**Day 15 - 5th July 2025**

**Task 1:**

**Create a node for a tree and include a constructor.**

**Answer:**

**// Tree Node class with constructor**

**class TreeNode {**

**int data; // Data part**

**TreeNode left; // Left reference**

**TreeNode right; // Right reference**

**// Constructor for creating a new node**

**public TreeNode(int data) {**

**this.data = data;**

**this.left = null;**

**this.right = null;**

**}**

**// Alternative constructor with all parameters**

**public TreeNode(int data, TreeNode left, TreeNode right) {**

**this.data = data;**

**this.left = left;**

**this.right = right;**

**}**

**// Default constructor**

**public TreeNode() {**

**this.data = 0;**

**this.left = null;**

**this.right = null;**

**}**

**// Method to check if node is a leaf**

**public boolean isLeaf() {**

**return this.left == null && this.right == null;**

**}**

**// Method to get node information**

**@Override**

**public String toString() {**

**return "TreeNode{data=" + data +**

**", hasLeft=" + (left != null) +**

**", hasRight=" + (right != null) + "}";**

**}**

**}**

**// Generic Tree Node for different data types**

**class GenericTreeNode<T> {**

**T data;**

**GenericTreeNode<T> left;**

**GenericTreeNode<T> right;**

**public GenericTreeNode(T data) {**

**this.data = data;**

**this.left = null;**

**this.right = null;**

**}**

**public GenericTreeNode(T data, GenericTreeNode<T> left, GenericTreeNode<T> right) {**

**this.data = data;**

**this.left = left;**

**this.right = right;**

**}**

**}**

**Task 2:**

**Create a class named Binary Search Tree with 2 insert operations**

**Answer:**

**class BinarySearchTree {**

**TreeNode root;**

**// Constructor**

**public BinarySearchTree() {**

**this.root = null;**

**}**

**// Insert operation 1: For inserting when tree is empty**

**public void insertInEmptyTree(int data) {**

**if (root == null) {**

**root = new TreeNode(data);**

**System.out.println("Inserted " + data + " as root node");**

**} else {**

**System.out.println("Tree is not empty. Use insertNode method instead.");**

**}**

**}**

**// Insert operation 2: For inserting when tree has 1 or more nodes**

**public void insertNode(int data) {**

**if (root == null) {**

**root = new TreeNode(data);**

**System.out.println("Inserted " + data + " as root node");**

**} else {**

**insertRecursive(root, data);**

**}**

**}**

**// Recursive helper method for insertion**

**private TreeNode insertRecursive(TreeNode current, int data) {**

**// Base case: if current is null, create new node**

**if (current == null) {**

**System.out.println("Inserted " + data);**

**return new TreeNode(data);**

**}**

**// If data is less than current node, go to left subtree**

**if (data < current.data) {**

**current.left = insertRecursive(current.left, data);**

**}**

**// If data is greater than current node, go to right subtree**

**else if (data > current.data) {**

**current.right = insertRecursive(current.right, data);**

**}**

**// If data equals current node data, don't insert (no duplicates)**

**else {**

**System.out.println("Duplicate value " + data + " not inserted");**

**}**

**return current;**

**}**

**// Public insert method (wrapper)**

**public void insert(int data) {**

**root = insertRecursive(root, data);**

**}**

**// Method to check if tree is empty**

**public boolean isEmpty() {**

**return root == null;**

**}**

**// Method to get root**

**public TreeNode getRoot() {**

**return root;**

