Searching in Java

Searching means **finding a required element** from a collection of data (array, list, tree, etc.).

There are mainly **two categories** of searching in Java:

1. Linear Search (Sequential Search)

Concept

It checks elements **one-by-one** from start to end until the key is found.

When to use?

- Array is **unsorted**
- Data is **small**
- Simple implementation is needed

Time Complexity

- Best: **O(1)**
- Worst: **O(n)**

Java Code

```
public class LinearSearch {
    public static int linearSearch(int[] arr, int key) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == key)
                return i;
        }
        return -1; // not found
    }

    public static void main(String[] args) {
        int[] nums = {10, 20, 30, 40, 50};
        int res = linearSearch(nums, 30);
        System.out.println("Element found at index: " + res);
    }
}
```

Binary Search

Concept

Binary Search works on **sorted arrays only**.
It repeatedly divides the array into **two halves**.

Steps

1. Find mid

2. If key == mid → return index
3. If key < mid → search in left half
4. Else → search in right half

Time Complexity

- Best: **O(1)**
- Worst: **O(log n)**

Java Code (Iterative Binary Search)

```
public class BinarySearch {
    public static int binarySearch(int[] arr, int key) {
        int low = 0, high = arr.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;

            if (arr[mid] == key)
                return mid;
            else if (key < arr[mid])
                high = mid - 1;
            else
                low = mid + 1;
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] nums = {10, 20, 30, 40, 50};
        int pos = binarySearch(nums, 40);
        System.out.println("Found at index: " + pos);
    }
}
```

Sorting in Java

Sorting means arranging data in a **specific order**, usually **ascending or descending**.

Example:
Unsorted → {5, 2, 9, 1, 6}
Sorted → {1, 2, 5, 6, 9}

Sorting improves searching, data processing, and optimizes algorithms.

Types of Sorting Algorithms

Sorting algorithms are mainly divided into **two categories**:

Simple Sorting Algorithms (Basic, Easy)

A) Bubble Sort

- Repeatedly compare **adjacent elements**
- Swap if wrong order
- Largest element "bubbles up" to end

Time Complexity:
Best – O(n)
Worst – O(n^2)

Java Code

```
public class BubbleSort {
    public static void bubbleSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = 0; j < arr.length - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
}
```

B) Selection Sort

- Select **minimum element** from unsorted part
- Put it at correct position

Time Complexity: Worst – O(n^2)

Code

```
public class SelectionSort {
    public static void selectionSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            int min = i;
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[j] < arr[min])
                    min = j;
            }
            int temp = arr[min];
            arr[min] = arr[i];
            arr[i] = temp;
        }
    }
}
```

C) Insertion Sort

- Insert each element into its **correct place** in sorted part

Time Complexity:
Best: O(n)
Worst: O(n²)

Code

```
public class InsertionSort {
    public static void insertionSort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            int key = arr[i];
            int j = i - 1;

            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }
}
```

Efficient Sorting Algorithms (Fast, Used in real world)

A) Merge Sort (Divide & Conquer)

- Divide array into two halves
- Sort each half
- Merge them

Time Complexity: O(n log n)
Stable sorting

Code

```
public class MergeSort {

    public static void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }

    static void merge(int[] arr, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        int[] L = new int[n1];
        int[] R = new int[n2];

        for (int i = 0; i < n1; i++)
            L[i] = arr[left + i];
        for (int i = 0; i < n2; i++)
            R[i] = arr[mid + 1 + i];
```

```
        int i = 0, j = 0, k = left;

        while (i < n1 && j < n2) {
            if (L[i] <= R[j])
                arr[k++] = L[i++];
            else
                arr[k++] = R[j++];
        }

        while (i < n1)
            arr[k++] = L[i++];
        while (j < n2)
            arr[k++] = R[j++];
    }
}
```

B) Quick Sort

- Pick a pivot
- Place pivot in correct position
- Elements smaller → left
- Elements larger → right

Time Complexity:
Best/Average → O(n log n)
Worst → $O(n^2)$

Code

```
public class QuickSort {

    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int p = partition(arr, low, high);
            quickSort(arr, low, p - 1);
            quickSort(arr, p + 1, high);
        }
    }

    static int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }
}
```

C) Heap Sort

- Build Max Heap
- Swap root with last element
- Heapify remaining array

Time Complexity: O(n log n)

