```
// Pushing elements...
// Pushed 5 → [5]
// Pushed 10 → [10, 5]
// Pushed 15 → [15, 10, 5]
// Pushed 20 → [20, 15, 10, 5]

// Final Stack: [20, 15, 10, 5]
```

Popping Elements from a Stack

Popping removes the **top-most element** from a stack and returns it.

Stacks follow **LIFO (Last In, First Out)** order, so the most recently pushed element is removed first.

Using Java's `Deque`, the `pop()` method removes the top element in **O(1)** time.

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {

    public static void main(String[] args) {

        // Creating a Stack using Deque
        Deque<Integer> stack = new ArrayDeque<>();

        // Push elements first
        stack.push(5);
        stack.push(10);
        stack.push(15);
        stack.push(20);

        System.out.println("Initial Stack: " + stack);

        // Pop elements from the stack
        System.out.println("\nPopping elements...");

        int popped1 = stack.pop();
        System.out.println("Popped: " + popped1 + " → " + stack);

        int popped2 = stack.pop();
        System.out.println("Popped: " + popped2 + " → " + stack);
```

```
        int popped3 = stack.pop();
        System.out.println("Popped: " + popped3 + " → " + stack);

        int popped4 = stack.pop();
        System.out.println("Popped: " + popped4 + " → " + stack);

        // Attempting to pop from an empty stack will cause an
exception
        // stack.pop(); // Uncomment to test
    }
}

// Initial Stack: [20, 15, 10, 5]

// Popping elements...
// Popped: 20 → [15, 10, 5]
// Popped: 15 → [10, 5]
// Popped: 10 → [5]
// Popped: 5 → []
```

Peeking the Top Element

Peeking allows you to look at the **top element of the stack without removing it**.

It is useful when you want to inspect the last pushed item but keep it in the stack.

Using Java's `Deque`, the `peek()` or `peekFirst()` method retrieves the top element in **O(1)** time.

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {

    public static void main(String[] args) {

        // Creating a Stack using Deque
        Deque<Integer> stack = new ArrayDeque<>();

        // Push elements
        stack.push(5);
        stack.push(10);
        stack.push(15);
        stack.push(20);

        System.out.println("Current Stack: " + stack);

        // Peek top element without removing it
        Integer top = stack.peek();

        System.out.println("Top element (peek): " + top);
        System.out.println("Stack after peeking: " + stack);
    }
}
```

```
// Current Stack: [20, 15, 10, 5]
// Top element (peek): 20
// Stack after peeking: [20, 15, 10, 5]
```

Checking if a Stack Is Empty

Before performing stack operations like pop or peek, it's important to check whether the stack is empty.

Java's `Deque` provides the `isEmpty()` method, which returns **true** if the stack contains no elements.

This helps prevent errors such as popping from an empty stack.

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {

    public static void main(String[] args) {

        // Creating a Stack using Deque
        Deque<Integer> stack = new ArrayDeque<>();

        // Check if empty initially
        System.out.println("Is stack empty? " + stack.isEmpty());

        // Push some elements
        stack.push(10);
        stack.push(20);

        System.out.println("Stack after pushing elements: " +
stack);

        // Check again
        System.out.println("Is stack empty now? " +
stack.isEmpty());

        // Pop all elements
        stack.pop();
        stack.pop();
```

```
        // Final check
        System.out.println("Is stack empty after popping all
elements? " + stack.isEmpty());
    }
}

// Is stack empty? true
// Stack after pushing elements: [20, 10]
// Is stack empty now? false
// Is stack empty after popping all elements? true
```

Searching for an Element in a Stack

Stacks do not provide a direct search function, but using a `Deque` as a stack, you can search manually by iterating through the elements.

Since a stack is LIFO, we inspect each element until we find the target.

This example shows how to search an element in a stack using a simple loop.

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {

    // Function to search an element in the stack
    public static boolean searchElement(Deque<Integer> stack, int
target) {

        for (int element : stack) {
            if (element == target) {
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {

        // Creating a stack using Deque
        Deque<Integer> stack = new ArrayDeque<>();

        // Push elements
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.push(40);
```

```
        System.out.println("Stack: " + stack);

        int target = 30;

        // Search for element
        boolean found = searchElement(stack, target);

        if (found) {
            System.out.println("Element " + target + " found in
the stack.");
        } else {
            System.out.println("Element " + target + " not found
in the stack.");
        }
    }
}

// Stack: [40, 30, 20, 10]
// Element 30 found in the stack.
```

Reversing a Stack

Reversing a stack means transforming the order so that the bottom-most element becomes the top-most.