**}**

**// Method to display tree structure**

**public void displayTree() {**

**if (root == null) {**

**System.out.println("Tree is empty");**

**return;**

**}**

**System.out.println("Tree Structure:");**

**displayTreeHelper(root, "", true);**

**}**

**private void displayTreeHelper(TreeNode node, String prefix, boolean isLast) {**

**if (node != null) {**

**System.out.println(prefix + (isLast ? "└── " : "├── ") + node.data);**

**if (node.left != null || node.right != null) {**

**if (node.left != null) {**

**displayTreeHelper(node.left, prefix + (isLast ? " " : "│ "), node.right == null);**

**}**

**if (node.right != null) {**

**displayTreeHelper(node.right, prefix + (isLast ? " " : "│ "), true);**

**}**

**}**

**}**

**}**

**}**

**Task 3:**

**Inorder traversal of the above code snippets from Task 1 and Task 2**

**Answer:**

**// Add this method to the BinarySearchTree class**

**public class BinarySearchTree {**

**// ... previous code ...**

**// Inorder traversal (Left -> Root -> Right)**

**public void inorderTraversal() {**

**System.out.print("Inorder Traversal: ");**

**if (root == null) {**

**System.out.println("Tree is empty");**

**return;**

**}**

**inorderRecursive(root);**

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

**}**

**// Recursive helper for inorder traversal**

**private void inorderRecursive(TreeNode node) {**

**if (node != null) {**

**inorderRecursive(node.left); // Visit left subtree**

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

**inorderRecursive(node.right); // Visit right subtree**

**}**

**}**

**// Preorder traversal (Root -> Left -> Right)**

**public void preorderTraversal() {**

**System.out.print("Preorder Traversal: ");**

**if (root == null) {**

**System.out.println("Tree is empty");**

**return;**

**}**

**preorderRecursive(root);**

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

**}**

**private void preorderRecursive(TreeNode node) {**

**if (node != null) {**

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

**preorderRecursive(node.left); // Visit left subtree**

**preorderRecursive(node.right); // Visit right subtree**

**}**

**}**

**// Postorder traversal (Left -> Right -> Root)**

**public void postorderTraversal() {**

**System.out.print("Postorder Traversal: ");**

**if (root == null) {**

**System.out.println("Tree is empty");**

**return;**

**}**

**postorderRecursive(root);**

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

**}**

**private void postorderRecursive(TreeNode node) {**

**if (node != null) {**

**postorderRecursive(node.left); // Visit left subtree**

**postorderRecursive(node.right); // Visit right subtree**

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

**}**

**}**

**// Level order traversal (Breadth-First)**

**public void levelOrderTraversal() {**

**System.out.print("Level Order Traversal: ");**

**if (root == null) {**

**System.out.println("Tree is empty");**

**return;**

**}**

**java.util.Queue<TreeNode> queue = new java.util.LinkedList<>();**

**queue.offer(root);**

**while (!queue.isEmpty()) {**

**TreeNode current = queue.poll();**

**System.out.print(current.data + " ");**

**if (current.left != null) {**

**queue.offer(current.left);**

**}**

**if (current.right != null) {**

**queue.offer(current.right);**

**}**

**}**

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

**}**

**}**

**Task 4:**

**Create a main method for Task 1, 2 and 3 and run the code.**

**Answer:**

**public class BinaryTreeDemo {**

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

