Here's a Java implementation of the Tower of Hanoi problem:

**java**

public class TowerOfHanoi {

public static void towerOfHanoi(int n, char fromRod, char toRod, char auxRod) {

if (n == 1) {

System.out.println("Move disk 1 from rod " + fromRod + " to rod " + toRod);

return;

}

towerOfHanoi(n - 1, fromRod, auxRod, toRod);

System.out.println("Move disk " + n + " from rod " + fromRod + " to rod " + toRod);

towerOfHanoi(n - 1, auxRod, toRod, fromRod);

}

public static void main(String[] args) {

int n = 3; *// number of disks*

towerOfHanoi(n, 'A', 'C', 'B');

}

}

**Step-by-Step Breakdown:**

Let's break down the recursive calls for the Tower of Hanoi problem with 3 disks.

1. towerOfHanoi(3, 'A', 'C', 'B'):
   * Move the top 2 disks from rod A to rod B, using rod C as a temporary storage.
   * Move the 3rd disk from rod A to rod C.
   * Move the 2 disks from rod B to rod C, using rod A as a temporary storage.
2. towerOfHanoi(2, 'A', 'B', 'C'):
   * Move the top disk from rod A to rod C.
   * Move the 2nd disk from rod A to rod B.
   * Move the top disk from rod C to rod B.
3. towerOfHanoi(1, 'A', 'C', 'B'):
   * Move the disk from rod A to rod C.
4. towerOfHanoi(2, 'B', 'C', 'A'):
   * Move the top disk from rod B to rod A.
   * Move the 2nd disk from rod B to rod C.
   * Move the top disk from rod A to rod C.

**Tree Representation:**

Here's a tree representation of the recursive calls for the Tower of Hanoi problem with 3 disks:

**code**

towerOfHanoi(3, 'A', 'C', 'B')

/ \

towerOfHanoi(2, 'A', 'B', 'C') Move disk 3 from rod A to rod C

/ \

towerOfHanoi(1, 'A', 'C', 'B') Move disk 2 from rod A to rod B

Move disk 1 from rod A to rod C

Move disk 1 from rod C to rod B

Move disk 2 from rod B to rod C

Move disk 1 from rod B to rod C

**Stack Call Representation:**

Here's a stack call representation of the recursive calls for the Tower of Hanoi problem with 3 disks:

**code**

Stack:

1. towerOfHanoi(3, 'A', 'C', 'B')

2. towerOfHanoi(2, 'A', 'B', 'C')

3. towerOfHanoi(1, 'A', 'C', 'B')

4. Move disk 1 from rod A to rod C

5. towerOfHanoi(1, 'A', 'B', 'C')

6. Move disk 2 from rod A to rod B

7. Move disk 1 from rod C to rod B

8. Move disk 2 from rod B to rod C

9. Move disk 1 from rod B to rod C

10. Move disk 3 from rod A to rod C

11. towerOfHanoi(2, 'B', 'C', 'A')

12. towerOfHanoi(1, 'B', 'A', 'C')

13. Move disk 1 from rod B to rod A

14. Move disk 2 from rod B to rod C

15. Move disk 1 from rod A to rod C

**Day 3**

**Arrays in Data Structures with Respect to Java**

**What is an Array?**

An **array** is a **fixed-size, sequential collection of elements** of the same data type. It is a linear data structure where elements are stored in **contiguous memory locations**.

**Key Characteristics of Arrays**

* **Homogeneous Elements**: All elements in an array must be of the same data type.
* **Fixed Size**: The size of an array is defined at the time of declaration and cannot be changed dynamically.
* **Indexed Access**: Each element in an array is accessed using an index, starting from 0 for the first element.
* **Efficient Retrieval**: Arrays provide **O(1) time complexity** for accessing elements because of direct indexing.

**Declaring and Initializing an Array in Java**

**1. Declaration of an Array**

In Java, an array is declared using the following syntax:

java

**datatype[] arrayName; // Preferred way**

// OR

**datatype arrayName[]; // Also valid but less preferred**

**Example:**

java

**int[] numbers; // Declaring an array of integers**

**2. Array Initialization**

There are different ways to initialize an array in Java:

**a) Using new Keyword**

**int[] numbers = new int[5];** // Creates an array of size 5 (default values = 0)

This creates an integer array with **5 elements** (default values = 0 for integers).

**b) Declaring and Initializing in One Step.**

**int[] numbers = {10, 20, 30, 40, 50};** // Direct initialization

This creates an array with predefined values.

**Accessing Elements of an Array**

We use an **index** to access elements in an array. The first index is 0, and the last index is size - 1.

**public class ArrayExample {**

**public static void main(String[] args) {**

**int[] numbers = {10, 20, 30, 40, 50};**

**// Accessing elements using index**

**System.out.println("First element: " + numbers[0]); // 10**

**System.out.println("Second element: " + numbers[1]); // 20**

**}**

**}**

* **Types of Arrays in Java**

**1. One-Dimensional Array**

A **1D array** is a simple array where elements are stored in a single row.

Example:

**public class OneDArray {**

**public static void main(String[] args) {**

**int[] arr = new int[3]; // Array of size 3**

**// Assigning values**

**arr[0] = 5;**

**arr[1] = 10;**

**arr[2] = 15;**

**// Printing elements**

**for (int i = 0; i < arr.length; i++) {**

**System.out.println("Element at index " + i + ": " + arr[i]);**

**}**

**}**

**}**

**Output:**

**Element at index 0: 5**

**Element at index 1: 10**

**Element at index 2: 15**

* **Multi-Dimensional Arrays**

A **multi-dimensional array** is an array of arrays, commonly used for matrices.

**Example: 2D Array**

**public class TwoDArray {**

**public static void main(String[] args) {**

**// Declare and initialize a 2D array**

**int[][] matrix = {**

**{1, 2, 3},**

**{4, 5, 6},**

**{7, 8, 9}**

**};**

**// Printing the 2D array**

**for (int i = 0; i < matrix.length; i++) {**

**for (int j = 0; j < matrix[i].length; j++) {**

**System.out.print(matrix[i][j] + " ");**

**}**

**System.out.println();**

**}**

**}**

**}**

**Output:**

1 2 3

4 5 6

7 8 9

**Array Operations in Java**

**1. Traversing an Array**

Traversing means accessing each element in an array **one by one**.

* Using a **for loop**:

**for (int i = 0; i < arr.length; i++) {**

**System.out.println(arr[i]);**

**}**

* Using an **enhanced for loop**:

**for (int num : arr) {**

**System.out.println(num);**

**}**

**2. Inserting Elements in an Array**

Since arrays have a **fixed size**, inserting an element means **replacing** an existing element.

Example:

arr[2] = 25; // Replaces the element at index 2 with 25

For inserting elements **dynamically**, we use **ArrayList** instead of arrays.

* **Searching for an Element in an Array**

**a) Linear Search (O(n))**

**public class LinearSearch {**

**public static int search(int[] arr, int key) {**

**for (int i = 0; i < arr.length; i++) {**

**if (arr[i] == key) {**

**return i; // Return index if found**

**}**

**}**

**return -1; // Return -1 if not found**

**}**

**public static void main(String[] args) {**

**int[] arr = {10, 20, 30, 40, 50};**

**int key = 30;**

**int index = search(arr, key);**

**System.out.println("Element found at index: " + index);**

**}**

**}**

**b) Binary Search (O(log n)) [for Sorted Arrays]**

**public class BinarySearchExample {**

**public static void main(String[] args) {**

**int[] arr = {10, 20, 30, 40, 50};**

**int key = 30;**

**int index = Arrays.binarySearch(arr, key);**

**System.out.println("Element found at index: " + index);**

**}**

**}**

**4. Deleting an Element from an Array**

Since Java arrays **cannot shrink in size**, we usually create a **new array without the deleted element**.

Example:

**public class DeleteElement {**

**public static int[] deleteElement(int[] arr, int index) {**

**if (index < 0 || index >= arr.length) {**

**return arr; // Invalid index, return original array**

**}**

**int[] newArr = new int[arr.length - 1];**

**for (int i = 0, j = 0; i < arr.length; i++) {**

**if (i != index) {**

**newArr[j++] = arr[i];**

**}**

**}**

**return newArr;**

**}**

**public static void main(String[] args) {**

**int[] arr = {10, 20, 30, 40, 50};**

**int[] newArr = deleteElement(arr, 2); // Delete element at index 2**

**for (int num : newArr) {**

**System.out.print(num + " ");**

**}**

**}**

**}**

**Output:**

10 20 40 50

**Advantages of Arrays**

✅ **Fast Access** – Direct index-based access (O(1))  
✅ **Cache-Friendly** – Stored in contiguous memory locations  
✅ **Easy Traversal** – Using loops

* **Linear Search**

**Linear Search** is a **simple searching algorithm** where we **traverse** the array **sequentially** and compare each element with the target value.

* It **works on both sorted and unsorted arrays**.
* **Time Complexity**: **O(n)** (Worst case: The element is at the last index or not present)
* **Space Complexity**: **O(1)** (No extra memory is used)

**Implementation of Linear Search in Java**

**public class LinearSearch {**

**// Linear search function**

**public static int search(int[] arr, int key) {**

**for (int i = 0; i < arr.length; i++) {**

**if (arr[i] == key) {**

**return i; // Return the index where key is found**

**}**

**}**

**return -1; // Return -1 if key is not found**

**}**

**public static void main(String[] args) {**

**int[] arr = {10, 20, 30, 40, 50}; // Example array**

**int key = 30; // Element to search**

**int index = search(arr, key);**

**if (index != -1) {**

**System.out.println("Element found at index: " + index);**

**} else {**

**System.out.println("Element not found");**

**}**

**}**

**}**

**Dry Run (Example)**

**arr = {10, 20, 30, 40, 50}, key = 30**

**Step 1: Compare key with arr[0] → 10 (No match)**

**Step 2: Compare key with arr[1] → 20 (No match)**

**Step 3: Compare key with arr[2] → 30 (Match found at index 2)**

Output: Element found at index: 2

* **Binary Search**

**Binary Search** is an **efficient searching algorithm** that works **only on sorted arrays**. It uses the **divide and conquer** technique:

1. Find the middle element of the array.
2. Compare the middle element with the target.
   * If it matches, return the index.
   * If it is smaller, search the **right half**.
   * If it is larger, search the **left half**.
3. Repeat until the element is found or the search space is empty.

* **Time Complexity**: **O(log n)** (as the array is divided in half each time)
* **Space Complexity**: **O(1)** (iterative) or **O(log n)** (recursive due to function calls)

**Implementation of Binary Search in Java**

**A. Iterative Binary Search**

**import java.util.Arrays;**

**public class BinarySearchIterative {**

**// Iterative Binary Search function**

**public static int binarySearch(int[] arr, int key) {**

**int left = 0, right = arr.length - 1;**

**while (left <= right) {**

**int mid = left + (right - left) / 2;**

**// If key is found at mid**

**if (arr[mid] == key) {**

**return mid;**

**}**

**// If key is smaller, search left half**

**if (arr[mid] > key) {**

**right = mid - 1;**

**}**

**// If key is larger, search right half**

**else {**

**left = mid + 1;**

**}**

**}**

**return -1; // Element not found**

**}**

**public static void main(String[] args) {**

**int[] arr = {10, 20, 30, 40, 50}; // Must be sorted**

**int key = 30;**

**int index = binarySearch(arr, key);**

**if (index != -1) {**

**System.out.println("Element found at index: " + index);**

**} else {**

**System.out.println("Element not found");**

**}**

**}**

**}**

**Dry Run (Example)**

arr = {10, 20, 30, 40, 50}, key = 30

Step 1: mid = (0+4)/2 = 2, arr[2] = 30 (Match found)

Output: Element found at index: 2

**B. Recursive Binary Search**

**public class BinarySearchRecursive {**

**// Recursive Binary Search function**

**public static int binarySearch(int[] arr, int left, int right, int key) {**

**if (left > right) {**

**return -1; // Base case: element not found**

**}**

**int mid = left + (right - left) / 2;**

**// If key is found**

**if (arr[mid] == key) {**

**return mid;**

**}**

**// If key is smaller, search left half**

**else if (arr[mid] > key) {**

**return binarySearch(arr, left, mid - 1, key);**

**}**

**// If key is larger, search right half**

**else {**

**return binarySearch(arr, mid + 1, right, key);**

**}**

**}**

**public static void main(String[] args) {**

**int[] arr = {10, 20, 30, 40, 50}; // Must be sorted**

**int key = 30;**

**int index = binarySearch(arr, 0, arr.length - 1, key);**

**if (index != -1) {**

**System.out.println("Element found at index: " + index);**

**} else {**

**System.out.println("Element not found");**

**}**

**}**

**}**

* **Comparison: Linear Search vs Binary Search**

| **Feature** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Best for** | Unsorted lists, small datasets | Sorted lists, large datasets |
| **Time Complexity** | O(n) | O(log n) |
| **Space Complexity** | O(1) | O(1) (Iterative) or O(log n) (Recursive) |
| **Approach** | Sequential search | Divide and Conquer |
| **Performance** | Slower for large inputs | Much faster for large inputs |
| **Works on Sorted Arrays?** | ✅ No | ✅ Yes |

**Binary Search using 1 + (h - 1) / 2 Formula**

**public class BinarySearchCustomMid {**

**// Binary Search function**

**public static int binarySearch(int[] arr, int key) {**

**int left = 0, right = arr.length - 1;**

**while (left <= right) {**

**// Mid calculation using 1 + (high - 1) / 2**

**int mid = 1 + (right - 1) / 2;**

**// If key is found at mid**

**if (arr[mid] == key) {**

**return mid;**

**}**

**// If key is smaller, search left half**

**if (arr[mid] > key) {**

**right = mid - 1;**

**}**

**// If key is larger, search right half**

**else {**

**left = mid + 1;**

**}**

**}**

**return -1; // Element not found**

**}**

**public static void main(String[] args) {**

**int[] arr = {10, 20, 30, 40, 50}; // Must be sorted**

**int key = 30;**

**int index = binarySearch(arr, key);**

**if (index != -1) {**

**System.out.println("Element found at index: " + index);**

**} else {**

**System.out.println("Element not found");**

**}**

**}**

**}**

**Explanation of 1 + (h - 1) / 2 Formula**

* The standard formula for mid is:

mid=low+(high−low)/2mid = low + (high - low) / 2mid=low+(high−low)/2

* In this program, we used a different approach:

mid=1+(h−1)/2mid = 1 + (h - 1) / 2mid=1+(h−1)/2

This formula modifies how mid is calculated but still follows a **divide-and-conquer** approach.

**Dry Run**

**Array: {10, 20, 30, 40, 50}**

**Target Key: 30**

| **Step** | **Left (l)** | **Right (h)** | **Mid (1 + (h-1)/2)** | **Element at Mid** | **Decision** |
| --- | --- | --- | --- | --- | --- |
| 1 | 0 | 4 | 1 + (4-1)/2 = 2 | 30 | Found! |

**Output:**

Element found at index: 2

* **Multi-Dimensional Arrays in Data Structures**

**1. Adjacency Matrix (Graph Representation)**

* Used to represent a **graph** as a **2D array**.
* If there is an edge between two nodes, the corresponding matrix element is 1, otherwise 0.
* **Pros**: Simple and easy to implement.
* **Cons**: Uses more space for sparse graphs.

**2. Diagonal Matrix**

* A **square matrix** where only the **main diagonal** contains non-zero elements.
* Used in **mathematical and scientific computing**.
* **Pros**: Efficient storage and fast operations.
* **Cons**: Limited applications compared to other matrix types.

**3. Sparse Matrix**

* A **matrix with a majority of zero elements**.
* Stored in **compressed form** to save memory.
* Used in **machine learning, graph algorithms, and scientific computing**.
* **Pros**: Saves memory and processing time.
* **Cons**: Requires special handling for operations.