Code (short)

```
public class HeapSort {

    public void sort(int[] arr) {
        int n = arr.length;

        for (int i = n/2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        for (int i = n - 1; i > 0; i--) {
            int temp = arr[0]; arr[0] = arr[i]; arr[i] = temp;
            heapify(arr, i, 0);
        }
    }

    void heapify(int[] arr, int n, int i) {
        int largest = i;
        int l = 2*i + 1;
        int r = 2*i + 2;

        if (l < n && arr[l] > arr[largest])
            largest = l;

        if (r < n && arr[r] > arr[largest])
            largest = r;

        if (largest != i) {
            int temp = arr[i]; arr[i] = arr[largest]; arr[largest] = temp;
            heapify(arr, n, largest);
        }
    }
}
```

Sorting Comparison Table

Sorting	Best	Worst	Stable	Use Case
Bubble	O(n)	$O(n^2)$	✔	Very small data
Selection	$O(n^2)$	$O(n^2)$	✖	When swaps should be minimized
Insertion	O(n)	$O(n^2)$	✔	Nearly sorted data
Merge	O(n log n)	O(n log n)	✔	Large data, stable need
Quick	O(n log n)	$O(n^2)$	✖	Most used, fast
Heap	O(n log n)	O(n log n)	✖	Priority queues

Built-in Sorting in Java

Java has powerful built-in methods:

Arrays.sort() (Dual-Pivot QuickSort)

```
Arrays.sort(arr);
```

Collections.sort() (Merge Sort)

```
Collections.sort(list);
```

GRAPH – Introduction

A **Graph** is a non-linear data structure used to represent **relationships** or **connections** between objects.

A graph consists of:

Vertices (Nodes) → represent objects

Edges (Links) → represent connections between objects

Example:
Cities = Vertices
Roads = Edges

Why Use Graphs?

Graphs represent real-world problems like:

- Social networks (friends)
- Google Maps (routes)
- Internet networks
- Airline connections
- Recommendation systems

Graph Terminology (Important)

Term	Meaning
Vertex (V)	Node of graph
Edge (E)	Connection between nodes
Adjacent	Two nodes with an edge between them
Degree	Number of edges connected to a node
Path	Sequence of edges from one node to another
Cycle	Path that starts and ends at the same node

Term	Meaning
Connected Graph	Every node can reach every other node
Component	Sub-graph where all nodes are connected

Types of Graphs

Directed Graph (Digraph)

Edges have direction.

Example: A → B (one-way)

Undirected Graph

Edges have no direction.

Example: A — B (two-way)

Weighted Graph

Edges have weights (distance/cost/time).

Example: A —5→ B (edge weight = 5)

Unweighted Graph

No weights on edges.

Cyclic Graph

Contains at least one cycle.

Acyclic Graph

No cycles. (Tree is an example)

Connected Graph (Undirected)

Every vertex is reachable.

Disconnected Graph

Some nodes cannot be reached.

Directed Acyclic Graph (DAG)

- Directed

- No cycles
  Used in: Scheduling, Compilers (topological sort)

Graph Representation

1. Adjacency Matrix

A 2D array `matrix[V][V]`
If edge exists between `i` and `j`, store `1` (or weight).

Example:

	0	**1**	**2**
0	0	1	0
1	1	0	1
2	0	1	0

Pros:

- Easy to implement
- Good for dense graphs

Cons:

- Uses $O(V^2)$ memory

2. Adjacency List (Most Used)

Each vertex stores a list of connected vertices.

Example:
0 → 1
1 → 0, 2
2 → 1

Pros:

- Uses less memory
- Best for sparse graphs

Cons:

- Harder to implement

Graph Traversal Algorithms

1. Depth First Search (DFS)

- Goes deep first
- Uses **Stack** (or recursion)

2. Breadth First Search (BFS)

- Visits level by level
- Uses **Queue**

Both run in:

Time Complexity: O(V + E)

Applications of Graphs

- GPS navigation
- Social networks
- Web crawling
- Shortest path algorithms (Dijkstra, Bellman-Ford)
- Network routing
- AI pathfinding
- Compiler design (DAG)

Java Code: Graph Using Adjacency List

```
import java.util.*;

class Graph {
    int V;
    ArrayList<ArrayList<Integer>> adj;

    Graph(int v) {
        V = v;
        adj = new ArrayList<>();

        for (int i = 0; i < v; i++)
            adj.add(new ArrayList<>());
    }

    // Add edge (undirected)
    void addEdge(int u, int v) {
        adj.get(u).add(v);
        adj.get(v).add(u);
    }

    // Print graph
    void printGraph() {
        for (int i = 0; i < V; i++) {
            System.out.print(i + " -> ");
            for (int x : adj.get(i)) {
                System.out.print(x + " ");
            }
            System.out.println();
        }
    }
```

```
    public static void main(String[] args) {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 3);

        g.printGraph();
    }
}
```