We can do this using **recursion** by repeatedly popping the top element and inserting it at the bottom of the stack.

This approach uses:

- `reverseStack()` — recursively reverses the stack
- `insertAtBottom()` — inserts an element at the bottom

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {

    // Insert element at bottom of stack
    public static void insertAtBottom(Deque<Integer> stack, int
value) {
        if (stack.isEmpty()) {
            stack.push(value);
            return;
        }

        int top = stack.pop();
        insertAtBottom(stack, value);
        stack.push(top);
    }

    // Reverse the stack
    public static void reverseStack(Deque<Integer> stack) {
        if (stack.isEmpty()) return;

        int top = stack.pop();
        reverseStack(stack);
```

```
        insertAtBottom(stack, top);
    }

    public static void main(String[] args) {

        Deque<Integer> stack = new ArrayDeque<>();
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.push(40);

        System.out.println("Original Stack: " + stack);

        reverseStack(stack);

        System.out.println("Reversed Stack: " + stack);
    }
}

// Original Stack: [40, 30, 20, 10]
// Reversed Stack: [10, 20, 30, 40]
```

Implementing a Stack Using an Array

A stack is a LIFO (Last-In, First-Out) data structure.

When implemented using an array, we manually control:

- `push()` → insert element at the top
- `pop()` → remove element from the top
- `peek()` → return the top element
- `isEmpty()` → check if the stack has no elements
- `isFull()` → check if the array is full

```
class ArrayStack {
    private int[] stack;
    private int top;
    private int capacity;

    // Constructor
    public ArrayStack(int size) {
        stack = new int[size];
        capacity = size;
        top = -1; // stack is empty
    }

    // Push element into stack
    public void push(int value) {
        if (isFull()) {
            System.out.println("Stack Overflow! Cannot push " +
value);
            return;
        }
        stack[++top] = value;
    }

    // Pop element from stack
    public int pop() {
```

```
        if (isEmpty()) {
            System.out.println("Stack Underflow! Cannot pop.");
            return -1;
        }
        return stack[top--];
    }

    // Peek top element
    public int peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty!");
            return -1;
        }
        return stack[top];
    }

    // Check if empty
    public boolean isEmpty() {
        return top == -1;
    }

    // Check if full
    public boolean isFull() {
        return top == capacity - 1;
    }

    // Display stack
    public void printStack() {
        if (isEmpty()) {
            System.out.println("Stack is empty!");
            return;
        }
        System.out.print("Stack: ");
        for (int i = 0; i <= top; i++) {
            System.out.print(stack[i] + " ");
```

```
        }
        System.out.println();
    }
}

public class Main {
    public static void main(String[] args) {
        ArrayStack stack = new ArrayStack(5);

        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.push(40);

        stack.printStack();

        System.out.println("Popped: " + stack.pop());
        System.out.println("Top Element: " + stack.peek());

        stack.printStack();

        stack.push(50);
        stack.push(60);
        stack.push(70); // overflow
    }
}

// Stack: 10 20 30 40
// Popped: 40
// Top Element: 30
// Stack: 10 20 30
// Stack Overflow! Cannot push 70
```

Implementing a Stack Using a Linked List

```
class Stack {

    private static class Node {
        int data;
        Node next;
        Node(int d) { data = d; }
    }

    private Node top;

    public void push(int x) {
        Node n = new Node(x);
        n.next = top;
        top = n;
    }

    public int pop() {
        if (top == null) {
            throw new RuntimeException("Stack Underflow");
        }
        int val = top.data;
        top = top.next;
        return val;
    }

    public int peek() {
        if (top == null) {
            throw new RuntimeException("Stack Empty");
        }
        return top.data;
    }

    public boolean isEmpty() {
```

```
        return top == null;
    }
}

public class Main {
    public static void main(String[] args) {
        Stack s = new Stack();
        s.push(10);
        s.push(20);
        s.push(30);

        System.out.println("Top: " + s.peek());
        System.out.println("Pop: " + s.pop());
        System.out.println("New Top: " + s.peek());
    }
}