**4. Tridiagonal Matrix**

* A **square matrix** where only the **main diagonal, upper diagonal, and lower diagonal** contain non-zero values.
* Used in **numerical methods** and **solving differential equations**.
* **Pros**: Reduced storage compared to full matrices.
* **Cons**: Not suitable for general-purpose matrix operations.
* **Time and Space Complexity in Data Structures**

**1. Time Complexity**

* It represents the **amount of time** an algorithm takes to complete based on input size n.
* It is expressed using **Big-O notation** (O(n), O(log n), etc.).
* **Common time complexities:**
  + **O(1)** → Constant time (e.g., accessing an element in an array).
  + **O(log n)** → Logarithmic time (e.g., binary search).
  + **O(n)** → Linear time (e.g., traversing an array).
  + **O(n log n)** → Log-linear time (e.g., merge sort, quick sort).
  + **O(n²)** → Quadratic time (e.g., bubble sort, selection sort).
  + **O(2^n), O(n!)** → Exponential and factorial time (very slow).

**2. Space Complexity**

* It represents the **amount of memory** required by an algorithm based on input size n.
* It includes:
  + **Auxiliary space** (temporary storage like variables, recursion stack).
  + **Input space** (memory required to store input).
* **Common space complexities:**
  + **O(1)** → Constant space (e.g., using a few extra variables).
  + **O(n)** → Linear space (e.g., storing n elements in an array).
  + **O(n²)** → Quadratic space (e.g., 2D matrices).
  + **O(log n)** → Logarithmic space (e.g., recursive binary search).

**Time and Space Complexity of Array Operations**

| **Operation** | **Time Complexity (Average/Worst Case)** | **Space Complexity** | **Explanation** |
| --- | --- | --- | --- |
| **Access** | O(1) | O(1) | Direct access using an index. |
| **Search (Linear Search)** | O(n) | O(1) | In worst case, it searches through all elements. |
| **Search (Binary Search - sorted array)** | O(log n) | O(1) | Repeatedly divides the array in half. |
| **Insertion (At End)** | O(1) | O(1) | Directly inserting at the last index. |
| **Insertion (At Middle or Start)** | O(n) | O(1) | Requires shifting elements to make space. |
| **Deletion (From End)** | O(1) | O(1) | Directly removes the last element. |
| **Deletion (From Middle or Start)** | O(n) | O(1) | Requires shifting elements to fill the gap. |
| **Traversal** | O(n) | O(1) | Visiting all elements once. |

**Key Observations**

* **Insertion and deletion at the end** are faster (**O(1)**).
* **Insertion and deletion in the middle or beginning** require shifting and take **O(n)** time.
* **Searching in an unsorted array** requires **O(n)**, while **binary search in a sorted array** takes **O(log n)**.
* **Java Program of Insertion, Deletion, Access, sorting, travesing**

Java Program for Array Operations

**import java.util.Arrays;**

**import java.util.Scanner;**

**public class ArrayOperations {**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**int[] arr = new int[10]; // Initial array with fixed size (Space Complexity: O(n))**

**int size = 5; // Current size of the array**

**// Initial elements**

**arr[0] = 10;**

**arr[1] = 50;**

**arr[2] = 30;**

**arr[3] = 40;**

**arr[4] = 20;**

**System.out.println("Initial Array: " + Arrays.toString(Arrays.copyOf(arr, size)));**

**// 1. \*\*Insertion Operation\*\* (O(n) in worst case)**

**insertElement(arr, size, 25, 2); // Insert 25 at index 2**

**size++;**

**// 2. \*\*Search Operation\*\* (O(n) for linear search)**

**searchElement(arr, size, 30);**

**// 3. \*\*Deletion Operation\*\* (O(n) in worst case)**

**deleteElement(arr, size, 40);**

**size--;**

**// 4. \*\*Traversal Operation\*\* (O(n))**

**traverseArray(arr, size);**

**// 5. \*\*Sorting Operation\*\* (O(n²) using Bubble Sort)**

**bubbleSort(arr, size);**

**System.out.println("Sorted Array: " + Arrays.toString(Arrays.copyOf(arr, size)));**

**scanner.close();**

**}**

**// \*\*1. Insert Element at a given position (O(n) in worst case)\*\***

**static void insertElement(int[] arr, int size, int element, int position) {**

**if (position > size) {**

**System.out.println("Invalid position!");**

**return;**

**}**

**for (int i = size; i > position; i--) { // Shift elements**

**arr[i] = arr[i - 1];**

**}**

**arr[position] = element;**

**System.out.println("After Insertion: " + Arrays.toString(Arrays.copyOf(arr, size + 1)));**

**}**

**// \*\*2. Search for an element (Linear Search O(n))\*\***

**static void searchElement(int[] arr, int size, int key) {**

**for (int i = 0; i < size; i++) {**

**if (arr[i] == key) {**

**System.out.println("Element " + key + " found at index " + i);**

**return;**

**}**

**}**

**System.out.println("Element " + key + " not found.");**

**}**

**// \*\*3. Delete an element (O(n) in worst case)\*\***

**static void deleteElement(int[] arr, int size, int key) {**

**int index = -1;**

**for (int i = 0; i < size; i++) {**

**if (arr[i] == key) {**

**index = i;**

**break;**

**}**

**}**

**if (index == -1) {**

**System.out.println("Element not found.");**

**return;**

**}**

**for (int i = index; i < size - 1; i++) { // Shift elements**

**arr[i] = arr[i + 1];**

**}**

**System.out.println("After Deletion: " + Arrays.toString(Arrays.copyOf(arr, size - 1)));**

**}**

**// \*\*4. Traverse the array (O(n))\*\***

**static void traverseArray(int[] arr, int size) {**

**System.out.print("Traversing Array: ");**

**for (int i = 0; i < size; i++) {**

**System.out.print(arr[i] + " ");**

**}**

**System.out.println();**

**}**

**// \*\*5. Bubble Sort (O(n²))\*\***

**static void bubbleSort(int[] arr, int size) {**

**for (int i = 0; i < size - 1; i++) {**

**for (int j = 0; j < size - i - 1; j++) {**

**if (arr[j] > arr[j + 1]) {**

**// Swap elements**

**int temp = arr[j];**

**arr[j] = arr[j + 1];**

**arr[j + 1] = temp;**

**}**

**}**

**}**

**}**

**}**

**Explanation of Each Operation with Time and Space Complexity**

| **Operation** | **Time Complexity** | **Space Complexity** | **Explanation** |
| --- | --- | --- | --- |
| **Insertion** | **O(n)** (worst case) | **O(1)** | Elements may need to be shifted to make space. |
| **Search** | **O(n)** (Linear Search) | **O(1)** | Each element is checked sequentially. |
| **Deletion** | **O(n)** (worst case) | **O(1)** | Elements after the deleted one need to be shifted. |
| **Traversal** | **O(n)** | **O(1)** | All elements are visited once. |
| **Sorting** | **O(n²)** (Bubble Sort) | **O(1)** | Compares each element and swaps them. |

**Example Output**

pgsql

CopyEdit

Initial Array: [10, 50, 30, 40, 20]

After Insertion: [10, 50, 25, 30, 40, 20]

Element 30 found at index 3

After Deletion: [10, 50, 25, 40, 20]

Traversing Array: 10 50 25 40 20

Sorted Array: [10, 20, 25, 40, 50]

**Key Changes in This Version**

✅ **Access operation removed**  
✅ **Sorting explicitly implemented using Bubble Sort (O(n²))**  
✅ **All operations are reflected with their time and space complexity**

**Day 4 :**

* **Understanding Time and Space Complexity in Detail**

**1. Introduction to Algorithm Analysis**

When we design an algorithm, we need to evaluate its efficiency in terms of the resources it consumes. The two main resources are:

* **Time Complexity** (How fast the algorithm runs)
* **Space Complexity** (How much memory it consumes)

Algorithm analysis helps in determining how an algorithm will perform as the input size grows.

**2. Time Complexity**

**Definition**

Time complexity refers to the amount of time an algorithm takes to complete as a function of the input size nnn. It helps us understand how an algorithm scales when the size of the input increases.

**Factors Affecting Time Complexity**

1. **Operations Count:** The number of basic operations executed by the algorithm.
2. **Input Size:** The number of elements in the input dataset.
3. **Processor Speed & Environment:** These are hardware-dependent and not considered in theoretical analysis.

**Types of Time Complexities with Examples**

1. **Constant Time O(1)O(1)O(1)**  
   An algorithm runs in constant time if the execution time does not change regardless of the input size.
2. **Linear Time O(n)O(n)O(n)**  
   The execution time grows linearly with input size.

def print\_elements(arr):

for element in arr:

print(element) # Runs 'n' times

``

**Detailed Explanation of Time and Space Complexity**

**Understanding Algorithm Analysis**

Algorithm analysis is the process of evaluating the efficiency of an algorithm. The efficiency is measured in terms of **time** (how fast the algorithm runs) and **space** (how much memory it consumes). The performance of an algorithm is important because it determines how well it will scale as the input size grows.

When analyzing an algorithm, we focus on **how the algorithm behaves as the input size increases** rather than on absolute execution time, as the execution time can vary based on hardware and system conditions.

**Time Complexity**

Time complexity refers to the amount of time an algorithm takes to complete as a function of the input size (nnn). It is expressed in terms of **asymptotic notation**, which describes the growth rate of the algorithm as nnn increases.

**Factors Affecting Time Complexity**

1. **Number of Operations:** The number of basic operations (such as comparisons, assignments, or arithmetic calculations) performed by the algorithm.
2. **Input Size:** The number of elements in the input dataset.
3. **Control Structures:** The complexity of loops, recursion, and conditional statements within the algorithm.
4. **Hardware & Environment:** While theoretical analysis ignores this, in practice, processor speed, cache memory, and parallelism also affect execution time.

**Types of Time Complexities**

1. **Constant Time O(1)O(1)O(1)**  
   If the execution time of an algorithm does not change regardless of input size, it is constant time.

def get\_first\_element(arr):

return arr[0] # Takes constant time, irrespective of array size

1. **Linear Time O(n)O(n)O(n)**  
   If the execution time grows proportionally with the input size, the algorithm is linear.
2. **Quadratic Time O(n2)O(n^2)O(n2)**  
   If the execution time is proportional to the square of the input size, the algorithm has quadratic complexity.

def print\_pairs(arr):

for i in range(len(arr)):

for j in range(len(arr)):

print(arr[i], arr[j]) # Nested loops run 'n^2' times

1. **Logarithmic Time O(log⁡n)O(\log n)O(logn)**  
   If the execution time increases logarithmically as input size increases, the algorithm has logarithmic complexity.

def binary\_search(arr, target):

low, high = 0, len(arr) - 1

while low <= high:

mid = (low + high) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

low = mid + 1

else:

high = mid - 1

return -1 # Logarithmic complexity

1. **Exponential Time O(2n)O(2^n)O(2n)**  
   If the execution time doubles with every increase in input size, the algorithm has exponential complexity.

def fibonacci(n):

if n <= 1:

return n

return fibonacci(n-1) + fibonacci(n-2) # Recursive calls grow exponentially

**Space Complexity**

Space complexity measures the amount of memory an algorithm uses as a function of input size. This includes:

* **Fixed part:** The memory required for constants, variables, and program code.
* **Variable part:** The memory required for dynamic structures like arrays, recursion stacks, and function calls.

**Types of Space Complexity**

1. **Constant Space O(1)O(1)O(1)**  
   The algorithm uses a fixed amount of memory regardless of input size.

def swap(a, b):

temp = a

a = b

b = temp # Uses constant memory

1. **Linear Space O(n)O(n)O(n)**  
   The memory usage grows linearly with input size.

def create\_list(n):

arr = [0] \* n # Uses 'n' memory

return arr

1. **Recursive Space O(n)O(n)O(n)**  
   When recursion is used, each function call adds a new frame to the stack, increasing memory consumption.

def factorial(n):

if n == 0:

return 1

return n \* factorial(n - 1) # Uses stack memory for each call

**Best Case, Worst Case, and Average Case Complexity**

When analyzing an algorithm, we consider different scenarios:

1. **Best Case Complexity:** The minimum time taken by an algorithm when the input is in the most favorable condition.
2. **Worst Case Complexity:** The maximum time taken when the input is in the worst possible condition.
3. **Average Case Complexity:** The expected time taken on random input cases.

For example, in **Binary Search**:

* **Best case:** O(1)O(1)O(1) (when the target is found in the first comparison)
* **Worst case:** O(log⁡n)O(\log n)O(logn) (when the target is at the end)
* **Average case:** O(log⁡n)O(\log n)O(logn) (based on probabilistic analysis)

**Asymptotic Notation**

Asymptotic notation helps describe an algorithm's growth rate as the input size increases.

1. **Big-O Notation O(f(n))O(f(n))O(f(n))**  
   Represents the upper bound (worst case) of an algorithm.
2. **Theta Notation Θ(f(n))\Theta(f(n))Θ(f(n))**  
   Represents the tight bound (average case) when an algorithm's complexity grows at the same rate in best and worst cases.
3. **Omega Notation Ω(f(n))\Omega(f(n))Ω(f(n))**  
   Represents the lower bound (best case), meaning the minimum time an algorithm takes.

For example, **Bubble Sort**:

* **Best Case:** Ω(n)\Omega(n)Ω(n) (if the array is already sorted)
* **Worst Case:** O(n2)O(n^2)O(n2) (if the array is in reverse order)
* **Average Case:** Θ(n2)\Theta(n^2)Θ(n2)

**Frequency Count Method**

This method counts the number of times each operation is executed in an algorithm to determine its complexity.

For example, in a loop:

python

CopyEdit

for i in range(n): # Runs 'n' times

for j in range(n): # Runs 'n' times inside

print(i, j)

The innermost statement executes n×n=n2n \times n = n^2n×n=n2 times, making the time complexity **O(n2)O(n^2)O(n2)**.

**Rules for Calculating Complexity**

1. Ignore constants (e.g., O(2n)O(2n)O(2n) is simplified to O(n)O(n)O(n)).
2. Consider the highest-order term (e.g., O(n2+n)O(n^2 + n)O(n2+n) simplifies to O(n2)O(n^2)O(n2)).
3. Drop lower-order terms in asymptotic analysis.

* **Permutation**

**1. Concept of Permutation Using Backtracking**

The backtracking approach systematically explores all possibilities by:

* Swapping elements to create different arrangements.
* Recursively calling the function to fix the next element.
* Reverting swaps (backtracking) to explore other possibilities.

For example, consider the string **"ABC"**. The possible permutations are:

1. ABC
2. ACB
3. BAC
4. BCA
5. CAB
6. CBA

**2. State Diagram Representation for "ABC"**

A **state diagram** helps visualize how permutations are generated using recursion and backtracking.

mathematica

CopyEdit

(ABC)

/ | \

A-BC B-AC C-AB

/ \ | / \

AB-C AC-B BA-C BC-A CA-B CB-A

| | | | | |

ABC ACB BAC BCA CAB CBA

Each **level** in the tree represents a step in the permutation process. The **edges** represent swaps, and each **leaf node** is a complete permutation.

* **First Level**: Choose one of the three letters to be placed at the first position.
* **Second Level**: Choose a letter from the remaining two for the second position.
* **Third Level**: Only one letter remains, so it's placed in the last position.

Once a permutation is formed, **backtracking** restores the original state to explore new possibilities.

**3. Code for Generating Permutations with Backtracking**

The code below implements the recursive permutation algorithm while using a **state diagram approach** for visualization.