**System.out.println("=== Binary Search Tree Demo ===\n");**

**// Create a new Binary Search Tree**

**BinarySearchTree bst = new BinarySearchTree();**

**// Test inserting in empty tree**

**System.out.println("--- Testing Insert in Empty Tree ---");**

**bst.insertInEmptyTree(50);**

**bst.displayTree();**

**// Try to insert in empty tree again (should show message)**

**System.out.println("\n--- Trying to Insert in Non-Empty Tree using Empty Tree Method ---");**

**bst.insertInEmptyTree(30);**

**// Test inserting nodes when tree has nodes**

**System.out.println("\n--- Inserting Multiple Nodes ---");**

**int[] values = {30, 70, 20, 40, 60, 80, 10, 25, 35, 45};**

**for (int value : values) {**

**bst.insertNode(value);**

**}**

**// Display tree structure**

**System.out.println("\n--- Final Tree Structure ---");**

**bst.displayTree();**

**// Test all traversals**

**System.out.println("\n--- Tree Traversals ---");**

**bst.inorderTraversal(); // Should print in sorted order**

**bst.preorderTraversal(); // Root first**

**bst.postorderTraversal(); // Root last**

**bst.levelOrderTraversal(); // Level by level**

**// Test duplicate insertion**

**System.out.println("\n--- Testing Duplicate Insertion ---");**

**bst.insertNode(50); // Should not insert**

**bst.insertNode(30); // Should not insert**

**// Test with a new tree using generic insert method**

**System.out.println("\n--- Testing Generic Insert Method ---");**

**BinarySearchTree bst2 = new BinarySearchTree();**

**int[] newValues = {15, 10, 20, 8, 12, 17, 25};**

**for (int value : newValues) {**

**bst2.insert(value);**

**}**

**System.out.println("\nSecond Tree Structure:");**

**bst2.displayTree();**

**System.out.println("\nSecond Tree Traversals:");**

**bst2.inorderTraversal();**

**bst2.preorderTraversal();**

**bst2.postorderTraversal();**

**bst2.levelOrderTraversal();**

**// Test empty tree operations**

**System.out.println("\n--- Testing Empty Tree Operations ---");**

**BinarySearchTree emptyBST = new BinarySearchTree();**

**System.out.println("Is empty tree empty? " + emptyBST.isEmpty());**

**emptyBST.inorderTraversal();**

**emptyBST.displayTree();**

**// Demonstrate TreeNode usage**

**System.out.println("\n--- TreeNode Demo ---");**

**TreeNode node1 = new TreeNode(100);**

**TreeNode node2 = new TreeNode(200, null, null);**

**TreeNode node3 = new TreeNode();**

**System.out.println("Node 1: " + node1);**

**System.out.println("Node 2: " + node2);**

**System.out.println("Node 3: " + node3);**

**System.out.println("Is Node 1 a leaf? " + node1.isLeaf());**

**// Create a small tree manually**

**node1.left = new TreeNode(50);**

**node1.right = new TreeNode(150);**

**System.out.println("After adding children - Is Node 1 a leaf? " + node1.isLeaf());**

**System.out.println("Updated Node 1: " + node1);**

**}**

**}**

**/\* Output:**

**=== Binary Search Tree Demo ===**

**--- Testing Insert in Empty Tree ---**

**Inserted 50 as root node**

**Tree Structure:**

**└── 50**

**--- Trying to Insert in Non-Empty Tree using Empty Tree Method ---**

**Tree is not empty. Use insertNode method instead.