// Top: 30
// Pop: 30
// New Top: 20
```

Balanced Parentheses Checker

```
import java.util.Stack;

public class Main {

    public static boolean isBalanced(String s) {
        Stack<Character> stack = new Stack<>();

        for (char ch : s.toCharArray()) {
            // Push opening brackets
            if (ch == '(' || ch == '{' || ch == '[') {
                stack.push(ch);
            }
            // Handle closing brackets
            else if (ch == ')' || ch == '}' || ch == ']') {
                if (stack.isEmpty()) return false;

                char top = stack.pop();

                if ((ch == ')' && top != '(') ||
                    (ch == '}' && top != '{') ||
                    (ch == ']' && top != '[')) {
                    return false;
                }
            }
        }

        return stack.isEmpty(); // If empty → balanced
    }

    public static void main(String[] args) {
        String test1 = "{[()]}";
        String test2 = "{[(])}";
        String test3 = "(())[]{}";
```

```
        String test4 = "((()))";

        System.out.println(test1 + " → " + isBalanced(test1));
        System.out.println(test2 + " → " + isBalanced(test2));
        System.out.println(test3 + " → " + isBalanced(test3));
        System.out.println(test4 + " → " + isBalanced(test4));
    }
}

// {[()]} → true
// {[(])} → false
// (())[]{} → true
// ((()) → false
```

Queues

Creating a Queue (Using LinkedList)

A **Queue** is a **FIFO (First-In-First-Out)** data structure, where elements are **added at the rear** and **removed from the front**. In Java, the `LinkedList` class implements the `Queue` interface, making it simple to create and use a queue.

```
import java.util.*;

public class Main {

    public static void main(String[] args) {
        // Create a Queue using LinkedList
        Queue<Integer> queue = new LinkedList<>();

        // Enqueue elements
        queue.add(10);
        queue.add(20);
        queue.add(30);
        queue.add(40);

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

        // Dequeue elements
        int removed = queue.remove(); // removes the front
element
        System.out.println("Removed element: " + removed);
        System.out.println("Queue after removal: " + queue);

        // Peek at the front element without removing
        int front = queue.peek();
        System.out.println("Front element: " + front);
    }
```

```
}

// Queue: [10, 20, 30, 40]
// Removed element: 10
// Queue after removal: [20, 30, 40]
// Front element: 20
```

Creating a Queue (Using ArrayDeque)

`ArrayDeque` is a **resizable-array implementation** of the `Deque` interface. It is **more efficient than `LinkedList`** for queue operations and can be used as a **FIFO queue**.

```
import java.util.*;

public class Main {

    public static void main(String[] args) {
        // Create a Queue using ArrayDeque
        Queue<Integer> queue = new ArrayDeque<>();

        // Enqueue elements
        queue.add(10);
        queue.add(20);
        queue.add(30);
        queue.add(40);

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

        // Dequeue elements
        int removed = queue.remove(); // removes the front
element
        System.out.println("Removed element: " + removed);
        System.out.println("Queue after removal: " + queue);

        // Peek at the front element without removing
        int front = queue.peek();
        System.out.println("Front element: " + front);
    }
}

// Queue: [10, 20, 30, 40]
```

```
// Removed element: 10
// Queue after removal: [20, 30, 40]
// Front element: 20
```

Enqueuing Elements

Enqueueing means **adding elements to the rear** of a queue. In Java, you can enqueue elements using methods like `add()` or `offer()`. Both work similarly, but `offer()` is generally preferred in **capacity-restricted queues** because it returns `false` instead of throwing an exception when the queue is full.

```
import java.util.*;

public class Main {

    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();

        // Enqueue elements
        queue.add("Apple");
        queue.add("Banana");
        queue.add("Cherry");
        queue.offer("Date"); // alternative method

        System.out.println("Queue after enqueuing elements: " +
queue);
    }
}

// Queue after enqueuing elements: [Apple, Banana, Cherry, Date]
```

Dequeuing Elements

Dequeuing means **removing elements from the front** of a queue. In Java, you can remove elements using `remove()` or `poll()`. Both remove the front element, but `poll()` returns `null` instead of throwing an exception if the queue is empty.