**Java Implementation**

java

CopyEdit

public class Permutation {

// Function to generate permutations

public static void permute(char[] arr, int l, int r) {

if (l == r) {

System.out.println(String.valueOf(arr)); // Print the permutation

} else {

for (int i = l; i <= r; i++) {

swap(arr, l, i); // Swap to fix the current element

permute(arr, l + 1, r); // Recursive call for next positions

swap(arr, l, i); // Backtrack to restore the original order

}

}

}

// Swap function to exchange characters

private static void swap(char[] arr, int i, int j) {

char temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

public static void main(String[] args) {

String str = "ABC";

char[] arr = str.toCharArray();

permute(arr, 0, arr.length - 1);

}

}

**Explanation of Code Execution Using the State Diagram**

1. **Initial Call**
   * permute(['A', 'B', 'C'], 0, 2)
   * Fix A at index 0 and permute BC.
2. **First Recursive Call**
   * permute(['A', 'B', 'C'], 1, 2)
   * Fix B at index 1, permute C.
3. **Base Case Reached**
   * permute(['A', 'B', 'C'], 2, 2) → Output: **"ABC"**
   * Backtrack to swap B and C.
4. **Next Iteration**
   * permute(['A', 'C', 'B'], 1, 2)
   * Fix C at index 1, permute B.
5. **Base Case Reached Again**
   * permute(['A', 'C', 'B'], 2, 2) → Output: **"ACB"**
   * Backtrack to original ABC order.
6. **Continue Swapping**
   * Swap A and B, proceed with ['B', 'A', 'C']
   * Recursively generate new permutations.

This process continues until all possible arrangements are explored.

**4. Complexity Analysis**

* The **time complexity** is **O(n×n!)O(n \times n!)O(n×n!)** since each permutation requires O(n)O(n)O(n) operations and there are n!n!n! permutations.
* The **space complexity** is **O(n)O(n)O(n)** due to recursive function calls in the stack.

**5. Conclusion**

Using backtracking and recursive swapping, we generate all possible permutations efficiently. The **state diagram** helps in visualizing how recursion and backtracking work together to explore all possibilities. This method is fundamental in solving problems like anagrams, pathfinding, and brute-force searches.

**Day 5:**

**1. What is a Linked List in Java?**

A **Linked List** consists of nodes where each node contains two parts:

1. **Data** – The actual value stored in the node.
2. **Pointer (next reference)** – The address of the next node in the sequence.

Java provides an inbuilt **LinkedList** class under the java.util package, but we can also implement linked lists manually using custom node structures.

**2. Types of Linked List**

**(A) Singly Linked List**

Each node has a reference to the next node but not the previous one. Traversal is allowed in only one direction.

**(B) Doubly Linked List**

Each node contains two references:

* One pointing to the next node.
* One pointing to the previous node.  
  This allows traversal in both directions.

**(C) Circular Linked List**

* **Singly Circular Linked List**: The last node points back to the first node.
* **Doubly Circular Linked List**: The last node points to the first node, and the first node’s previous pointer points to the last node.

**3. Operations on Linked List**

1. **Insertion** (At the beginning, end, and middle)
2. **Deletion** (At the beginning, end, and middle)
3. **Searching for an element**
4. **Traversal**
5. **Reversing a linked list**
6. **Sorting a linked list**
7. **Detecting a loop in a linked list**
8. **Finding the middle element**

**4. Implementation of a Singly Linked List in Java**

**Node Structure**

java

CopyEdit

class Node {

int data; // Data stored in the node

Node next; // Pointer to the next node

public Node(int data) {

this.data = data;

this.next = null;

}

}

**Basic Linked List Class**

java

CopyEdit

class LinkedList {

Node head; // Head of the list (first node)

// Insert at the beginning

public void insertAtBeginning(int data) {

Node newNode = new Node(data);

newNode.next = head;

head = newNode;

}

// Insert at the end

public void insertAtEnd(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

return;

}

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

// Insert at a specific position

public void insertAtPosition(int data, int position) {

Node newNode = new Node(data);

if (position == 0) {

newNode.next = head;

head = newNode;

return;

}

Node temp = head;

for (int i = 0; temp != null && i < position - 1; i++) {

temp = temp.next;

}

if (temp == null) return;

newNode.next = temp.next;

temp.next = newNode;

}

// Delete from the beginning

public void deleteFromBeginning() {

if (head != null) {

head = head.next;

}

}

// Delete from the end

public void deleteFromEnd() {

if (head == null) return;

if (head.next == null) {

head = null;

return;

}

Node temp = head;

while (temp.next.next != null) {

temp = temp.next;

}

temp.next = null;

}

// Delete from a specific position

public void deleteAtPosition(int position) {

if (head == null) return;

if (position == 0) {

head = head.next;

return;

}

Node temp = head;

for (int i = 0; temp != null && i < position - 1; i++) {

temp = temp.next;

}

if (temp == null || temp.next == null) return;

temp.next = temp.next.next;

}

// Search for an element

public boolean search(int key) {

Node temp = head;

while (temp != null) {

if (temp.data == key) return true;

temp = temp.next;

}

return false;

}

// Display the linked list

public void display() {

Node temp = head;

while (temp != null) {

System.out.print(temp.data + " -> ");

temp = temp.next;

}

System.out.println("NULL");

}

}

**Usage of Linked List**

java

CopyEdit

public class LinkedListDemo {

public static void main(String[] args) {

LinkedList list = new LinkedList();

list.insertAtEnd(10);

list.insertAtEnd(20);

list.insertAtEnd(30);

list.insertAtBeginning(5);

list.insertAtPosition(15, 2);

System.out.println("Linked List:");

list.display();

list.deleteFromEnd();

System.out.println("After deleting last element:");

list.display();

list.deleteFromBeginning();

System.out.println("After deleting first element:");

list.display();

System.out.println("Searching for 20: " + list.search(20));

System.out.println("Searching for 100: " + list.search(100));

}

}

**Output**

rust

CopyEdit

Linked List:

5 -> 10 -> 15 -> 20 -> 30 -> NULL

After deleting last element:

5 -> 10 -> 15 -> 20 -> NULL

After deleting first element:

10 -> 15 -> 20 -> NULL

Searching for 20: true

Searching for 100: false

**5. Doubly Linked List Implementation**

In a doubly linked list, each node has two references:

1. **prev** → Points to the previous node.
2. **next** → Points to the next node.

java

CopyEdit

class DoublyNode {

int data;

DoublyNode prev, next;

public DoublyNode(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

Similarly, the linked list class will have methods for insertion, deletion, and traversal.

**6. Circular Linked List Implementation**

A circular linked list has its last node pointing back to the first node, forming a cycle.

java

CopyEdit

class CircularNode {

int data;

CircularNode next;

public CircularNode(int data) {

this.data = data;

this.next = this; // Circular reference

}

}

**7. Advanced Linked List Operations**

**(A) Reverse a Linked List**

java

CopyEdit

public void reverse() {

Node prev = null, current = head, next = null;

while (current != null) {

next = current.next;

current.next = prev;

prev = current;

current = next;

}

head = prev;

}

**(B) Detect Loop in a Linked List**

Using Floyd’s Cycle Detection Algorithm:

java

CopyEdit

public boolean detectLoop() {

Node slow = head, fast = head;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

if (slow == fast) return true;

}

return false;

}

Day 6 :

**What is a Stack?**

A **Stack** is a **linear data structure** that follows the **Last In, First Out (LIFO)** principle. This means that the last element added to the stack will be the first one to be removed. Imagine a **stack of plates**—the last plate placed on top is the first one to be removed.

**Key Characteristics of a Stack**

1. **LIFO Order**: The last inserted element is removed first.
2. **Restricted Access**: Elements can only be added or removed from the top.
3. **Operations take place in constant time O(1)**.

Stacks are used in:

* Function calls and recursion
* Expression evaluation
* Undo-redo operations in applications
* Backtracking algorithms (like solving mazes)
* Depth First Search (DFS)

**Stack Operations in Java**

Java provides multiple ways to implement stacks:

* Using the **Stack** class from java.util
* Using an **array-based** stack
* Using a **linked list-based** stack

**1. Push Operation (Insertion)**

* This operation adds an element to the top of the stack.
* If using an array and the stack is full, it results in **Stack Overflow**.

**Code Example in Java (Using Stack Class)**

java

CopyEdit

import java.util.Stack;

public class StackExample {

public static void main(String[] args) {

Stack<Integer> stack = new Stack<>();

stack.push(10);

stack.push(20);

stack.push(30);

System.out.println(stack); // Output: [10, 20, 30]

}

}

**2. Pop Operation (Deletion)**

* Removes and returns the top element from the stack.
* If the stack is empty, it results in **Stack Underflow**.

**Code Example**

java

CopyEdit

System.out.println(stack.pop()); // Output: 30

System.out.println(stack); // Output: [10, 20]

**3. isEmpty()**

* Checks whether the stack is empty.
* Returns true if empty, otherwise false.

**Code Example**

java

CopyEdit

System.out.println(stack.isEmpty()); // Output: false

**4. isFull()**

* In an **array-based stack**, we check if the stack is full.
* Not applicable for Java's built-in Stack class since it dynamically resizes.

java

CopyEdit

public boolean isFull() {

return top == capacity - 1;

}

**5. Peek Operation**

* Returns the **top element** of the stack without removing it.

**Code Example**

java

CopyEdit

System.out.println(stack.peek()); // Output: 20

**6. Count (Size of the Stack)**

* Returns the number of elements present in the stack.

**Code Example**

java

CopyEdit

System.out.println(stack.size()); // Output: 2

**7. Displaying the Stack**

* Prints the stack elements.

**Code Example**

java

CopyEdit

System.out.println(stack); // Output: [10, 20]

**Representation of Stack in Java**

**1. Array Representation of Stack**

* Uses an array to store elements.
* Fixed size, prone to **overflow**.

**Code Example**

java

CopyEdit

class StackArray {

int arr[];

int top;

int capacity;

StackArray(int size) {

arr = new int[size];

capacity = size;

top = -1;

}

void push(int x) {

if (top == capacity - 1) {

System.out.println("Stack Overflow");

return;

}

arr[++top] = x;

}

int pop() {

if (top == -1) {

System.out.println("Stack Underflow");

return -1;

}

return arr[top--];

}

}

**2. Linked List Representation of Stack**

* No fixed size.
* Uses **nodes** with data and next pointers.

**Code Example**

java

CopyEdit

class StackLinkedList {

class Node {

int data;

Node next;

}

Node top;

void push(int x) {

Node newNode = new Node();

newNode.data = x;

newNode.next = top;

top = newNode;

}

int pop() {

if (top == null) {

System.out.println("Stack Underflow");

return -1;

}

int x = top.data;

top = top.next;

return x;

}

}

**Time Complexity of Stack Operations**

| **Operation** | **Time Complexity** |
| --- | --- |
| Push | O(1) |
| Pop | O(1) |
| Peek | O(1) |
| isEmpty | O(1) |
| Size | O(1) |

**Applications of Stack**

1. **String Reversal**
   * Push each character into a stack and pop them back.
   * **Code Example**:

java

CopyEdit

public class ReverseString {

public static void main(String[] args) {

String str = "Ankita";

Stack<Character> stack = new Stack<>();

for (char ch : str.toCharArray()) {

stack.push(ch);

}

while (!stack.isEmpty()) {

System.out.print(stack.pop());

}

}

}

**Output**: atiknA

1. **Balancing Parentheses**
   * Uses a stack to track opening and closing brackets.
   * **Code Example**:

java

CopyEdit

boolean isBalanced(String expr) {

Stack<Character> stack = new Stack<>();

for (char ch : expr.toCharArray()) {

if (ch == '(') stack.push(ch);

else if (ch == ')') {

if (stack.isEmpty()) return false;

stack.pop();

}

}

return stack.isEmpty();

}

1. **Recursion**
   * Function calls are stored in the stack.
2. **Backtracking**
   * Used in puzzles like **Sudoku, N-Queens**.
3. **Depth-First Search (DFS)**
   * DFS for graphs uses a stack.
4. **Polish Notation (Postfix, Prefix, Infix)**
   * **Infix to Postfix** conversion:
     + A + B (Infix) → AB+ (Postfix)
5. **Expression Conversion**
   * Uses precedence rules:

vbnet

CopyEdit

Operators: +, -, \*, /

Precedence: \*, / > +, -

Associativity: Left to Right

* + Stack stores operators and operands for conversion.

**Day 7**

* **Doubly Linked List :-**

A **Doubly Linked List (DLL)** is a type of linked list in which each node contains three parts:

1. **Data** – The value stored in the node.
2. **Previous Pointer** – A pointer/reference to the previous node in the list.
3. **Next Pointer** – A pointer/reference to the next node in the list.

**Advantages of Doubly Linked List over Singly Linked List**

* Can be traversed in both forward and backward directions.
* Deletion of a node is more efficient since we have access to the previous node.
* Insertions and deletions are more flexible compared to singly linked lists.

**Operations on a Doubly Linked List**

1. **Insertion**
   * At the beginning
   * At the end
   * At a specific position
2. **Deletion**
   * From the beginning
   * From the end
   * A specific node
3. **Traversal**
   * Forward traversal
   * Backward traversal
4. **Search**
5. **Reverse the list**

**Java Implementation of Doubly Linked List**

Here's a complete Java program that implements all the operations on a doubly linked list:

java

CopyEdit

// Java program to implement Doubly Linked List with all operations

class DoublyLinkedList {

// Node structure for Doubly Linked List

class Node {

int data;

Node prev;

Node next;

public Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

private Node head, tail;

// Constructor

public DoublyLinkedList() {

this.head = null;

this.tail = null;

}

// Insert at the beginning

public void insertAtBeginning(int data) {

Node newNode = new Node(data);

if (head == null) {

head = tail = newNode;

} else {

newNode.next = head;

head.prev = newNode;

head = newNode;

}

System.out.println(data + " inserted at the beginning.");

}

// Insert at the end

public void insertAtEnd(int data) {

Node newNode = new Node(data);

if (tail == null) {

head = tail = newNode;

} else {

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

}

System.out.println(data + " inserted at the end.");

}

// Insert at a specific position

public void insertAtPosition(int data, int position) {

if (position <= 0) {

System.out.println("Invalid position!");

return;

}

Node newNode = new Node(data);

if (position == 1) {

insertAtBeginning(data);

return;

}

Node temp = head;

for (int i = 1; i < position - 1 && temp != null; i++) {

temp = temp.next;

}

if (temp == null) {

System.out.println("Position out of bounds.");

} else {

newNode.next = temp.next;

if (temp.next != null) {

temp.next.prev = newNode;

} else {

tail = newNode;

}

temp.next = newNode;

newNode.prev = temp;

System.out.println(data + " inserted at position " + position);

}

}

// Delete from the beginning

public void deleteFromBeginning() {

if (head == null) {

System.out.println("List is empty!");

return;

}

System.out.println(head.data + " deleted from the beginning.");

head = head.next;

if (head != null) {

head.prev = null;

} else {

tail = null;

}

}

// Delete from the end

public void deleteFromEnd() {

if (tail == null) {

System.out.println("List is empty!");

return;

}

System.out.println(tail.data + " deleted from the end.");

tail = tail.prev;

if (tail != null) {

tail.next = null;

} else {

head = null;

}

}

// Delete a specific node by value

public void deleteByValue(int value) {

if (head == null) {

System.out.println("List is empty!");

return;

}

Node temp = head;

while (temp != null && temp.data != value) {

temp = temp.next;

}

if (temp == null) {

System.out.println("Value not found in the list!");

return;

}

System.out.println(value + " deleted from the list.");

if (temp == head) {

deleteFromBeginning();

} else if (temp == tail) {

deleteFromEnd();

} else {

temp.prev.next = temp.next;

temp.next.prev = temp.prev;

}

}

// Forward Traversal

public void displayForward() {

if (head == null) {

System.out.println("List is empty!");

return;

}

Node temp = head;

System.out.print("Forward List: ");

while (temp != null) {

System.out.print(temp.data + " ⇄ ");

temp = temp.next;

}

System.out.println("null");

}

// Backward Traversal

public void displayBackward() {

if (tail == null) {

System.out.println("List is empty!");

return;

}

Node temp = tail;

System.out.print("Backward List: ");

while (temp != null) {

System.out.print(temp.data + " ⇄ ");

temp = temp.prev;

}

System.out.println("null");

}

// Search an element

public boolean search(int key) {

Node temp = head;

int position = 1;

while (temp != null) {

if (temp.data == key) {

System.out.println(key + " found at position " + position);

return true;

}

temp = temp.next;

position++;

}

System.out.println(key + " not found in the list.");

return false;

}

// Reverse the Doubly Linked List

public void reverse() {

if (head == null) {

System.out.println("List is empty!");

return;

}

Node current = head, temp = null;

while (current != null) {

temp = current.prev;

current.prev = current.next;

current.next = temp;

current = current.prev;

}

if (temp != null) {

head = temp.prev;

}

System.out.println("List reversed successfully.");

}

// Main method to test the doubly linked list

public static void main(String[] args) {

DoublyLinkedList dll = new DoublyLinkedList();

dll.insertAtEnd(10);

dll.insertAtEnd(20);

dll.insertAtEnd(30);

dll.insertAtBeginning(5);

dll.insertAtPosition(15, 3);

dll.displayForward();

dll.displayBackward();

dll.deleteFromBeginning();

dll.deleteFromEnd();

dll.deleteByValue(20);

dll.displayForward();

dll.search(30);

dll.reverse();

dll.displayForward();

}

}

**Explanation of Code**

1. **Class Node**
   * Defines the structure of each node with data, prev, and next pointers.
2. **Class DoublyLinkedList**
   * Manages the operations on the DLL.
3. **Operations Implemented**
   * **Insertion:** Beginning, End, Specific Position.
   * **Deletion:** Beginning, End, Specific Value.
   * **Traversal:** Forward & Backward.
   * **Search:** Finds an element and prints its position.
   * **Reverse:** Reverses the list efficiently.