**

**--- Inserting Multiple Nodes ---**

**Inserted 30**

**Inserted 70**

**Inserted 20**

**Inserted 40**

**Inserted 60**

**Inserted 80**

**Inserted 10**

**Inserted 25**

**Inserted 35**

**Inserted 45**

**--- Final Tree Structure ---**

**Tree Structure:**

**└── 50**

**├── 30**

**│ ├── 20**

**│ │ ├── 10**

**│ │ └── 25**

**│ └── 40**

**│ ├── 35**

**│ └── 45**

**└── 70**

**├── 60**

**└── 80**

**--- Tree Traversals ---**

**Inorder Traversal: 10 20 25 30 35 40 45 50 60 70 80**

**Preorder Traversal: 50 30 20 10 25 40 35 45 70 60 80**

**Postorder Traversal: 10 25 20 35 45 40 30 60 80 70 50**

**Level Order Traversal: 50 30 70 20 40 60 80 10 25 35 45**

**--- Testing Duplicate Insertion ---**

**Duplicate value 50 not inserted**

**Duplicate value 30 not inserted**

**--- Testing Generic Insert Method ---**

**Inserted 15 as root node**

**Inserted 10**

**Inserted 20**

**Inserted 8**

**Inserted 12**

**Inserted 17**

**Inserted 25**

**Second Tree Structure:**

**└── 15**

**├── 10**

**│ ├── 8**

**│ └── 12**

**└── 20**

**├── 17**

**└── 25**

**Second Tree Traversals:**

**Inorder Traversal: 8 10 12 15 17 20 25**

**Preorder Traversal: 15 10 8 12 20 17 25**

**Postorder Traversal: 8 12 10 17 25 20 15**

**Level Order Traversal: 15 10 20 8 12 17 25**

**--- Testing Empty Tree Operations ---**

**Is empty tree empty? true**

**Inorder Traversal: Tree is empty**

**Tree is empty**

**--- TreeNode Demo ---**

**Node 1: TreeNode{data=100, hasLeft=false, hasRight=false}**

**Node 2: TreeNode{data=200, hasLeft=false, hasRight=false}**

**Node 3: TreeNode{data=0, hasLeft=false, hasRight=false}**

**Is Node 1 a leaf? true**

**After adding children - Is Node 1 a leaf? false**

**Updated Node 1: TreeNode{data=100, hasLeft=true, hasRight=true}**

**\*/**

**Task 5:**

**Applications of Trees**

**Answer:**

**Applications of Trees**

**1. File Systems**

* **Directory Structure: Folders and subfolders in operating systems**
* **Example: Windows Explorer, Mac Finder, Linux file system**
* **Use: Hierarchical organization of files and directories**

**2. Database Systems**

* **B-Trees: Used in database indexing for fast data retrieval**
* **B+ Trees: Used in database management systems (MySQL, PostgreSQL)**
* **Example: Index structures for efficient querying**

**3. Web Development**

* **DOM (Document Object Model): HTML structure representation**
* **XML Parsing: Hierarchical data representation**
* **React Component Tree: Component hierarchy in React applications**

**4. Artificial Intelligence**

* **Decision Trees: Machine learning algorithms for classification**
* **Game Trees: Chess, checkers, tic-tac-toe game strategies**
* **Parse Trees: Natural language processing**

**5. Networking**

* **Routing Algorithms: Network packet routing**
* **Spanning Trees: Network topology in switches**
* **Multicast Trees: Efficient data broadcasting**

**6. Expression Evaluation**

* **Syntax Trees: Compiler design for parsing expressions**
* **Abstract Syntax Trees (AST): Programming language interpreters**
* **Mathematical Expressions: Calculator applications**

**7. Data Compression**

* **Huffman Trees: Lossless data compression algorithms**
* **File Compression: ZIP, GZIP compression techniques**

**8. Graphics and Gaming**

* **Scene Graphs: 3D graphics rendering**
* **Quadtrees/Octrees: Spatial partitioning in games**
* **Animation Trees: Character animation systems**

**9. Search Operations**

* **Binary Search Trees: Fast searching and sorting**
* **Trie (Prefix Trees): Autocomplete, spell checkers**
* **Segment Trees: Range query operations**

**10. Operating Systems**

* **Process Trees: Parent-child process relationships**
* **Memory Management: Heap organization**
* **Thread Scheduling: Task management**