```
import java.util.*;

public class Main {

    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();

        // Enqueue elements
        queue.add("Apple");
        queue.add("Banana");
        queue.add("Cherry");

        System.out.println("Queue before dequeuing: " + queue);

        // Dequeue elements
        String removed1 = queue.remove(); // removes front
element
        System.out.println("Removed element: " + removed1);
        String removed2 = queue.poll(); // removes front element
safely
        System.out.println("Removed element: " + removed2);

        System.out.println("Queue after dequeuing: " + queue);
    }
}

// Queue before dequeuing: [Apple, Banana, Cherry]
// Removed element: Apple
// Removed element: Banana
```

```
// Queue after dequeuing: [Cherry]
```

Peeking Front and Rear Elements

Peeking allows you to look at the front or rear element of a queue without removing it. In Java:

- **Front element:** `peek()` or `element()`
- **Rear element:** For `LinkedList` and `ArrayDeque`, you can use `peekLast()` (Deque interface)

```
import java.util.*;

public class Main {

    public static void main(String[] args) {
        Deque<String> queue = new LinkedList<>();

        // Enqueue elements
        queue.add("Apple");
        queue.add("Banana");
        queue.add("Cherry");

        // Peek front element
        String front = queue.peek();      // equivalent to
peekFirst()
        System.out.println("Front element: " + front);

        // Peek rear element
        String rear = queue.peekLast();
        System.out.println("Rear element: " + rear);

        System.out.println("Queue remains: " + queue);
    }
}
// Front element: Apple
// Rear element: Cherry
// Queue remains: [Apple, Banana, Cherry]
```

Checking if a Queue Is Empty

Before performing operations like **dequeue** or **peek**, it's often important to check whether the queue has any elements. In Java, the **`isEmpty()`** method is used to determine if a queue is empty.

```
import java.util.*;

public class Main {

    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();

        // Initially empty
        System.out.println("Is queue empty? " + queue.isEmpty());
// true

        // Enqueue elements
        queue.add("Apple");
        queue.add("Banana");

        System.out.println("Queue: " + queue);
        System.out.println("Is queue empty? " + queue.isEmpty());
// false

        // Dequeue all elements
        queue.remove();
        queue.remove();

        System.out.println("Queue after removing all elements: "
+ queue);
        System.out.println("Is queue empty now? " +
queue.isEmpty()); // true
    }
}
```

```
// Is queue empty? true
// Queue: [Apple, Banana]
// Is queue empty? false
// Queue after removing all elements: []
// Is queue empty now? true
```

Trees

Implementing a Binary Tree

A **Binary Tree** is a hierarchical data structure where each node has **at most two children**, commonly referred to as the **left** and **right child**. Binary trees are the foundation for many tree-based structures like **BST, Heap, and Expression Trees**.

```
// Class representing a node in the binary tree
class TreeNode {
    int data;
    TreeNode left;
    TreeNode right;

    TreeNode(int data) {
        this.data = data;
        left = null;
        right = null;
    }
}

// Class representing the binary tree
class BinaryTree {
    TreeNode root;

    // Constructor
    BinaryTree() {
        root = null;
    }

    // Preorder Traversal (Root -> Left -> Right)
    void preorder(TreeNode node) {
        if (node == null) return;
```

```
        System.out.print(node.data + " ");
        preorder(node.left);
        preorder(node.right);
    }

    // Inorder Traversal (Left -> Root -> Right)
    void inorder(TreeNode node) {
        if (node == null) return;
        inorder(node.left);
        System.out.print(node.data + " ");
        inorder(node.right);
    }

    // Postorder Traversal (Left -> Right -> Root)
    void postorder(TreeNode node) {
        if (node == null) return;
        postorder(node.left);
        postorder(node.right);
        System.out.print(node.data + " ");
    }
}

// Main class to run the program
public class Main {
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();

        // Creating the binary tree manually
        tree.root = new TreeNode(1);
        tree.root.left = new TreeNode(2);
        tree.root.right = new TreeNode(3);
        tree.root.left.left = new TreeNode(4);
        tree.root.left.right = new TreeNode(5);

        System.out.print("Preorder traversal: ");
```

```
        tree.preorder(tree.root);
        System.out.println();

        System.out.print("Inorder traversal: ");
        tree.inorder(tree.root);
        System.out.println();

        System.out.print("Postorder traversal: ");
        tree.postorder(tree.root);
        System.out.println();
    }
}

// Preorder traversal: 1 2 4 5 3
// Inorder traversal: 4 2 5 1 3
// Postorder traversal: 4 5 2 3 1
```

Preorder Traversal (DFS)

Preorder Traversal is a **Depth-First Search (DFS)** method where nodes are visited in the order:

Root → Left Subtree → Right Subtree

It is commonly used for **copying a tree, prefix expressions, or serialization**.