* **Tree**

**Tree in Data Structure and Algorithm**

A **Tree** is a non-linear data structure that consists of nodes connected by edges. Unlike arrays, linked lists, stacks, and queues, which are linear data structures, trees represent a hierarchical structure, meaning each node can have multiple children but only one parent (except for the root node). Trees are widely used in applications like databases, file systems, and search algorithms.

**Structure of a Tree**

A tree is made up of **nodes** and **edges**:

* A **node** is a fundamental part of the tree that holds some data and references to child nodes.
* An **edge** is a connection between two nodes that defines the relationship (parent-child).
* The topmost node in a tree is called the **root** node.
* Every node (except the root) has a **parent node** and can have multiple **child nodes**.
* Nodes that do not have children are called **leaf nodes**.
* The **depth** of a node is the number of edges from the root to that node.
* The **height** of a node is the number of edges on the longest path from that node to a leaf.
* The **height of a tree** is the height of the root node.

A simple representation of a tree structure:

mathematica

CopyEdit

A <-- Root

/ \

B C

/ \ \

D E F <-- Leaf Nodes

In this tree:

* A is the root node.
* B and C are children of A.
* D and E are children of B, and F is a child of C.
* D, E, and F are leaf nodes because they do not have children.

**Definition of a Tree**

A **tree** is a collection of nodes where:

1. There is a special node called the **root** that acts as the starting point.
2. Every node (except the root) has exactly **one parent**.
3. There is a unique path from the root to every other node, ensuring there are no cycles.
4. The number of children a node has is determined by the type of tree (binary tree, n-ary tree, etc.).

**Tree Terminologies Explained**

**1. Root Node**

The topmost node in a tree is called the **root node**. It is the starting point of the tree, and there is only one root node in any tree.

**2. Parent and Child Nodes**

A node that has children is called a **parent node**, and the nodes that extend from it are called **child nodes**.

For example, in the tree below, A is the **parent** of B and C, while B and C are **children** of A.

css

CopyEdit

A

/ \

B C

**3. Leaf Nodes**

A **leaf node** is a node that does not have any children. It is the end point of a branch.

In this tree, D and E are **leaf nodes**:

mathematica

CopyEdit

A

/ \

B C

/ \

D E

**4. Siblings**

Nodes that share the same parent are called **siblings**.

For example, in the tree below, B and C are **siblings** because they have the same parent (A).

css

CopyEdit

A

/ \

B C

**5. Ancestor and Descendant**

* An **ancestor** of a node is any node that is above it in the tree.
* A **descendant** of a node is any node that is below it.

For example, in the tree below:

* A is an **ancestor** of D, E, and F.
* D and E are **descendants** of B.

mathematica

CopyEdit

A

/ \

B C

/ \ \

D E F

**6. Degree of a Node**

The **degree of a node** refers to the number of children it has.

* A node with no children has a degree of **0** (leaf node).
* A node with three children has a degree of **3**.

For example:

less

CopyEdit

A

/ | \

B C D

Here, the **degree of A is 3** because it has three children: B, C, and D.

**7. Degree of a Tree**

The **degree of a tree** is the maximum degree of any node in the tree.

**8. Depth of a Node**

The **depth** of a node is the number of edges from the root to that node.  
For example:

less

CopyEdit

A (depth = 0)

/ \

B C (depth = 1)

/

D (depth = 2)

* Depth of A = **0**
* Depth of B and C = **1**
* Depth of D = **2**

**9. Height of a Node**

The **height** of a node is the number of edges on the longest path from that node to a leaf.  
For example:

css

CopyEdit

A

/ \

B C

/

D

* Height of D = **0** (since it’s a leaf node).
* Height of B = **1** (one edge from B to D).
* Height of A = **2** (longest path is from A to D).

**10. Height of a Tree**

The **height of a tree** is the height of its root node. It is the maximum depth of any node in the tree.

**Properties of Trees**

**1. Number of Edges**

A tree with n nodes always has n-1 edges because each node (except the root) is connected by exactly one edge.

**2. Minimum and Maximum Nodes in a Binary Tree**

* In a **binary tree** (where each node has at most 2 children), a tree of height h has at least h+1 nodes and at most 2^(h+1) - 1 nodes.

**3. Relationship Between Nodes, Edges, and Levels**

* If n is the number of nodes, the number of edges is n-1.
* If h is the height, the maximum number of nodes in a **full binary tree** is 2^(h+1) - 1.

**4. Complete Binary Tree**

A **complete binary tree** is a binary tree in which every level is completely filled except possibly the last, which is filled from left to right.

**5. Balanced Tree**

A **balanced tree** ensures that the difference between the heights of the left and right subtrees of any node is at most 1. This keeps operations like searching and insertion efficient (O(log n)).

**Conclusion**

A tree is a crucial data structure used in various applications like searching, sorting, network routing, and hierarchical databases. Understanding its properties, terminology, and structure helps in implementing efficient algorithms for different real-world problems.

Would you like me to explain any specific type of tree in more detail, such as **Binary Trees, Binary Search Trees (BST), AVL Trees, or B-Trees**?

* **Binary Tree :**

**Binary Tree in Data Structure**

A **Binary Tree** is a type of hierarchical data structure where each node has at most **two children**. These children are referred to as:

* **Left Child**
* **Right Child**

Binary Trees are widely used in searching, sorting, and hierarchical data representation, such as expression trees, decision trees, and binary search trees.

**Structure of a Binary Tree**

A Binary Tree consists of **nodes** connected by **edges**. Each node contains:

1. **Data** – The value stored in the node.
2. **Left Child** – A pointer/reference to the left subtree.
3. **Right Child** – A pointer/reference to the right subtree.

Example of a simple Binary Tree:

mathematica

CopyEdit

A

/ \

B C

/ \ \

D E F

* A is the **root** node.
* B and C are **children** of A.
* D, E, and F are **leaf nodes** (since they have no children).

**Types of Binary Trees**

Binary Trees can be categorized into different types based on their structure and properties.

**1. Full Binary Tree**

A **Full Binary Tree** (also known as a **Proper Binary Tree**) is a binary tree in which:

* Every node has **either 0 or 2 children**.

Example:

mathematica

CopyEdit

A

/ \

B C

/ \ / \

D E F G

Here, each node has either **zero or two children**.

**2. Complete Binary Tree**

A **Complete Binary Tree** is a binary tree where:

* All levels are **completely filled**, except possibly the last level.
* The last level is filled **from left to right**.

Example:

mathematica

CopyEdit

A

/ \

B C

/ \ /

D E F

* All levels except the last are **completely filled**.
* The last level nodes (D, E, F) are filled from **left to right**.

**3. Perfect Binary Tree**

A **Perfect Binary Tree** is a binary tree in which:

* **All internal nodes have two children**.
* **All leaf nodes are at the same level**.

Example:

mathematica

CopyEdit

A

/ \

B C

/ \ / \

D E F G

* Every node has exactly **two children**.
* All **leaf nodes** are at the same level.

**4. Balanced Binary Tree**

A **Balanced Binary Tree** is a binary tree where:

* The difference between the **height of the left and right subtrees** of every node is **at most 1**.

Example:

mathematica

CopyEdit

A

/ \

B C

/ / \

D E F

The left and right subtree heights are **balanced**.

A special type of Balanced Binary Tree is an **AVL Tree**, where the height difference is strictly **≤ 1** for every node.

**5. Degenerate (Skewed) Binary Tree**

A **Degenerate Binary Tree** is a tree where:

* Each node has **only one child**.
* It behaves like a **linked list**.

Example (Right-Skewed Tree):

css

CopyEdit

A

\

B

\

C

\

D

Example (Left-Skewed Tree):

css

CopyEdit

A

/

B

/

C

/

D

In both cases, the tree becomes **unbalanced**.

**Binary Tree Terminologies**

**1. Root Node**

* The **topmost node** in the tree.
* The entry point of the tree.

**2. Parent and Child Nodes**

* A node that has a **sub-node** is called a **parent node**.
* The connected **sub-node** is the **child node**.

**3. Leaf Node**

* A node that has **no children**.

**4. Internal Node**

* A node that has **at least one child**.

**5. Degree of a Node**

* The **number of children** a node has.
* In a Binary Tree, a node's degree is either **0, 1, or 2**.

**6. Depth of a Node**

* The number of edges from the **root node** to that node.

**7. Height of a Node**

* The number of edges on the **longest path** from the node to a leaf node.

**8. Height of a Tree**

* The height of the **root node**.
* Maximum depth of any node in the tree.

**Properties of Binary Tree**

1. **Maximum Nodes in a Binary Tree**
   * A Binary Tree of height h can have at most **2^(h+1) - 1** nodes.
2. **Minimum Height of a Binary Tree**
   * The minimum height of a tree with n nodes is **log₂(n+1) - 1**.
3. **Number of Leaf Nodes in a Perfect Binary Tree**
   * A perfect binary tree of height h has **2^h** leaf nodes.
4. **Total Number of Nodes in a Perfect Binary Tree**
   * Total nodes in a **Perfect Binary Tree** = **2^(h+1) - 1**.
5. **In a Full Binary Tree**
   * If there are n leaf nodes, then total nodes = **2n - 1**.

**Binary Tree Traversal Methods**

Traversal means visiting **each node exactly once** in a specific order.

**1. Depth-First Search (DFS) Traversals**

DFS explores as deep as possible before backtracking.

* **Inorder (Left, Root, Right)**

scss

CopyEdit

void inorder(Node root) {

if (root == null) return;

inorder(root.left);

System.out.print(root.data + " ");

inorder(root.right);

}

* **Preorder (Root, Left, Right)**

scss

CopyEdit

void preorder(Node root) {

if (root == null) return;

System.out.print(root.data + " ");

preorder(root.left);

preorder(root.right);

}

* **Postorder (Left, Right, Root)**

scss

CopyEdit

void postorder(Node root) {

if (root == null) return;

postorder(root.left);

postorder(root.right);

System.out.print(root.data + " ");

}

**2. Breadth-First Search (BFS) / Level Order Traversal**

This method visits nodes **level by level**.

pgsql

CopyEdit

void levelOrder(Node root) {

if (root == null) return;

Queue<Node> queue = new LinkedList<>();

queue.add(root);

while (!queue.isEmpty()) {

Node temp = queue.poll();

System.out.print(temp.data + " ");

if (temp.left != null) queue.add(temp.left);

if (temp.right != null) queue.add(temp.right);

}

}

**Day : 8**

* **. Introduction to Binary Trees**

**🔹 What is a Binary Tree?**

A **Binary Tree** is a hierarchical data structure in which each node has **at most two children** (left and right). It follows a parent-child relationship, where:

* The **topmost node** is called the **root**.
* Each node contains **data** and **pointers** to its left and right child.
* Nodes without children are called **leaf nodes**.

**🔹 Why do we need Binary Trees?**

* **Faster searching** compared to lists/arrays (especially BST).
* Used in **databases, file systems, and AI decision trees**.
* Supports hierarchical relationships, like **family trees, organizational structures**, etc.

**📌 Basic Structure of a Binary Tree**

Each node in a Binary Tree contains:

1. **Data** (value stored in the node).
2. **Left child** (pointer/reference to the left subtree).
3. **Right child** (pointer/reference to the right subtree).

**🖼️ Diagram Representation:**

lua

CopyEdit

(10) <-- Root

/ \

(5) (15)

/ \ / \

(2) (7) (12) (20)

Each node \*\*can have 0, 1, or 2 children

4o

**You said:**

i told you i dont want pints for above i wan tin detail explainations of all each i mentioned

**ChatGPT said:**

Got it! I'll explain **Binary Trees and all related concepts in-depth** as if you are learning from scratch. I will take a **sequential, structured approach** with **detailed explanations, diagrams, and Java code** where necessary.

**🌳 Binary Tree: A Complete Beginner’s Guide**

Let's assume you have **never** heard about **Binary Trees** before. By the end of this explanation, you will **fully understand**:

1. **What a Binary Tree is**
2. **How it is structured**
3. **Its different types and properties**
4. **How to represent it in memory**
5. **Tree traversal methods**
6. **Special Binary Trees (BST, AVL)**
7. **Java Code for implementation**

Let’s **begin our journey into trees!** 🚀

**📌 1. What is a Binary Tree?**

A **Binary Tree** is a **hierarchical** data structure, which means it is organized in a **tree-like format**. Unlike arrays or linked lists (which are **linear structures**), a **tree grows in multiple directions**.

**🌱 Basic Definition**

A **Binary Tree** is a structure where:

* **Each node has at most TWO children**.
* The **first node** (topmost) is called the **root**.
* Each **node is connected** to its **left** and **right child**.

Think of a **family tree**:

markdown

CopyEdit

Grandparent (Root)

/ \

Parent1 Parent2

/ \ / \

Child1 Child2 Child3 Child4

Each **parent has children**, and each **child can be a parent** to other children.

**📌 2. Structure of a Binary Tree**

A Binary Tree consists of **nodes**, and each node contains:

1. **Data** (A value stored in the node)
2. **Left Child** (Pointer to the left node)
3. **Right Child** (Pointer to the right node)

**🖼️ Diagram Representation**

lua

CopyEdit

(10) <-- Root Node

/ \

(5) (15)

/ \ / \

(2) (7) (12) (20)

Here, **each node follows the rule** of having at most **two children**.

In **Java**, we can represent a node like this:

java

CopyEdit

class Node {

int data;

Node left, right;

public Node(int value) {

data = value;

left = right = null;

}

}

Here:

* **data** holds the value.
* **left** stores the reference to the left child.
* **right** stores the reference to the right child.

**📌 3. Properties of a Binary Tree**

A Binary Tree follows some **important rules**:

**🔹 1. Maximum Nodes at Level L**

At level **L**, the maximum number of nodes a tree can have is:

Max Nodes=2L\text{Max Nodes} = 2^LMax Nodes=2L

For example:

* Level 0 → 1 node.
* Level 1 → 2 nodes.
* Level 2 → 4 nodes.