Output

```
0 -> 1 2
1 -> 0 2
2 -> 0 1 3
3 -> 2
```

Summary (One-Page Notes)

- Graph → Non-linear DS with nodes (V) and edges (E)
- Types → Directed/Undirected, Weighted/Unweighted, Cyclic/Acyclic
- Representations → Adjacency Matrix, Adjacency List
- Traversal → DFS, BFS
- Applications → Networks, Maps, Social Media, AI

Depth-First Search (DFS) – Java Code (Adjacency List)

DFS means exploring **as far as possible** along one branch before backtracking.

DFS Using Recursion (Most Common Method)

Java Code

```
import java.util.*;

class Graph {
    private int vertices;
    private LinkedList<Integer>[] adj;

    Graph(int v) {
        vertices = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; i++) {
            adj[i] = new LinkedList<>();
        }
    }

    void addEdge(int src, int dest) {
        adj[src].add(dest);
        adj[dest].add(src);  // For undirected graph
    }

    void DFSUtil(int node, boolean[] visited) {
```

```
        visited[node] = true;
        System.out.print(node + " ");

        // visit all neighbours
        for (int next : adj[node]) {
            if (!visited[next]) {
                DFSUtil(next, visited);
            }
        }
    }

    void DFS(int start) {
        boolean[] visited = new boolean[vertices];
        DFSUtil(start, visited);
    }

    public static void main(String[] args) {
        Graph graph = new Graph(5);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);

        System.out.println("DFS Traversal:");
        graph.DFS(0);   // Start DFS from node 0
    }
}
```

OUTPUT

```
DFS Traversal:
0 1 3 4 2
```

Order may differ based on adjacency list.

Explanation

DFS Steps

1. Start from a node.
2. Mark it **visited**.
3. Move to its **unvisited neighbour**.
4. Continue until no neighbour remains → **Backtrack**.
5. Repeat for all connected nodes.

Time & Space Complexity

Case	Complexity
Time Complexity	**O(V + E)**
Space Complexity	**O(V)** (visited array + recursion stack)

DFS Applications

- Path finding
- Detecting cycles in graph
- Topological sorting
- Solving puzzles (maze, sudoku)
- Tree traversals
- Connected components

BFS – Breadth-First Search

BFS explores nodes **level by level** (breadth-wise).
It uses a **Queue** to visit all neighbours first before moving deeper.

Java Code for BFS (Adjacency List)

```
import java.util.*;

class Graph {
    private int vertices;
    private LinkedList<Integer>[] adj;

    Graph(int v) {
        vertices = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; i++) {
            adj[i] = new LinkedList<>();
        }
    }

    void addEdge(int src, int dest) {
        adj[src].add(dest);
        adj[dest].add(src);  // For undirected graph
    }

    void BFS(int start) {
        boolean[] visited = new boolean[vertices];
        Queue<Integer> queue = new LinkedList<>();

        visited[start] = true;
        queue.add(start);

        while (!queue.isEmpty()) {
            int node = queue.poll();
            System.out.print(node + " ");

            // visit all neighbours
            for (int next : adj[node]) {
                if (!visited[next]) {
                    visited[next] = true;
                    queue.add(next);
                }
            }
        }
    }

    public static void main(String[] args) {
        Graph graph = new Graph(6);
```

```
        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);
        graph.addEdge(2, 5);

        System.out.println("BFS Traversal:");
        graph.BFS(0);  // Start BFS from node 0
    }
}
```

Output

```
BFS Traversal:
0 1 2 3 4 5
```

Order may vary depending on edges.

Explanation

BFS Steps

1. Start at a node
2. Mark it **visited**
3. Push it in **queue**
4. Pop from queue and visit all **unvisited neighbours**
5. Repeat until queue is empty

Time & Space Complexity

Case	Complexity
Time Complexity	**O(V + E)**
Space Complexity	**O(V)** (visited array + queue)

Applications of BFS

- Shortest path in **unweighted graphs**
- Web crawling
- GPS navigation
- Level-order traversal of trees
- Finding connected components
- Checking bipartite graph