```
// Class representing a node in the binary tree
class TreeNode {
    int data;
    TreeNode left, right;

    TreeNode(int data) {
        this.data = data;
        left = right = null;
    }
}

// Binary Tree class with preorder traversal
class BinaryTree {
    TreeNode root;

    BinaryTree() {
        root = null;
    }

    // Preorder Traversal (DFS)
    void preorder(TreeNode node) {
        if (node == null) return;
        System.out.print(node.data + " "); // Visit root
        preorder(node.left);               // Traverse left
subtree
```

```
        preorder(node.right);              // Traverse right
subtree
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();

        // Create binary tree
        tree.root = new TreeNode(1);
        tree.root.left = new TreeNode(2);
        tree.root.right = new TreeNode(3);
        tree.root.left.left = new TreeNode(4);
        tree.root.left.right = new TreeNode(5);

        System.out.print("Preorder traversal: ");
        tree.preorder(tree.root);
    }
}

// Preorder traversal: 1 2 4 5 3
```

Inorder Traversal (DFS)

Inorder Traversal is a **Depth-First Search (DFS)** technique where nodes are visited in the following order:

Left Subtree → Root → Right Subtree

This traversal is especially useful for **Binary Search Trees (BSTs)** because it returns nodes in **ascending sorted order**.

```
// Class representing a node in the binary tree
class TreeNode {
    int data;
    TreeNode left, right;

    TreeNode(int data) {
        this.data = data;
        left = right = null;
    }
}

// Binary Tree class with inorder traversal
class BinaryTree {
    TreeNode root;

    BinaryTree() {
        root = null;
    }

    // Inorder Traversal (DFS)
    void inorder(TreeNode node) {
        if (node == null) return;
        inorder(node.left);                 // Visit left subtree
        System.out.print(node.data + " "); // Visit root
        inorder(node.right);                // Visit right
subtree
    }
```

```
}

// Main class
public class Main {
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();

        // Creating the binary tree manually
        tree.root = new TreeNode(1);
        tree.root.left = new TreeNode(2);
        tree.root.right = new TreeNode(3);
        tree.root.left.left = new TreeNode(4);
        tree.root.left.right = new TreeNode(5);

        System.out.print("Inorder traversal: ");
        tree.inorder(tree.root);
    }
}

// Inorder traversal: 4 2 5 1 3
```

Postorder Traversal (DFS)

Postorder Traversal is a **Depth-First Search (DFS)** method where nodes are visited in the following order:

Left Subtree → Right Subtree → Root

This traversal is useful for tasks such as **deleting a tree, evaluating postfix expressions, or freeing memory in tree structures**.

```
// Class representing a node in the binary tree
class TreeNode {
    int data;
    TreeNode left, right;

    TreeNode(int data) {
        this.data = data;
        left = right = null;
    }
}

// Binary Tree class with postorder traversal
class BinaryTree {
    TreeNode root;

    BinaryTree() {
        root = null;
    }

    // Postorder Traversal (DFS)
    void postorder(TreeNode node) {
        if (node == null) return;
        postorder(node.left);               // Traverse left
subtree
        postorder(node.right);              // Traverse right
subtree
```

```
        System.out.print(node.data + " "); // Visit root
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();

        // Creating the binary tree manually
        tree.root = new TreeNode(1);
        tree.root.left = new TreeNode(2);
        tree.root.right = new TreeNode(3);
        tree.root.left.left = new TreeNode(4);
        tree.root.left.right = new TreeNode(5);

        System.out.print("Postorder traversal: ");
        tree.postorder(tree.root);
    }
}

// Postorder traversal: 4 5 2 3 1
```

Level Order Traversal (BFS)

Level Order Traversal is a **Breadth-First Search (BFS)** method where nodes are visited **level by level**, from top to bottom and from left to right at each level.

It is commonly used for printing trees by level, finding the height of a tree, or serialization.