**🔹 2. Maximum Nodes in a Tree of Height h**

The **maximum** number of nodes in a Binary Tree of height h is:

2h−12^h - 12h−1

For example:

* Height 3 → 2^3 - 1 = 7 nodes.

**🔹 3. Minimum Height for N Nodes**

To fit N nodes, the **minimum height** is:

h=log⁡2(N+1)h = \log\_2(N + 1)h=log2​(N+1)

**🔹 4. Full Binary Tree Condition**

A **Full Binary Tree** is one in which **every node has either 0 or 2 children**.

**📌 4. Types of Binary Trees**

There are several variations of Binary Trees:

**🔸 1. Full Binary Tree**

A **Full Binary Tree** is a tree where **every node has either 0 or 2 children** (no node has only one child).

**Example:**

markdown

CopyEdit

(1)

/ \

(2) (3)

/ \ / \

(4) (5) (6) (7)

Each node has either **two children or none**.

**🔸 2. Complete Binary Tree**

A **Complete Binary Tree** is a tree where **all levels are completely filled** except the last level, which is **filled from left to right**.

**Example:**

markdown

CopyEdit

(1)

/ \

(2) (3)

/ \ /

(4) (5) (6)

Here, node (6) is **filled from left to right**.

**🔸 3. Perfect Binary Tree**

A **Perfect Binary Tree** is a **Full Binary Tree where all leaves are at the same level**.

**Example:**

markdown

CopyEdit

(1)

/ \

(2) (3)

/ \ / \

(4) (5)(6) (7)

This is also a **Full Binary Tree**, but all leaves are **at the same level**.

**🔸 4. Degenerate (Skewed) Binary Tree**

A **Degenerate Tree** is a tree where **each parent has only one child**, making it look like a linked list.

**Example (Left Skewed Tree):**

markdown

CopyEdit

(1)

/

(2)

/

(3)

/

(4)

This is similar to a **linked list**.

**🔸 5. Strict Binary Tree**

A **Strict Binary Tree** is a tree where **every node has either 2 or 0 children**.

**📌 5. Representation of a Binary Tree**

A Binary Tree can be stored in two ways:

**🔹 1. Using Arrays**

We can store a **Complete Binary Tree** in an **array** by using **indexing**:

css

CopyEdit

(1) → A[0]

/ \

(2) (3) → A[1], A[2]

/ \ /

(4) (5) (6) → A[3], A[4], A[5]

**Formula for Array Representation:**

* Left child of i: **2 \* i + 1**
* Right child of i: **2 \* i + 2**
* Parent of i: **(i - 1) / 2**

**🔹 2. Using Linked Representation**

Each node contains:

1. **Data**
2. **Pointer to Left Child**
3. **Pointer to Right Child**

java

CopyEdit

class Node {

int data;

Node left, right;

Node(int value) {

data = value;

left = right = null;

}

}

**📌 6. Tree Traversal**

Traversing a tree means **visiting** all nodes.

**🔹 Depth First Search (DFS)**

1. **Inorder (Left → Root → Right)**
2. **Preorder (Root → Left → Right)**
3. **Postorder (Left → Right → Root)**

**🔹 Breadth First Search (BFS)**

* Also called **Level Order Traversal** (visit nodes level by level).

**📌 7. Binary Search Tree (BST)**

A **Binary Search Tree (BST)** is a **special Binary Tree** where:

* **Left subtree contains smaller values**.
* **Right subtree contains larger values**.

**Example:**

markdown

CopyEdit

(10)

/ \

(5) (15)

/ \ / \

(2) (7) (12) (20)

This **allows fast searching** (O(log N)).

**Operations in BST**

1. **Insertion** (Insert nodes while maintaining BST properties)
2. **Deletion** (Remove nodes correctly)
3. **Search** (Find an element)

**📌 8. Balanced Trees (AVL Tree)**

An **AVL Tree** is a **BST with a height balance condition**.

* The **difference in height** between left and right subtrees must be **at most 1**.

**Example:**

markdown

CopyEdit

(10)

/ \

(5) (15)

/ \ / \

(3) (7) (12) (20)

**🌳 1. Binary Tree Implementation in Java**

A general **Binary Tree** does **not** follow any ordering rule (unlike BST). We will use a **linked representation**.

**Operations Covered**

* **Insertion**
* **Deletion of a node**
* **Traversals** (Inorder, Preorder, Postorder, Level Order)
* **Search a node**

**Code: Binary Tree Implementation**

java

CopyEdit

import java.util.LinkedList;

import java.util.Queue;

class Node {

int data;

Node left, right;

public Node(int value) {

data = value;

left = right = null;

}

}

class BinaryTree {

Node root;

// Insert node (Level Order)

void insert(int value) {

Node newNode = new Node(value);

if (root == null) {

root = newNode;

return;

}

Queue<Node> queue = new LinkedList<>();

queue.add(root);

while (!queue.isEmpty()) {

Node temp = queue.poll();

if (temp.left == null) {

temp.left = newNode;

return;

} else queue.add(temp.left);

if (temp.right == null) {

temp.right = newNode;

return;

} else queue.add(temp.right);

}

}

// Inorder Traversal (Left, Root, Right)

void inorder(Node node) {

if (node == null) return;

inorder(node.left);

System.out.print(node.data + " ");

inorder(node.right);

}

// Preorder Traversal (Root, Left, Right)

void preorder(Node node) {

if (node == null) return;

System.out.print(node.data + " ");

preorder(node.left);

preorder(node.right);

}

// Postorder Traversal (Left, Right, Root)

void postorder(Node node) {

if (node == null) return;

postorder(node.left);

postorder(node.right);

System.out.print(node.data + " ");

}

// Level Order Traversal (BFS)

void levelOrder() {

if (root == null) return;

Queue<Node> queue = new LinkedList<>();

queue.add(root);

while (!queue.isEmpty()) {

Node temp = queue.poll();

System.out.print(temp.data + " ");

if (temp.left != null) queue.add(temp.left);

if (temp.right != null) queue.add(temp.right);

}

}

// Search in Binary Tree (Level Order)

boolean search(int key) {

if (root == null) return false;

Queue<Node> queue = new LinkedList<>();

queue.add(root);

while (!queue.isEmpty()) {

Node temp = queue.poll();

if (temp.data == key) return true;

if (temp.left != null) queue.add(temp.left);

if (temp.right != null) queue.add(temp.right);

}

return false;

}

public static void main(String[] args) {

BinaryTree tree = new BinaryTree();

tree.insert(10);

tree.insert(20);

tree.insert(30);

tree.insert(40);

tree.insert(50);

tree.insert(60);

tree.insert(70);

System.out.println("\nInorder Traversal:");

tree.inorder(tree.root);

System.out.println("\nPreorder Traversal:");

tree.preorder(tree.root);

System.out.println("\nPostorder Traversal:");

tree.postorder(tree.root);

System.out.println("\nLevel Order Traversal:");

tree.levelOrder();

System.out.println("\nSearch for 30: " + tree.search(30));

System.out.println("Search for 100: " + tree.search(100));

}

}

**✅ Output**

yaml

CopyEdit

Inorder Traversal:

40 20 50 10 60 30 70

Preorder Traversal:

10 20 40 50 30 60 70

Postorder Traversal:

40 50 20 60 70 30 10

Level Order Traversal:

10 20 30 40 50 60 70

Search for 30: true

Search for 100: false

**🌳 2. Binary Search Tree (BST) Implementation in Java**

A **Binary Search Tree (BST)** is a special Binary Tree where:

* **Left subtree** contains values **smaller** than the root.
* **Right subtree** contains values **greater** than the root.

**Operations Covered**

* **Insertion**
* **Deletion**
* **Search**
* **Traversals (Inorder, Preorder, Postorder)**

**Code: Binary Search Tree Implementation**

java

CopyEdit

class BST {

class Node {

int data;

Node left, right;

public Node(int value) {

data = value;

left = right = null;

}

}

Node root;

// Insert into BST

void insert(int value) {

root = insertRec(root, value);

}

Node insertRec(Node root, int value) {

if (root == null) {

root = new Node(value);

return root;

}

if (value < root.data)

root.left = insertRec(root.left, value);

else if (value > root.data)

root.right = insertRec(root.right, value);

return root;

}

// Inorder Traversal (Sorted Order)

void inorder(Node root) {

if (root == null) return;

inorder(root.left);

System.out.print(root.data + " ");

inorder(root.right);

}

// Search in BST

boolean search(int key) {

return searchRec(root, key) != null;

}

Node searchRec(Node root, int key) {

if (root == null || root.data == key) return root;

if (key < root.data) return searchRec(root.left, key);

return searchRec(root.right, key);

}

// Delete node from BST

void delete(int key) {

root = deleteRec(root, key);

}

Node deleteRec(Node root, int key) {

if (root == null) return root;

if (key < root.data)

root.left = deleteRec(root.left, key);

else if (key > root.data)

root.right = deleteRec(root.right, key);

else {

// Case 1: No child

if (root.left == null && root.right == null)

return null;

// Case 2: One child

if (root.left == null) return root.right;

if (root.right == null) return root.left;

// Case 3: Two children

root.data = minValue(root.right);

root.right = deleteRec(root.right, root.data);

}

return root;

}

int minValue(Node root) {

int min = root.data;

while (root.left != null) {

min = root.left.data;

root = root.left;

}

return min;

}

public static void main(String[] args) {

BST tree = new BST();

tree.insert(50);

tree.insert(30);

tree.insert(70);

tree.insert(20);

tree.insert(40);

tree.insert(60);

tree.insert(80);

System.out.println("\nInorder Traversal:");

tree.inorder(tree.root);

System.out.println("\nSearch 40: " + tree.search(40));

System.out.println("Search 100: " + tree.search(100));

tree.delete(30);

System.out.println("\nInorder After Deletion:");

tree.inorder(tree.root);

}

}

**✅ Output**

yaml

CopyEdit

Inorder Traversal:

20 30 40 50 60 70 80

Search 40: true

Search 100: false

Inorder After Deletion:

20 40 50 60 70 80

**Day 9**

* A **tree** is a special type of data structure that looks like an upside-down tree. It is used to store hierarchical data.

**Basic Structure of a Tree**

* A **tree** consists of **nodes** connected by **edges**.
* The **topmost node** is called the **root**.
* Each node can have child nodes, except the **leaf nodes**, which have no children.

**Example of a Simple Tree**

mathematica

CopyEdit

A <-- Root

/ \

B C

/ \ \

D E F <-- Leaf Nodes

* A is the **root node**.
* B and C are **children of A**.
* D, E, and F are **leaf nodes** (no further children).

**Where Are Trees Used?**

* **File System** (Folders within folders)
* **Organization Chart** (CEO → Managers → Employees)
* **Social Networks** (Friendship relationships)
* **AI Decision Making** (Chess moves, Google Maps pathfinding)

**2. Internal and External Sorting**

Sorting means arranging data in **ascending or descending order**.

**Internal Sorting**

* Sorting is done **completely in RAM (memory)**.
* Used when the data fits in the system's **main memory**.
* Faster because everything happens in memory.
* Examples:
  + **Bubble Sort**
  + **Merge Sort**
  + **Quick Sort**

**External Sorting**

* Used when the data is **too large to fit in RAM**.
* Data is divided into **chunks** and stored on **disk storage**.
* Sorting is done in parts, then merged.
* Example:
  + **Merge Sort on Large Files (used in databases, cloud storage, etc.)**

**Difference Between Internal and External Sorting**

| **Feature** | **Internal Sorting** | **External Sorting** |
| --- | --- | --- |
| Where it happens | In **RAM** (Memory) | Uses **Disk & RAM** |
| Speed | Fast | Slow (because disk is slower than RAM) |
| Data size | Small to medium datasets | Large datasets (TBs of data) |
| Example Methods | Quick Sort, Merge Sort | External Merge Sort |

**3. Stable and Unstable Sorting**

Sorting algorithms can be **stable** or **unstable**, depending on how they handle **duplicate values**.

**Stable Sorting**

A **stable sorting algorithm** maintains the **relative order of duplicate elements**.

**Example:**

Imagine sorting these students by **marks**:

mathematica

CopyEdit

Before Sorting:

[ (A, 80), (B, 75), (C, 80), (D, 90) ]

If we sort **stably by marks**, A and C (both 80 marks) will remain **in the same order**.

mathematica

CopyEdit

After Stable Sorting:

[ (B, 75), (A, 80), (C, 80), (D, 90) ]

**Unstable Sorting**

An **unstable sorting algorithm** **does not** guarantee the order of duplicate elements.

mathematica

CopyEdit

After Unstable Sorting:

[ (B, 75), (C, 80), (A, 80), (D, 90) ]

Here, A and C swapped places!

**Why Does Stability Matter?**

* **Stable Sorting** is used when sorting **must** preserve the order of duplicate elements (e.g., sorting students by **name** first, then by **marks**).
* **Unstable Sorting** is okay when order does not matter.

**Examples of Sorting Algorithms**

| **Sorting Algorithm** | **Stable?** |
| --- | --- |
| Bubble Sort | ✅ Yes |
| Merge Sort | ✅ Yes |
| Quick Sort | ❌ No |
| Heap Sort | ❌ No |

**4. Visual Representation of Sorting**

Here’s a simple example of **Merge Sort** (which is stable):

**Before Sorting:**

csharp

CopyEdit

[38, 27, 43, 3, 9, 82, 10]

**Splitting Phase (Dividing into sub-arrays)**

css

CopyEdit

[38, 27, 43, 3] [9, 82, 10]

[38, 27] [43, 3] [9, 82] [10]

[38] [27] [43] [3] [9] [82] [10]

**Merging Phase**

css

CopyEdit

[27, 38] [3, 43] [9, 82] [10]

[3, 27, 38, 43] [9, 10, 82]

[3, 9, 10, 27, 38, 43, 82] <-- Sorted List!

This is how **merge sort works internally**.

* **Sorting Algorithms - Detailed Explanation**

**1. Introduction to Sorting**

Sorting is the process of arranging elements in a specific order (ascending or descending). Sorting is crucial in computer science for searching, optimization, and data organization.

**2. Bubble Sort**

**Explanation:**

Bubble Sort repeatedly swaps adjacent elements if they are in the wrong order. The largest element bubbles up to the end of the list.

**Node Representation:**

Initial Array: [5, 3, 8, 4, 2]

5 - 3 - 8 - 4 - 2

↓ ↓

3 - 5 - 8 - 4 - 2

↓ ↓

3 - 5 - 8 - 4 - 2

↓ ↓

3 - 5 - 4 - 8 - 2

↓ ↓

3 - 5 - 4 - 2 - 8

**Java Code:**

class BubbleSort {

void bubbleSort(int arr[]) {

int n = arr.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

void printArray(int arr[]) {

for (int value : arr) {

System.out.print(value + " ");

}

System.out.println();

}

public static void main(String args[]) {

BubbleSort obj = new BubbleSort();

int arr[] = {5, 3, 8, 4, 2};

obj.bubbleSort(arr);

obj.printArray(arr);

}

}

**Output:**

2 3 4 5 8

**3. Selection Sort**

**Explanation:**

Selection Sort finds the minimum element and swaps it with the first unsorted element.