**// Example: Simple File System Tree**

**class FileSystemNode {**

**String name;**

**boolean isDirectory;**

**List<FileSystemNode> children;**

**public FileSystemNode(String name, boolean isDirectory) {**

**this.name = name;**

**this.isDirectory = isDirectory;**

**this.children = isDirectory ? new ArrayList<>() : null;**

**}**

**public void addChild(FileSystemNode child) {**

**if (isDirectory && children != null) {**

**children.add(child);**

**}**

**}**

**public void displayStructure(String prefix) {**

**System.out.println(prefix + (isDirectory ? "📁 " : "📄 ") + name);**

**if (children != null) {**

**for (FileSystemNode child : children) {**

**child.displayStructure(prefix + " ");**

**}**

**}**

**}**

**}**

**// Usage Example**

**public class FileSystemExample {**

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

**FileSystemNode root = new FileSystemNode("C:", true);**

**FileSystemNode users = new FileSystemNode("Users", true);**

**FileSystemNode programs = new FileSystemNode("Program Files", true);**

**FileSystemNode john = new FileSystemNode("John", true);**

**FileSystemNode documents = new FileSystemNode("Documents", true);**

**FileSystemNode resume = new FileSystemNode("resume.pdf", false);**

**root.addChild(users);**

**root.addChild(programs);**

**users.addChild(john);**

**john.addChild(documents);**

**documents.addChild(resume);**

**root.displayStructure("");**

**}**

**}**

**Task 6:**

**Create a binary search operation on tree**

**Answer:**

**// Add these methods to the BinarySearchTree class**

**public class BinarySearchTree {**

**// ... previous code ...**

**// Binary search operation - search for a value**

**public boolean search(int data) {**

**return searchRecursive(root, data);**

**}**

**// Recursive helper for search**

**private boolean searchRecursive(TreeNode node, int data) {**

**// Base case: if node is null, element not found**

**if (node == null) {**

**return false;**

**}**

**// If data matches current node**

**if (data == node.data) {**

**return true;**

**}**

**// If data is less than current node, search left subtree**

**if (data < node.data) {**

**return searchRecursive(node.left, data);**

**}**

**// If data is greater than current node, search right subtree**

**return searchRecursive(node.right, data);**

**}**

**// Search with path tracking**

**public boolean searchWithPath(int data) {**

**System.out.print("Searching for " + data + ": ");**

**boolean found = searchWithPathHelper(root, data, "");**

**System.out.println(found ? " -> Found!" : " -> Not Found!");**

**return found;**

**}**

**private boolean searchWithPathHelper(TreeNode node, int data, String path) {**

**// Base case: if node is null**

**if (node == null) {**

**return false;**

**}**

**// Print current node**

**System.out.print(node.data);**

**// If data matches current node**

**if (data == node.data) {**

**return true;**

**}**

**// If data is less than current node, go left**

**if (data < node.data) {**

**System.out.print(" -> ");**

**return searchWithPathHelper(node.left, data, path + "L");**

**}**

**// If data is greater than current node, go right**

**System.out.print(" -> ");**

**return searchWithPathHelper(node.right, data, path + "R");**

**}**

**// Find minimum value in tree**

**public int findMin() {**

**if (root == null) {**

**throw new RuntimeException("Tree is empty");**

**}**

**return findMinRecursive(root);**

**}**

**private int findMinRecursive(TreeNode node) {**

**// Keep going left until you find the leftmost node**

**if (node.left == null) {**

**return node.data;**

**}**

**return findMinRecursive(node.left);**

**}**

**// Find maximum value in tree**

**public int findMax() {**

**if (root == null) {**

**throw new RuntimeException("Tree is empty");**

**}**

**return findMaxRecursive(root);**

**}**

**private int findMaxRecursive(TreeNode node) {**

**// Keep going right until you find the rightmost node**

**if (node.right == null) {**

**return node.data;**

**}**

**return findMaxRecursive(node.right);**

**}**

**// Find height of tree**

**public int getHeight() {**

**return getHeightRecursive(root);**

**}**

**private int getHeightRecursive(TreeNode node) {**

**if (node == null) {**

**return -1; // Height of empty tree is -1**

**}**

**int leftHeight = getHeightRecursive(node.left);**

**int rightHeight = getHeightRecursive(node.right);**

**return Math.max(leftHeight, rightHeight) + 1;**

**}**

**// Count total nodes in tree**

**public int countNodes() {**

**return countNodesRecursive(root);**

**}**

**private int countNodesRecursive(TreeNode node) {**

**if (node == null) {**

**return 0;**

**}**

**return 1 + countNodesRecursive(node.left) + countNodesRecursive(node.right);**

**}**

**// Delete a node from BST**

**public void delete(int data) {**

**root = deleteRecursive(root, data);**

**}**

**private TreeNode deleteRecursive(TreeNode node, int data) {**

**// Base case: if tree