```
import java.util.*;

// Class representing a node in the binary tree
class TreeNode {
    int data;
    TreeNode left, right;

    TreeNode(int data) {
        this.data = data;
        left = right = null;
    }
}

// Binary Tree class with Level Order Traversal
class BinaryTree {
    TreeNode root;

    BinaryTree() {
        root = null;
    }

    // Level Order Traversal (BFS)
    void levelOrder() {
        if (root == null) return;

        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
```

```
        while (!queue.isEmpty()) {
            TreeNode current = queue.poll();
            System.out.print(current.data + " ");

            if (current.left != null)
                queue.add(current.left);
            if (current.right != null)
                queue.add(current.right);
        }
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();

        // Creating the binary tree manually
        tree.root = new TreeNode(1);
        tree.root.left = new TreeNode(2);
        tree.root.right = new TreeNode(3);
        tree.root.left.left = new TreeNode(4);
        tree.root.left.right = new TreeNode(5);

        System.out.print("Level Order traversal: ");
        tree.levelOrder();
    }
}

// Level Order traversal: 1 2 3 4 5
```

Graphs

Representing a Graph with Adjacency List

An **Adjacency List** is a **space-efficient way** to represent a graph. Each vertex stores a **list of its neighbors**, making it ideal for **sparse graphs**.

- **Directed Graph:** Edge points from one vertex to another
- **Undirected Graph:** Edge connects both vertices bidirectionally

```
import java.util.*;

// Class representing a graph
class Graph {
    private int V; // number of vertices
    private LinkedList<Integer>[] adjList;

    // Constructor
    Graph(int vertices) {
        V = vertices;
        adjList = new LinkedList[V];
        for (int i = 0; i < V; i++) {
            adjList[i] = new LinkedList<>();
        }
    }

    // Add an edge (directed)
    void addEdge(int u, int v) {
        adjList[u].add(v);
    }

    // Add an edge (undirected)
    void addUndirectedEdge(int u, int v) {
```

```
        adjList[u].add(v);
        adjList[v].add(u);
    }

    // Print the adjacency list
    void printGraph() {
        for (int i = 0; i < V; i++) {
            System.out.print(i + " -> ");
            for (int neighbor : adjList[i]) {
                System.out.print(neighbor + " ");
            }
            System.out.println();
        }
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Graph g = new Graph(5);

        // Adding directed edges
        g.addEdge(0, 1);
        g.addEdge(0, 4);
        g.addEdge(1, 2);
        g.addEdge(1, 3);
        g.addEdge(1, 4);
        g.addEdge(2, 3);
        g.addEdge(3, 4);

        System.out.println("Adjacency List of Directed Graph:");
        g.printGraph();

        // Example for undirected graph
        Graph undirected = new Graph(5);
```

```
        undirected.addUndirectedEdge(0, 1);
        undirected.addUndirectedEdge(0, 4);
        undirected.addUndirectedEdge(1, 2);
        undirected.addUndirectedEdge(1, 3);
        undirected.addUndirectedEdge(1, 4);
        undirected.addUndirectedEdge(2, 3);
        undirected.addUndirectedEdge(3, 4);

        System.out.println("\nAdjacency List of Undirected
Graph:");
        undirected.printGraph();
    }
}

// Adjacency List of Directed Graph:
// 0 -> 1 4
// 1 -> 2 3 4
// 2 -> 3
// 3 -> 4
// 4 ->

// Adjacency List of Undirected Graph:
// 0 -> 1 4
// 1 -> 0 2 3 4
// 2 -> 1 3
// 3 -> 1 2 4
// 4 -> 0 1 3
```

Representing a Graph with Adjacency Matrix

An **Adjacency Matrix** represents a graph using a **2D array**. Each cell `(i, j)` indicates whether there is an edge from vertex `i` to vertex `j` (and optionally stores the weight).

- **Directed Graph:** `matrix[i][j] = 1` if edge from `i` to `j` exists
- **Undirected Graph:** `matrix[i][j] = matrix[j][i] = 1`

```
import java.util.*;

// Class representing a graph using adjacency matrix
class Graph {
    private int V; // number of vertices
    private int[][] adjMatrix;

    // Constructor
    Graph(int vertices) {
        V = vertices;
        adjMatrix = new int[V][V];
    }

    // Add a directed edge
    void addEdge(int u, int v) {
        adjMatrix[u][v] = 1;
    }

    // Add an undirected edge
    void addUndirectedEdge(int u, int v) {
        adjMatrix[u][v] = 1;
        adjMatrix[v][u] = 1;
    }

    // Print adjacency matrix
    void printMatrix() {
```

```
        System.out.println("Adjacency Matrix:");
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                System.out.print(adjMatrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Graph g = new Graph(5);

        // Add edges (directed)
        g.addEdge(0, 1);
        g.addEdge(0, 4);
        g.addEdge(1, 2);
        g.addEdge(1, 3);
        g.addEdge(1, 4);
        g.addEdge(2, 3);
        g.addEdge(3, 4);

        g.printMatrix();