**Node Representation:**

Initial Array: [5, 3, 8, 4, 2]

[2] - 3 - 8 - 4 - 5

2 - [3] - 8 - 4 - 5

2 - 3 - [4] - 8 - 5

2 - 3 - 4 - [5] - 8

**Java Code:**

class SelectionSort {

void selectionSort(int arr[]) {

int n = arr.length;

for (int i = 0; i < n - 1; i++) {

int minIdx = i;

for (int j = i + 1; j < n; j++) {

if (arr[j] < arr[minIdx]) {

minIdx = j;

}

}

int temp = arr[minIdx];

arr[minIdx] = arr[i];

arr[i] = temp;

}

}

void printArray(int arr[]) {

for (int value : arr) {

System.out.print(value + " ");

}

System.out.println();

}

public static void main(String args[]) {

SelectionSort obj = new SelectionSort();

int arr[] = {5, 3, 8, 4, 2};

obj.selectionSort(arr);

obj.printArray(arr);

}

}

**Output:**

2 3 4 5 8

**4. Insertion Sort**

**Explanation:**

Insertion Sort builds the sorted array one element at a time by inserting elements into their correct positions.

**Java Code:**

class InsertionSort {

void insertionSort(int arr[]) {

int n = arr.length;

for (int i = 1; i < n; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

void printArray(int arr[]) {

for (int value : arr) {

System.out.print(value + " ");

}

System.out.println();

}

public static void main(String args[]) {

InsertionSort obj = new InsertionSort();

int arr[] = {5, 3, 8, 4, 2};

obj.insertionSort(arr);

obj.printArray(arr);

}

}

**Output:**

2 3 4 5 8

**5. Merge Sort**

**Explanation:**

Merge Sort follows the Divide and Conquer approach:

1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the sorted halves.

**Java Code:**

class MergeSort {

void merge(int arr[], int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

int L[] = new int[n1];

int R[] = new int[n2];

for (int i = 0; i < n1; ++i)

L[i] = arr[l + i];

for (int j = 0; j < n2; ++j)

R[j] = arr[m + 1 + j];

int i = 0, j = 0, k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void sort(int arr[], int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

sort(arr, l, m);

sort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

}

**Output:**

2 3 4 5 8

* **Heap Sort – Detailed Explanation**

Heap Sort is a **comparison-based sorting algorithm** that uses a **binary heap data structure** to sort elements efficiently. It is based on the **Heap Data Structure**, where elements are arranged in **Max Heap** or **Min Heap** order.

**1. Understanding Heaps**

A **heap** is a **complete binary tree** (all levels except possibly the last are fully filled, and nodes are as left as possible) that satisfies the **heap property**:

* **Max Heap Property**: The parent node is always **greater than or equal** to its children.
* **Min Heap Property**: The parent node is always **smaller than or equal** to its children.

**1.1 Types of Heaps**

1. **Max Heap**: The largest element is at the root.
2. **Min Heap**: The smallest element is at the root.

**2. Heap Representation (Node Representation)**

Consider the following array:  
[4, 10, 3, 5, 1]

**Max Heap Representation:**

markdown

CopyEdit

10

/ \

5 3

/ \

4 1

**Heap Property**:

* 10 is the largest and is at the root.
* Every parent node is greater than its children.

**Min Heap Representation:**

markdown

CopyEdit

1

/ \

4 3

/ \

10 5

**Heap Property**:

* 1 is the smallest and is at the root.
* Every parent node is smaller than its children.

**3. Steps in Heap Sort**

1. **Build a Max Heap** (for sorting in ascending order).
2. **Extract the maximum element (root node)** and place it at the end of the array.
3. **Heapify** the remaining elements to restore the heap property.
4. **Repeat until all elements are sorted.**

**4. Heap Sort Algorithm Explanation (Step-by-Step with Node Representation)**

**Example: Sort [4, 10, 3, 5, 1] using Heap Sort**

**Step 1: Build Max Heap**

Convert the array into a **Max Heap**:

markdown

CopyEdit

10

/ \

5 3

/ \

4 1

Max Heap is created. Now we perform heap sort.

**Step 2: Swap Root with Last Element**

Swap **10** (root) with the last element **1**:

markdown

CopyEdit

1

/ \

5 3

/ \

4 10 (swapped)

Now, **10 is sorted** at the last index.

**Step 3: Heapify the Remaining Elements**

After removing **10**, **heapify** the remaining elements:

markdown

CopyEdit

5

/ \

1 3

/

4

Heap property is restored.

**Step 4: Swap Root with Last Element Again**

Swap **5** with **4**:

markdown

CopyEdit

4

/ \

1 3

Now, **5 is sorted** at the second-last index.

**Step 5: Heapify Again**

After removing **5**, heapify the remaining elements:

markdown

CopyEdit

4

/ \

1 3

No changes are required.

**Step 6: Continue Swapping Until Sorted**

* Swap **4** with **3** → Heapify.
* Swap **3** with **1** → Heapify.
* **Sorted Array: [1, 3, 4, 5, 10]**

**5. Java Code for Heap Sort (Max Heap)**

java

CopyEdit

import java.util.Arrays;

public class HeapSort {

public void heapSort(int arr[]) {

int n = arr.length;

// Step 1: Build Max Heap

for (int i = n / 2 - 1; i >= 0; i--) {

heapify(arr, n, i);

}

// Step 2: Extract elements from heap one by one

for (int i = n - 1; i > 0; i--) {

// Move current root to end

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

// Heapify the reduced heap

heapify(arr, i, 0);

}

}

void heapify(int arr[], int n, int i) {

int largest = i; // Root node

int left = 2 \* i + 1; // Left child

int right = 2 \* i + 2; // Right child

// If left child is larger

if (left < n && arr[left] > arr[largest]) {

largest = left;

}

// If right child is larger

if (right < n && arr[right] > arr[largest]) {

largest = right;

}

// If largest is not root

if (largest != i) {

int swap = arr[i];

arr[i] = arr[largest];

arr[largest] = swap;

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

public static void main(String args[]) {

int arr[] = {4, 10, 3, 5, 1};

HeapSort hs = new HeapSort();

hs.heapSort(arr);

System.out.println("Sorted array: " + Arrays.toString(arr));

}

}

**6. Time and Space Complexity Analysis**

| **Operation** | **Time Complexity** |
| --- | --- |
| Building Heap | **O(n)** |
| Heapify (per node) | **O(log n)** |
| Extracting Elements | **O(n log n)** |
| **Overall Complexity** | **O(n log n)** |

* **Space Complexity**: **O(1)** (in-place sorting)

**7. Key Takeaways**

* **Heap Sort uses a Binary Heap** (Max Heap for ascending order, Min Heap for descending order).
* **It is not a stable sort** (relative order of equal elements is not preserved).
* **Time Complexity is O(n log n)**, making it more efficient than Bubble Sort or Insertion Sort.
* **No additional space required**, making it **memory-efficient**.

**Day 10:**

**🔶 Part 1: Quick Sort – In-Depth Explanation**

**🔸 What is Quick Sort?**

Quick Sort is a **recursive** sorting algorithm based on the **Divide and Conquer** approach. It is one of the most efficient sorting algorithms for large datasets. The main idea behind Quick Sort is to **divide the array** into smaller sub-arrays using a **pivot** element, then **rearrange** the elements so that values **less than the pivot come before it**, and values **greater come after it**. This process is called **partitioning**.

**🔸 Understanding with a Real-Life Analogy**

Imagine you are organizing books on a shelf. You pick one book (say, by thickness), and then:

* Move **thinner books** to the **left** of it.
* Move **thicker books** to the **right**.
* You repeat the same process on the left and right piles, until each group has only one book or is sorted.

**🔸 Step-by-Step Working of Quick Sort**

Let’s consider this array for demonstration:

java

CopyEdit

[6, 3, 9, 5, 2, 8]

We’ll walk through **every step**.

**🔹 Step 1: Choose a Pivot**

Pick the **last element** as the pivot.  
So, **pivot = 8**

**🔹 Step 2: Partition the Array**

Now, rearrange the array so:

* Elements **less than pivot (8)** are moved to the **left**.
* Elements **greater than pivot (8)** are moved to the **right**.

We keep a pointer i that tracks the last position of the smaller element.

Let’s go through one by one:

* 6 < 8 → increment i, swap 6 with itself
* 3 < 8 → increment i, swap 3 with itself
* 9 > 8 → do nothing
* 5 < 8 → increment i, swap with 9
* 2 < 8 → increment i, swap with 9 again

Now swap 8 with the element after i. The array becomes:

java

CopyEdit

[6, 3, 5, 2, 8, 9]

Now:

* Left side: [6, 3, 5, 2] (less than 8)
* Right side: [9] (greater than 8)
* Pivot 8 is in correct position

**🔹 Step 3: Recursively Apply on Subarrays**

Repeat the process for left [6, 3, 5, 2]:

Pivot = 2

Now partition:

* All other elements are greater than 2, so we move 2 to the front:

java

CopyEdit

[2, 3, 5, 6]

Continue recursively on [3, 5, 6]:

* Pivot = 6, everything before is less ⇒ sorted

Final sorted array:

java

CopyEdit

[2, 3, 5, 6, 8, 9]

**🔸 Visualization as a Tree (Divide and Conquer Tree)**

This shows how the array is divided recursively:

csharp

CopyEdit

[6, 3, 9, 5, 2, 8]

↓ pivot=8

┌───────────────┴───────────────┐

[6, 3, 5, 2] [9]

↓ pivot=2

┌──────────┴──────────┐

[] [6, 3, 5]

↓ pivot=5

┌────────┴────────┐

[3] [6]

**🔸 Java Code – Full Explanation**

java

CopyEdit

public class QuickSortExample {

static void quickSort(int[] arr, int low, int high) {

if (low < high) {

// Partition the array

int pivotIndex = partition(arr, low, high);

// Sort elements before pivot and after pivot

quickSort(arr, low, pivotIndex - 1); // left subarray

quickSort(arr, pivotIndex + 1, high); // right subarray

}

}

static int partition(int[] arr, int low, int high) {

int pivot = arr[high]; // Choosing last element as pivot

int i = low - 1; // Index of smaller element

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

// Swap arr[i] and arr[j]

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Swap the pivot element to the correct position

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return i + 1; // Return pivot index

}

public static void main(String[] args) {

int[] arr = {6, 3, 9, 5, 2, 8};

quickSort(arr, 0, arr.length - 1);

for (int i : arr) {

System.out.print(i + " ");

}

}

}

**🔶 Part 2: Queue – In-Depth Explanation**

**🔸 What is a Queue?**

A **Queue** is a **linear data structure** that follows **FIFO** (First In, First Out). This means the element that is inserted first will be removed first.

Think of a **queue of people** at a bus stop:

* The first person to arrive is the first to get on the bus.
* The last one will wait until others move.

**🔸 Real-Life Analogy**

* **Ticket Counter:** People line up to buy tickets. The one who comes **first** gets the ticket **first**.
* **Printer Queue:** Files sent to the printer are printed **in the order** they were sent.

**🔸 Operations in a Queue**

1. **Enqueue**: Add an element at the **rear (end)** of the queue.
2. **Dequeue**: Remove an element from the **front (beginning)** of the queue.
3. **Peek / Front**: View the front element without removing it.
4. **isEmpty**: Check if the queue has no elements.

**🔸 Ways to Implement a Queue**

**✅ Using Arrays**

This is the basic way but has a fixed size.

**✅ Using Linked List**

This is dynamic and doesn’t require resizing.

**✅ Using Java’s Built-in Queue Interface**

You can use Queue<Integer> q = new LinkedList<>(); from Java Collections.

**🔸 Node Representation (Linked List)**

Here’s how a queue looks internally when using a linked list:

rust

CopyEdit

Front -> [10] -> [20] -> [30] -> null ← Rear

* 10 is at the front (will be removed first).
* 30 is the last added item (rear).

**🔸 Java Code – Queue Using Linked List (Manual Implementation)**

java

CopyEdit

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

class Queue {

Node front, rear;

// Enqueue: Add at rear

void enqueue(int value) {

Node newNode = new Node(value);

if (rear == null) {

front = rear = newNode;

return;

}

rear.next = newNode;

rear = newNode;

}

// Dequeue: Remove from front

int dequeue() {

if (front == null) {

System.out.println("Queue is empty");

return -1;

}

int result = front.data;

front = front.next;

if (front == null)

rear = null;

return result;

}

// Display queue

void display() {

Node temp = front;

while (temp != null) {

System.out.print(temp.data + " → ");

temp = temp.next;

}

System.out.println("null");

}

}

public class QueueDemo {

public static void main(String[] args) {

Queue q = new Queue();

q.enqueue(10);

q.enqueue(20);

q.enqueue(30);

q.display(); // Output: 10 → 20 → 30 → null

q.dequeue(); // Removes 10

q.display(); // Output: 20 → 30 → null

}

}

**🔸 Java Code – Queue Using Java Collections (Easier Way)**

java

CopyEdit

import java.util.\*;

public class BuiltInQueue {

public static void main(String[] args) {

Queue<Integer> q = new LinkedList<>();

q.add(10); // enqueue

q.add(20);

q.add(30);

System.out.println("Queue: " + q);

q.remove(); // dequeue

System.out.println("After Dequeue: " + q);

}

}

**1. Simple (Linear) Queue**

**🔸 Concept**

A **simple queue** is the most basic form of queue that follows the **FIFO** principle.

* Elements are inserted from the **rear**.
* Elements are deleted from the **front**.

**🔸 Internal Behavior (using Array)**

If we declare a queue of size 5:

text

CopyEdit

[10][20][30][-1][-1]

^ ^

Front Rear

After 2 dequeues:

text

CopyEdit

[-1][-1][30][-1][-1]

^ ^

Front Rear

Even though we have empty spaces, we **can't go back** — this is the main limitation of simple queues.

**🔸 Java Example: Simple Queue Using Array**

java

CopyEdit

class SimpleQueue {

int size = 5;

int[] queue = new int[size];

int front = 0, rear = -1;

void enqueue(int value) {

if (rear == size - 1) {

System.out.println("Queue is full");

return;

}

queue[++rear] = value;

}

void dequeue() {

if (front > rear) {

System.out.println("Queue is empty");

return;

}

System.out.println("Dequeued: " + queue[front++]);

}

void display() {

if (front > rear) {

System.out.println("Queue is empty");

return;

}

System.out.print("Queue: ");

for (int i = front; i <= rear; i++) {

System.out.print(queue[i] + " ");

}

System.out.println();

}

}

**🔶 2. Circular Queue**

**🔸 Concept**

A **circular queue** connects the rear back to the front, making the queue behave like a **circle**.

👉 It solves the **wastage of space** problem in a simple queue.

When the rear reaches the end of the array, it wraps around to the front if space is available.

**🔸 Internal Behavior**

Initial State (size = 5):

text

CopyEdit

[ ][ ][ ][ ][ ]

↑ ↑

F R

After enqueue 5 elements:

text

CopyEdit

[10][20][30][40][50]

↑ ↑

F R

After 2 dequeues:

text

CopyEdit

[-1][-1][30][40][50]

↑ ↑

F R

Now insert 2 more (wrap-around):

text

CopyEdit

[60][70][-1][40][50]

↑ ↑

R F

**🔸 Java Example: Circular Queue Using Array**

java

CopyEdit

class CircularQueue {

int size = 5;

int[] queue = new int[size];

int front = -1, rear = -1;

boolean isFull() {

return (front == (rear + 1) % size);

}

boolean isEmpty() {

return (front == -1);

}

void enqueue(int value) {

if (isFull()) {

System.out.println("Queue is full");

return;

}

if (isEmpty()) {

front = rear = 0;

} else {

rear = (rear + 1) % size;

}

queue[rear] = value;

}

void dequeue() {

if (isEmpty()) {

System.out.println("Queue is empty");

return;

}

System.out.println("Dequeued: " + queue[front]);

if (front == rear) { // only one element

front = rear = -1;

} else {

front = (front + 1) % size;

}

}

void display() {

if (isEmpty()) {

System.out.println("Queue is empty");

return;

}

System.out.print("Queue: ");

int i = front;

while (true) {

System.out.print(queue[i] + " ");

if (i == rear)

break;

i = (i + 1) % size;

}

System.out.println();

}

}

**🔶 3. Priority Queue**

**🔸 Concept**

In a **priority queue**, elements are ordered based on **priority** (not FIFO). The element with the **highest priority is served first**, not the one that came first.

If two elements have the same priority, then FIFO is used.

**🔸 Real-Life Example**

* In a hospital, a patient with a **serious condition** is treated before others, no matter when they arrived.
* Operating system processes are handled based on priority.

**🔸 Java Example: Priority Queue Using Java Collection**

Java provides a built-in class PriorityQueue (by default, it’s a **Min Heap**):

java

CopyEdit

import java.util.PriorityQueue;

public class PriorityQueueExample {

public static void main(String[] args) {

PriorityQueue<Integer> pq = new PriorityQueue<>();

pq.add(30);

pq.add(10);

pq.add(20);

while (!pq.isEmpty()) {

System.out.println("Dequeued: " + pq.poll()); // smallest first

}

}

}

If you want **highest value first**, use a **max-heap** with comparator:

java

CopyEdit

PriorityQueue<Integer> maxPQ = new PriorityQueue<>((a, b) -> b - a);

**🔶 4. Deque (Double-Ended Queue)**

**🔸 Concept**

**Deque** stands for **Double-Ended Queue**. It allows **insertion and deletion from both front and rear ends**.

This makes it more flexible than a simple or circular queue.

**🔸 Use Cases**

* Useful in **palindrome checking**, **sliding window problems**, and **browser history (forward and back)**.
* Can act as **stack** or **queue** depending on usage.

**🔸 Java Example: Deque Using Java Collections**

java

CopyEdit

import java.util.\*;

public class DequeExample {

public static void main(String[] args) {

Deque<Integer> dq = new ArrayDeque<>();

// Add at both ends

dq.addFirst(10); // front

dq.addLast(20); // rear

dq.addFirst(5);

dq.addLast(25);

System.out.println("Deque: " + dq);

// Remove from both ends

dq.removeFirst(); // removes 5

dq.removeLast(); // removes 25

System.out.println("After Deletion: " + dq);

}

}

**🧠 Summary Table**

| **Type** | **Insert At** | **Remove From** | **Special Feature** |
| --- | --- | --- | --- |
| Simple Queue | Rear | Front | FIFO |
| Circular Queue | Rear (Circular) | Front (Circular) | Reuses space using circular movement |
| Priority Queue | Based on priority | Based on priority | High-priority served first |
| Deque | Both ends | Both ends | Supports Stack and Queue operations |

**🔶 What is Hashing?**

**🔸 Concept:**

**Hashing** is a technique used to **uniquely identify and access data** using a **hash function**. It transforms **a key into an index** (called the **hash code**) in a data structure called a **hash table**.

🔑 **Key Idea:**  
Instead of searching for data linearly, we use a **hash function** to directly compute where the data should be placed or searched — making operations like insertion, deletion, and search very fast (average **O(1)** time).

**🔸 Hashing in Real Life:**

Imagine a **library** where instead of walking through every book to find a title, there’s a special formula (hash function) that tells you the exact shelf number for a book — super efficient!

**🔸 Hash Table Structure:**

A **hash table** is usually an array of fixed size. The hash function maps keys to **indices** in this array.

text

CopyEdit

Hash Table (size = 10)

Index: 0 1 2 3 4 5 6 7 8 9

- - 12 - 22 - - 32 - -

If we use hash function key % 10, the keys:

* 12 → index 2
* 22 → index 2 (collision!)
* 32 → index 2 (collision again!)

So collision handling becomes important.

**🧠 Applications of Hashing**

1. **Databases:** Quick data retrieval using keys.
2. **Compilers:** Hashing variable/function names.
3. **Cryptography:** Secure storage of passwords.
4. **Caching:** Store web pages or results of expensive computations.
5. **Blockchain:** Blocks are identified by hash values.
6. **Load Balancing:** Hashing decides which server will handle a request.
7. **Symbol Tables in Programming Languages**

**🔷 Hashing Techniques (Hash Functions)**

Let’s now go through different types of **hash functions** with detailed explanation and examples:

**1. ✅ Direct Hashing (Identity Method)**

**🔸 Concept:**

The key itself is used as the address/index.

java

CopyEdit

index = key;

**🔸 Example:**

Key = 100 → stored at index 100

**🔸 When to use:**

* When keys are small integers and space isn’t a constraint.
* Not memory efficient for large keys.

**2. ✅ Division Method**

**🔸 Concept:**

Take the **remainder of key divided by table size (a prime number)**.

java

CopyEdit

index = key % table\_size;

**🔸 Why prime table size?**

To reduce patterns and distribute keys evenly.

**🔸 Example:**

If table\_size = 7 and key = 10 → 10 % 7 = 3 → store at index 3

**3. ✅ Multiplicative Method**

**🔸 Concept:**

Multiply key with a **fractional constant**, extract decimal part, then multiply by table size.

java

CopyEdit

index = floor(table\_size × ((key × A) mod 1))

Where:

* A is a constant (0 < A < 1), often ≈ **0.618033** (inverse of golden ratio).

**🔸 Example:**

Let key = 1234, table\_size = 10, A = 0.618033  
(1234 × 0.618033) = 763.447  
Take decimal: 0.447 × 10 = 4.47 → floor(4.47) = 4  
So store at index 4

✅ Very efficient and widely used in low-level implementations.

**4. ✅ Folding Method**

**🔸 Concept:**

* Break the key into equal parts.
* Add them together.
* Apply modulo if needed.

**🔸 Example:**

Key = 123456, break into 12, 34, 56 → sum = 12 + 34 + 56 = 102  
If table size is 10 → 102 % 10 = 2

🔁 If key length is not evenly divisible, reverse middle or alternate parts before summing.

**5. ✅ Mid-Square Method**

**🔸 Concept:**

* Square the key.
* Take **middle digits** from the result to form the index.

**🔸 Example:**

Key = 123 → 123² = **15129**  
Middle digits = **51** → Use this as index (or % table\_size)

This method gives good distribution but might need post-processing if the table size is small.

**🔶 What is Collision?**

**🔸 Concept:**

A **collision** occurs when **two keys hash to the same index**.

Example:

text

CopyEdit

key1 = 12 → index = 2

key2 = 22 → index = 2

Both want the same spot → collision!

**🔷 Collision Handling Techniques**

**1. ✅ Open Addressing**

All elements stored **within the hash table itself**. If a position is occupied, search for the next available slot.

**📌 Types:**

**a) Linear Probing**

If collision at index, check index + 1, then index + 2, etc.

java

CopyEdit

index = (hash(key) + i) % table\_size

🔁 Problem: Clustering (many elements in a row).

**b) Quadratic Probing**

Instead of linear step, use quadratic jumps.

java

CopyEdit

index = (hash(key) + i²) % table\_size

Reduces clustering, but harder to implement.

**c) Double Hashing**

Use a second hash function to calculate jump size.

java

CopyEdit

index = (hash1(key) + i \* hash2(key)) % table\_size

Very efficient but needs two good hash functions.

**2. ✅ Chaining (Separate Chaining)**

Each index in the hash table stores a **linked list** (or a bucket). All elements mapping to same index are stored in the list.

text

CopyEdit

Index 3 → [12] → [22] → [32]

📌 This is more memory-efficient for high-collision situations.

**🔸 Java Code: Simple Hash Table with Chaining**

java

CopyEdit

import java.util.LinkedList;

class HashTable {

private final int SIZE = 10;

LinkedList<Integer>[] table;

public HashTable() {

table = new LinkedList[SIZE];

for (int i = 0; i < SIZE; i++) {

table[i] = new LinkedList<>();

}

}

int hash(int key) {

return key % SIZE;

}

void insert(int key) {

int index = hash(key);

table[index].add(key);

}

void display() {

for (int i = 0; i < SIZE; i++) {

System.out.print("Index " + i + ": ");

for (int val : table[i]) {

System.out.print(val + " → ");

}

System.out.println("null");

}

}

}

**🧠 Summary Table**

| **Technique** | **Formula/Logic** | **Efficiency** |
| --- | --- | --- |
| Direct | index = key | Fast, high space |
| Division | index = key % size | Simple, popular |
| Multiplicative | floor(size × ((key × A) mod 1)) | Good distribution |
| Folding | Sum of parts of key | Easy to implement |
| Mid-Square | Use middle digits of squared key | Moderate |
| Linear Probing | (index + i) % size | Clustering |
| Quadratic Probing | (index + i²) % size | Less clustering |
| Double Hashing | (hash1 + i × hash2) % size | Excellent |
| Chaining | Linked list per index | Best for frequent collisions |
|  |  |  |

**Day11**

**🧠 What is a Graph?**

A **Graph** is a collection of:

* **Vertices (Nodes)**: Represent entities or points.
* **Edges (Links)**: Represent relationships or paths between nodes.

A graph can be used to model social networks, maps, computer networks, etc.

**🧱 COMPONENTS OF A GRAPH**

**🔹 Vertex (Node)**

A point in the graph.

css

CopyEdit

A

Here, A is a single vertex.

**🔹 Edge**

A line connecting two vertices.

css

CopyEdit

A --- B

A and B are connected by an edge.

**🔄 HOW GRAPHS WORK INTERNALLY?**

Let’s say:

less

CopyEdit

A -- B

|

C

You can represent this graph in two ways:

**1. Adjacency Matrix (2D Array)**

|  | **A** | **B** | **C** |
| --- | --- | --- | --- |
| A | 0 | 1 | 1 |
| B | 1 | 0 | 0 |
| C | 1 | 0 | 0 |

* 1 means there's a connection.
* 0 means no connection.

**2. Adjacency List**

java

CopyEdit

A → [B, C]

B → [A]

C → [A]

**📚 TYPES OF GRAPHS (DETAILED EXPLANATION)**

We’ll now look at **each type of graph** in detail with a **diagram and explanation**.

**1. Null Graph**

**Definition**: A graph with **no edges** at all.

**Diagram:**

css

CopyEdit

A B C

Each node stands alone.

🔍 **Use**: Used to model situations where objects exist but are not yet connected.

**2. Trivial Graph**

**Definition**: A graph with **only one vertex** and **no edges**.

**Diagram:**

css

CopyEdit

A

🔍 **Use**: A theoretical base case for recursion or graph algorithms.

**3. Undirected Graph**

**Definition**: A graph where edges have **no direction**. It’s a two-way connection.

**Diagram:**

less

CopyEdit

A --- B

| |

C --- D

* If A is connected to B, then B is also connected to A.

🔍 **Use**: Facebook friendships, road maps (without direction).

**4. Directed Graph (Digraph)**

**Definition**: A graph where each edge has a **direction** (from one node to another).

**Diagram:**

css

CopyEdit

A → B

↓ ↑

C → D

* A → B means A is connected to B, but **not vice versa**.

🔍 **Use**: Twitter followings, task dependencies, traffic direction.

**5. Weighted Graph**

**Definition**: A graph where edges have a **weight or cost** (e.g., distance or time).

**Diagram:**

less

CopyEdit

A --4-- B

| /

2 6

| /

C

🔍 **Use**: Google Maps, network latency, minimum spanning trees.

**6. Connected Graph**

**Definition**: Every vertex is **reachable** from every other vertex.

**Diagram:**

css

CopyEdit

A -- B -- C

\ /

\-- D

🔍 **Use**: Any fully working computer network.

**7. Disconnected Graph**

**Definition**: At least one vertex **is not reachable** from others.

**Diagram:**

css

CopyEdit

A -- B D E

* D and E are disconnected.

🔍 **Use**: Social networks where some people are not connected to anyone.

**8. Cyclic Graph**

**Definition**: A graph that contains **at least one cycle** (a path that starts and ends at the same vertex).

**Diagram:**

css

CopyEdit

A → B → C

↑ ↓

←--- D --

🔍 **Use**: Detecting loops in scheduling, deadlocks in OS.

**9. Acyclic Graph**

**Definition**: A graph that has **no cycles**.

**Diagram:**

css

CopyEdit

A → B → C → D

🔍 **Use**: Task scheduling, dependency resolution.

**10. Directed Acyclic Graph (DAG)**

**Definition**: A **Directed** graph with **no cycles**.

**Diagram:**

css

CopyEdit

A → B → C

↓

D

🔍 **Use**: Build systems (Makefile), version control systems (Git), task scheduling.

**11. Strongly Connected Graph**

**Definition**: In a **directed graph**, **every node is reachable from every other node**.

**Diagram:**

css

CopyEdit

A → B

↑ ↓

C ← D

You can reach any node from any other node **following directions**.

🔍 **Use**: Road systems with full navigability, web crawlers.

**12. Bipartite Graph**

**Definition**: A graph whose vertices can be **divided into 2 groups**, and **edges only connect from one group to another**, never within the same group.

**Diagram:**

makefile

CopyEdit

Set1: A B

\ /

Set2: C D

Edges: A–C, A–D, B–C, B–D  
No edge between A–B or C–D

🔍 **Use**: Job assignment, matching problems, two-party relationships.

**13. Complete Graph**

**Definition**: A graph where **every vertex is connected** to **every other vertex**.

**Diagram:**

less

CopyEdit

A --- B

|\ /|

| \ / |

| C |

\\_\_\_\_\_/

Edges between every pair.

🔍 **Use**: Theoretical models, brute-force traveling salesman.

**🧑‍💻 Java Tip for Representing Any Graph**

Use **adjacency list** for sparse graphs and **adjacency matrix** for dense graphs.

java

CopyEdit

// Adjacency List using HashMap

Map<Integer, List<Integer>> graph = new HashMap<>();

graph.put(0, Arrays.asList(1, 2));

graph.put(1, Arrays.asList(0, 3));

// etc.

For **weighted graphs**, use:

java

CopyEdit

Map<Integer, List<Pair<Integer, Integer>>> weightedGraph = new HashMap<>();

// Node, List of (neighbor, weight)

**✅ Summary Table (Quick Look)**

| **Graph Type** | **Direction** | **Weight** | **Cycles** | **Special Property** |
| --- | --- | --- | --- | --- |
| Null | No | No | No | No edges at all |
| Trivial | No | No | No | One vertex only |
| Undirected | No | Optional | Optional | Two-way edges |
| Directed | Yes | Optional | Optional | One-way edges |
| Weighted | Optional | Yes | Optional | Each edge has cost/weight |
| Connected | Optional | Optional | Optional | All nodes reachable |
| Disconnected | Optional | Optional | Optional | Some nodes unreachable |
| Cyclic | Optional | Optional | Yes | Has at least one cycle |
| Acyclic | Optional | Optional | No | No cycles |
| DAG | Yes | Optional | No | Directed & Acyclic |
| Strongly Connected | Yes | Optional | Optional | Every node reachable from others |
| Bipartite | Optional | Optional | Optional | Two groups, only cross-group edges |
| Complete | Optional | Optional | Optional | Every node connects to every other |

**1. Adjacency Matrix (Graph Representation)**

**Definition**: An adjacency matrix is a 2D array used to represent a graph. Each cell in the matrix indicates whether there is an edge between a pair of vertices.

**Undirected Graph**

Let the graph have 4 vertices: A, B, C, D

Edges: A-B, A-D, B-C

**Matrix Representation** (0 = no edge, 1 = edge):

A B C D

A [ 0 1 0 1 ]

B [ 1 0 1 0 ]

C [ 0 1 0 0 ]

D [ 1 0 0 0 ]

* Since the graph is undirected, the matrix is symmetric.

**Directed Graph**

Edges: A -> B, A -> D, B -> C

**Matrix Representation**:

A B C D

A [ 0 1 0 1 ]

B [ 0 0 1 0 ]

C [ 0 0 0 0 ]

D [ 0 0 0 0 ]

* Not symmetric; direction matters.