        // Example for undirected graph
        Graph undirected = new Graph(5);
        undirected.addUndirectedEdge(0, 1);
        undirected.addUndirectedEdge(0, 4);
        undirected.addUndirectedEdge(1, 2);
        undirected.addUndirectedEdge(1, 3);
        undirected.addUndirectedEdge(1, 4);
        undirected.addUndirectedEdge(2, 3);
        undirected.addUndirectedEdge(3, 4);
```

```
        System.out.println("\nAdjacency Matrix of Undirected
Graph:");
        undirected.printMatrix();
    }
}

// Adjacency Matrix:
// 0 1 0 0 1
// 0 0 1 1 1
// 0 0 0 1 0
// 0 0 0 0 1
// 0 0 0 0 0

// Adjacency Matrix of Undirected Graph:
// Adjacency Matrix:
// 0 1 0 0 1
// 1 0 1 1 1
// 0 1 0 1 0
// 0 1 1 0 1
// 1 1 0 1 0
```

Adding and Removing Edges

Manipulating edges is fundamental when working with graph data structures. You can **add** or **remove** edges in both **adjacency lists** and **adjacency matrices**.

1. Using Adjacency List

```
import java.util.*;

class Graph {
    private int V;
    private LinkedList<Integer>[] adjList;

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

    // Add an edge (directed)
    void addEdge(int u, int v) {
        adjList[u].add(v);
    }

    // Remove an edge (directed)
    void removeEdge(int u, int v) {
        adjList[u].remove(Integer.valueOf(v));
    }

    void printGraph() {
        for (int i = 0; i < V; i++) {
            System.out.print(i + " -> ");
            for (int neighbor : adjList[i])
                System.out.print(neighbor + " ");
```

```
            System.out.println();
        }
    }
}

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

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

        System.out.println("Graph after adding edges:");
        g.printGraph();

        g.removeEdge(1, 3);
        System.out.println("\nGraph after removing edge 1->3:");
        g.printGraph();
    }
}


// 4 ->

// Graph after removing edge 1->3:
// 0 -> 1 4
// 1 -> 2
// 2 ->
// 3 ->
// 4 ->
```

2. Using Adjacency Matrix

```
class GraphMatrix {
```

```
    private int V;
    private int[][] adjMatrix;

    GraphMatrix(int vertices) {
        V = vertices;
        adjMatrix = new int[V][V];
    }

    // Add directed edge
    void addEdge(int u, int v) {
        adjMatrix[u][v] = 1;
    }

    // Remove directed edge
    void removeEdge(int u, int v) {
        adjMatrix[u][v] = 0;
    }

    void printMatrix() {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++)
                System.out.print(adjMatrix[i][j] + " ");
            System.out.println();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        GraphMatrix g = new GraphMatrix(5);

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

```
        System.out.println("Adjacency Matrix after adding
edges:");
        g.printMatrix();

        g.removeEdge(1, 3);
        System.out.println("\nAdjacency Matrix after removing
edge 1->3:");
        g.printMatrix();
    }
}

// Adjacency Matrix after adding edges:
// 0 1 0 0 1
// 0 0 1 1 0
// 0 0 0 0 0
// 0 0 0 0 0
// 0 0 0 0 0

// Adjacency Matrix after removing edge 1->3:
// 0 1 0 0 1
// 0 0 1 0 0
// 0 0 0 0 0
// 0 0 0 0 0
// 0 0 0 0 0
```

Depth-First Search (DFS)

Depth-First Search (DFS) is a **graph traversal algorithm** that explores as far as possible along each branch before backtracking.

- Can be implemented **recursively** or **using a stack**
- Useful for **connected components, topological sort, cycle detection, pathfinding**

1. DFS Using Adjacency List (Recursive)

```
import java.util.*;

class Graph {
    private int V;
    private LinkedList<Integer>[] adjList;

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

    void addEdge(int u, int v) {
        adjList[u].add(v);
    }

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

    private void dfsUtil(int node, boolean[] visited) {
        visited[node] = true;
        System.out.print(node + " ");
```

```
        for (int neighbor : adjList[node]) {
            if (!visited[neighbor]) {
                dfsUtil(neighbor, visited);
            }
        }
    }
}

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

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

        System.out.print("DFS starting from node 0: ");
        g.DFS(0);
    }
}

// DFS starting from node 0: 0 1 2 3 4
```