**Space Complexity**: O(V^2), where V is the number of vertices. Even if no edge exists, space is still consumed.

**2. Adjacency List (Graph Representation)**

**Definition**: Stores a list of all neighbors (connected nodes) for each vertex.

**Undirected Graph**

A -> [B, D]

B -> [A, C]

C -> [B]

D -> [A]

**Directed Graph**

A -> [B, D]

B -> [C]

C -> []

D -> []

**Space Complexity**: O(V + E), where V = vertices, E = edges.

* More efficient for sparse graphs (less edges).

**3. Graph Traversals**

**Definition**: Visiting all the nodes in a graph in a specific order.

Two main types:

* **BFS (Breadth-First Search)**
* **DFS (Depth-First Search)**

**4. Breadth-First Search (BFS)**

**Concept**: Explore all neighbors at the current depth before moving to the next depth level.

**Uses**: Finding shortest path in unweighted graphs, level-order traversal.

**How It Works**:

* Use a **queue**.
* Mark each node as visited.
* Traverse level by level.

**Example Graph**:

A -- B -- C

| |

D E

**Start from A**: Queue: A → B → D → C → E

**Java-like steps**:

1. Enqueue A, mark visited
2. Dequeue A, enqueue neighbors B and D
3. Dequeue B, enqueue C
4. Dequeue D
5. Dequeue C, enqueue E
6. Dequeue E

**Traversal Order**: A → B → D → C → E

**5. Depth-First Search (DFS)**

**Concept**: Explore as far as possible along one branch before backtracking.

**Uses**: Pathfinding, topological sorting, cycle detection.

**How It Works**:

* Use a **stack** (or recursion).
* Visit a node, explore neighbors recursively.

**Same Graph**:

A -- B -- C

| |

D E

**Start from A**: Stack/Recursion path: A → B → C → E → D

**Java-like steps**:

1. Visit A
2. Go to B
3. Go to C
4. Go to E
5. Backtrack, visit D

**Traversal Order**: A → B → C → E → D

**6. Minimum Spanning Tree (MST)**

**Definition**: A subset of edges that connects all vertices with the **minimum total weight** and no cycles.

**Applicable to**: Weighted, connected, undirected graphs.

**Algorithms**:

* **Prim's Algorithm**: Start from one node, grow tree by adding the lowest weight edge.
* **Kruskal's Algorithm**: Sort all edges, keep adding edges with least weight that don’t form cycles.

**Use**: Network design, circuit layout.

**7. Differences: BFS, DFS, Weighted Graph**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **BFS** | **DFS** | **Weighted Graph** |
| Search Type | Level-wise | Depth-wise | Based on edge weight |
| Data Structure | Queue | Stack / Recursion | Priority Queue (for MST) |
| Use Case | Shortest path (unweighted) | Topology, cycle detection | MST, Dijkstra, A\* |
| Time Complexity | O(V + E) | O(V + E) | O(E log V) (Prim's/Kruskal) |

**8. Greedy Algorithms**

**Definition**: Builds up a solution piece by piece, choosing the best option at each step.

**Examples**:

* Prim’s and Kruskal’s MST algorithms
* Dijkstra’s Shortest Path

**Working**:

* No backtracking
* Make locally optimal choice hoping for global optimum

**Limitation**: Not always gives best result (e.g., for NP-Hard problems)

**9. Principle of Optimality**

**Definition**: If a solution is optimal, then its sub-solutions must also be optimal.

**Used In**: Dynamic Programming, Greedy Algorithms

**Example**:

* In Dijkstra’s algorithm, the shortest path to a node includes the shortest paths to its predecessors.

**1. Adjacency Matrix (Graph Representation)**

**Definition**: An adjacency matrix is a 2D array used to represent a graph. Each cell in the matrix indicates whether there is an edge between a pair of vertices.

**Undirected Graph**

Let the graph have 4 vertices: A, B, C, D

Edges: A-B, A-D, B-C

**Matrix Representation** (0 = no edge, 1 = edge):

A B C D

A [ 0 1 0 1 ]

B [ 1 0 1 0 ]

C [ 0 1 0 0 ]

D [ 1 0 0 0 ]

* Since the graph is undirected, the matrix is symmetric.

**Directed Graph**

Edges: A -> B, A -> D, B -> C

**Matrix Representation**:

A B C D

A [ 0 1 0 1 ]

B [ 0 0 1 0 ]

C [ 0 0 0 0 ]

D [ 0 0 0 0 ]

* Not symmetric; direction matters.

**Space Complexity**: O(V^2), where V is the number of vertices. Even if no edge exists, space is still consumed.

**2. Adjacency List (Graph Representation)**

**Definition**: Stores a list of all neighbors (connected nodes) for each vertex.

**Undirected Graph**

A -> [B, D]

B -> [A, C]

C -> [B]

D -> [A]

**Directed Graph**

A -> [B, D]

B -> [C]

C -> []

D -> []

**Space Complexity**: O(V + E), where V = vertices, E = edges.

* More efficient for sparse graphs (less edges).

**3. Graph Traversals**

**Definition**: Visiting all the nodes in a graph in a specific order.

Two main types:

* **BFS (Breadth-First Search)**
* **DFS (Depth-First Search)**

**4. Breadth-First Search (BFS)**

**Concept**: Explore all neighbors at the current depth before moving to the next depth level.

**Uses**: Finding shortest path in unweighted graphs, level-order traversal.

**How It Works**:

* Use a **queue**.
* Mark each node as visited.
* Traverse level by level.

**Example Graph**:

A -- B -- C

| |

D E

**Start from A**: Queue: A → B → D → C → E

**Java-like steps**:

1. Enqueue A, mark visited
2. Dequeue A, enqueue neighbors B and D
3. Dequeue B, enqueue C
4. Dequeue D
5. Dequeue C, enqueue E
6. Dequeue E

**Traversal Order**: A → B → D → C → E

**5. Depth-First Search (DFS)**

**Concept**: Explore as far as possible along one branch before backtracking.

**Uses**: Pathfinding, topological sorting, cycle detection.

**How It Works**:

* Use a **stack** (or recursion).
* Visit a node, explore neighbors recursively.

**Same Graph**:

A -- B -- C

| |

D E

**Start from A**: Stack/Recursion path: A → B → C → E → D

**Java-like steps**:

1. Visit A
2. Go to B
3. Go to C
4. Go to E
5. Backtrack, visit D

**Traversal Order**: A → B → C → E → D

**6. Minimum Spanning Tree (MST)**

**Definition**: A subset of edges that connects all vertices with the **minimum total weight** and no cycles.

**Applicable to**: Weighted, connected, undirected graphs.

**Algorithms**:

* **Prim's Algorithm**: Start from one node, grow tree by adding the lowest weight edge.
* **Kruskal's Algorithm**: Sort all edges, keep adding edges with least weight that don’t form cycles.

**Use**: Network design, circuit layout.

**7. Differences: BFS, DFS, Weighted Graph**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **BFS** | **DFS** | **Weighted Graph** |
| Search Type | Level-wise | Depth-wise | Based on edge weight |
| Data Structure | Queue | Stack / Recursion | Priority Queue (for MST) |
| Use Case | Shortest path (unweighted) | Topology, cycle detection | MST, Dijkstra, A\* |
| Time Complexity | O(V + E) | O(V + E) | O(E log V) (Prim's/Kruskal) |

**8. Greedy Algorithms**

**Definition**: Builds up a solution piece by piece, choosing the best option at each step.

**Examples**:

* Prim’s and Kruskal’s MST algorithms
* Dijkstra’s Shortest Path

**Working**:

* No backtracking
* Make locally optimal choice hoping for global optimum

**Limitation**: Not always gives best result (e.g., for NP-Hard problems)

**9. Principle of Optimality**

**Definition**: If a solution is optimal, then its sub-solutions must also be optimal.

**Used In**: Dynamic Programming, Greedy Algorithms

**Example**:

* In Dijkstra’s algorithm, the shortest path to a node includes the shortest paths to its predecessors.

**10. Dynamic Programming (DP)**

**Definition**: A technique to solve problems by breaking them down into simpler subproblems and storing their solutions (memoization).

**Hierarchy**:

* Optimization Problem
  + Overlapping Subproblems
  + Optimal Substructure
  + Memoization/Tabulation

**Example Problems**:

* Fibonacci Sequence
* Knapsack Problem
* Longest Common Subsequence

**Memoization (Top Down)**: Store results of expensive function calls and reuse them. **Tabulation (Bottom Up)**: Solve subproblems first, build solution from them.

**Time Complexity**: Reduced from exponential to polynomial (O(n^2), O(n))

**11. Kruskal's Algorithm (Detailed)**

**Steps**:

1. Sort edges by increasing weight
2. Initialize each vertex as a disjoint set
3. Add edges one by one, if they don’t form a cycle (Union-Find)

**Diagram**: Graph:

A --1-- B

| /

4 3

| /

C

Edges: (A-B,1), (B-C,3), (A-C,4)

Sorted Edges: (A-B), (B-C), (A-C)

Choose A-B, then B-C → MST is A-B and B-C

**Java Logic**:

* Create Edge class
* Sort edges
* Use Union-Find (Disjoint Set)

**12. AVL Tree (Self-Balancing BST)**

**Definition**: Binary Search Tree that maintains balance factor = height(left) - height(right)

**Balance Factor**: -1, 0, or +1

**Rotations**:

* **LL Rotation**: Right Rotate
* **RR Rotation**: Left Rotate
* **LR Rotation**: Left Rotate + Right Rotate
* **RL Rotation**: Right Rotate + Left Rotate

**Formula**:

* Height(Node) = 1 + max(height(left), height(right))

**Java Insertion Program**: Includes rotation functions and balance check

**Operations**:

* Insertion
* Deletion
* Search

**13. Red-Black Tree**

**Definition**: Self-balancing BST with extra color property (Red, Black)

**Properties**:

1. Node is red or black
2. Root is black
3. Red nodes cannot have red children
4. Every path from root to leaf has same number of black nodes

**Balance Property**: Ensures tree height is O(log n)

**Java Program**:

* Node with color
* Fix violations using rotations (similar to AVL)

**14. Dijkstra’s Algorithm**

**Use**: Find shortest path in weighted graph (no negative weights)

**Steps**:

1. Set distances[] = ∞, source = 0
2. Use PriorityQueue to pick smallest unvisited node
3. Update neighbor distances

**Time Complexity**: O(V + E log V)

**15. Bellman-Ford Algorithm**

**Use**: Shortest path with negative weights

**Steps**:

1. Initialize distances
2. Repeat V-1 times: for each edge, update distance
3. Detect negative cycles in 1 more iteration

**Time**: O(VE)

**16. Floyd Warshall Algorithm**

**Use**: All-pairs shortest paths

**Steps**:

1. Create dist[i][j] matrix
2. For each intermediate k, update dist[i][j] = min(dist[i][j], dist[i][k]+dist[k][j])

**Time**: O(V^3)

**17. Prim’s Algorithm**

**Use**: Minimum Spanning Tree (MST)

**Steps**:

1. Pick any start node
2. Use Priority Queue to select minimum weight edge connecting to tree
3. Repeat until all nodes added

**Time**: O(E log V)

**Comparison**:

* **Prim**: Grows tree
* **Kruskal**: Connects components

**14. Dijkstra’s Algorithm (Detailed Explanation)**

**Definition**: Dijkstra’s Algorithm is used to find the shortest path from a single source node to all other nodes in a **weighted graph with non-negative edge weights**. It is widely used in routing and navigation systems.

**Core Idea:**

It works by continuously selecting the node with the smallest known distance from the source and exploring all of its adjacent, unvisited neighbors.

**How It Works:**

* Maintains an array distance[] where distance[i] stores the current shortest distance from the source node to vertex i.
* Uses a **min-priority queue (or min-heap)** to pick the node with the least known distance.
* Once a node is marked visited, its shortest path is finalized and not updated again.

**Algorithm Steps:**

1. **Initialize** the distance[] array. Set the source distance to 0 and all others to ∞.
2. Insert all vertices into a **min-heap** with their distances.
3. While the priority queue is not empty:
   * Extract the vertex u with the **minimum distance**.
   * For each neighbor v of u, check if the path through u offers a **shorter distance**:
     + If yes, update distance[v] and push v into the queue with the new distance.

**Time Complexity:**

* Using Priority Queue: **O((V + E) log V)**
* V = vertices, E = edges

**Graph Example:**

A

/ \

4 1

/ \

B---1---C

\

2

\

D

Start from A:

* Distance[A] = 0
* Distance[B, C, D] = ∞

Iteration:

* Visit A → update B (4), C (1)
* Visit C → update B (C to B = 1), so Distance[B] = 2 now
* Visit B → update D (B to D = 2), Distance[D] = 4

Final distances: A: 0, B: 2, C: 1, D: 4

**Applications:**

* GPS and navigation systems
* Network routing protocols (like OSPF)
* Games (pathfinding algorithms like A\*)

**15. Kruskal's Algorithm (Minimum Spanning Tree)**

**Definition**: Kruskal’s Algorithm is a **greedy algorithm** that builds the Minimum Spanning Tree (MST) by **choosing the smallest possible edge that doesn't form a cycle**, until all vertices are connected.

**Core Idea:**

Sort edges by weight and keep adding the smallest edge unless it creates a cycle. It uses **disjoint-set (Union-Find)** data structures to detect cycles.

**Steps:**

1. Sort all edges by weight in **increasing order**.
2. Initialize MST as empty. Each vertex is its own set.
3. For each edge (u, v):
   * If u and v belong to **different sets**, include this edge in MST and **union** the sets.
   * If they belong to the same set, **skip** the edge (to avoid cycle).
4. Repeat until MST contains (V - 1) edges.

**Time Complexity:**

* **O(E log E)** due to edge sorting and Union-Find operations.

**Graph Example:**

(1)

A ------- B

\ /

4\ /3

\ /

C

Edges: A-B (1), B-C (3), A-C (4)

Sorted Edges: A-B, B-C, A-C

* Pick A-B
* Pick B-C → Now A, B, C are connected → Done

MST Edges: A-B, B-C

**Important Concepts:**

* Uses Union-Find to detect cycles
* Builds forest first, then merges trees
* **Greedy**: Picks the best local option

**Applications:**

* Designing least-cost network (telecom, computer networks)
* Clustering algorithms (in ML)

**16. Prim’s Algorithm (Minimum Spanning Tree)**

**Definition**: Prim’s Algorithm builds the Minimum Spanning Tree by **growing the MST one vertex at a time**, always choosing the **minimum-weight edge** that connects a visited node to an unvisited node.

**Core Idea:**

Unlike Kruskal’s that adds edges, Prim’s adds **vertices** by always choosing the next lightest edge connecting a visited and unvisited node.

**Steps:**

1. Start from an arbitrary node (say node A).
2. Mark it as visited.
3. Insert all its adjacent edges into a **min-heap**.
4. Pick the **minimum edge** from the heap:
   * If it connects to an **unvisited vertex**, add it to the MST and mark the vertex as visited.
   * Add all edges of this new vertex to the heap.
5. Repeat until all vertices are visited.

**Time Complexity:**

* With Min Heap + Adjacency List: **O(E log V)**

**Example Graph:**

1

A ------- B

| |

4 3

| |

C ------- D

2

Start from A:

* Include A-B (1)
* Include B-D (3)
* Include D-C (2)

MST: A-B, B-D, D-C

**Difference Between Prim’s and Kruskal’s:**

| **Feature** | **Prim’s** | **Kruskal’s** |
| --- | --- | --- |
| Grows from | A starting vertex | Independent edges |
| Uses | Priority Queue | Union-Find Set |
| Suitable for | Dense Graphs | Sparse Graphs |
| Edge selection | From visited to unvisited | Global edge weight |

**Applications:**

* Network designs (fiber, cable routing)
* Approximate algorithms for NP-hard problems (e.g., Traveling Salesman)

**Key Points for Interviews:**

* Be ready to compare Prim vs Kruskal
* Understand how both avoid cycles
* Know when to prefer one over the other