is empty**

**if (node == null) {**

**System.out.println("Value " + data + " not found for deletion");**

**return node;**

**}**

**// If data is less than current node, go to left subtree**

**if (data < node.data) {**

**node.left = deleteRecursive(node.left, data);**

**}**

**// If data is greater than current node, go to right subtree**

**else if (data > node.data) {**

**node.right = deleteRecursive(node.right, data);**

**}**

**// If data equals current node, this is the node to be deleted**

**else {**

**System.out.println("Deleting node with value: " + data);**

**// Case 1: Node has no children (leaf node)**

**if (node.left == null && node.right == null) {**

**return null;**

**}**

**// Case 2: Node has only right child**

**else if (node.left == null) {**

**return node.right;**

**}**

**// Case 3: Node has only left child**

**else if (node.right == null) {**

**return node.left;**

**}**

**// Case 4: Node has both children**

**else {**

**// Find inorder successor (minimum value in right subtree)**

**int minValue = findMinRecursive(node.right);**

**node.data = minValue; // Replace with successor's value**

**// Delete the successor**

**node.right = deleteRecursive(node.right, minValue);**

**}**

**}**

**return node;**

**}**

**// Main method to demonstrate binary search operations**

**public static void demonstrateSearchOperations() {**

**System.out.println("=== Binary Search Tree Operations Demo ===\n");**

**BinarySearchTree bst = new BinarySearchTree();**

**// Insert values**

**int[] values = {50, 30, 70, 20, 40, 60, 80, 10, 25, 35, 45};**

**System.out.println("Inserting values: " + java.util.Arrays.toString(values));**

**for (int value : values) {**

**bst.insert(value);**

**}**

**System.out.println("\nTree Structure:");**

**bst.displayTree();**

**System.out.println("\nInorder Traversal (should be sorted):");**

**bst.inorderTraversal();**

**// Search operations**

**System.out.println("\n--- Search Operations ---");**

**int[] searchValues = {45, 25, 100, 50, 5};**

**for (int value : searchValues) {**

**boolean found = bst.search(value);**

**System.out.println("Search " + value + ": " + (found ? "Found" : "Not Found"));**

**}**

**// Search with path**

**System.out.println("\n--- Search with Path ---");**

**for (int value : new int[]{45, 25, 100}) {**

**bst.searchWithPath(value);**

**}**

**// Tree statistics**

**System.out.println("\n--- Tree Statistics ---");**

**System.out.println("Minimum value: " + bst.findMin());**

**System.out.println("Maximum value: " + bst.findMax());**

**System.out.println("Tree height: " + bst.getHeight());**

**System.out.println("Total nodes: " + bst.countNodes());**

**// Delete operations**

**System.out.println("\n--- Delete Operations ---");**

**System.out.println("Deleting leaf node (10):");**

**bst.delete(10);**

**bst.inorderTraversal();**

**System.out.println("\nDeleting node with one child (20):");**

**bst.delete(20);**

**bst.inorderTraversal();**

**System.out.println("\nDeleting node with two children (30):");**

**bst.delete(30);**

**bst.inorderTraversal();**

**System.out.println("\nFinal tree structure:");**

**bst.displayTree();**

**}**

**}**

**// Test the search operations**

**public class BinarySearchDemo {**

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

**BinarySearchTree.demonstrateSearchOperations();**

**}**

**}**

**Task 7:**

**Types of binary trees:**

**Answer:**

**Types of Binary Trees**

**1. Full Binary Tree (Strict Binary Tree)**

* **Definition: Every node has either 0 or 2 children (no node has only 1 child)**
* **Properties: All internal nodes have exactly 2 children; leaf nodes have 0 children**

**class FullBinaryTree {**

**// Example of Full Binary Tree**

**// 1**

**// / \**

**// 2 3**

**// / \ / \**

**// 4 5 6 7**

**public static TreeNode createFullBinaryTree() {**

**TreeNode root = new TreeNode(1);**

**root.left = new TreeNode(2);**

**root.right = new TreeNode(3);**

**root.left.left = new TreeNode(4);**

**root.left.right = new TreeNode(5);**

**root.right.left = new TreeNode(6);**

**root.right.right = new TreeNode(7);**

**return root;**

**}**

**// Check if tree is full binary tree**

**public static boolean isFullBinaryTree(TreeNode node) {**

**if (node == null) return true;**

**// If leaf node**

**if (node.left == null && node.right == null) {**

**return true;**

**}**

**// If both children exist**

**if (node.left != null && node.right != null) {**

**return isFullBinaryTree(node.left) && isFullBinaryTree(node.right);**

**}**

**// If only one child exists**

**return false;**

**}**

**}**

**2. Complete Binary Tree**

* **Definition: All levels are completely filled except possibly the last level, which is filled from left to right**
* **Properties: Used in heap data structures**

**class CompleteBinaryTree {**

**// Example of Complete Binary Tree**

**// 1**

**// / \**

**// 2 3**

**// / \ /**

**// 4 5 6**

**public static boolean isCompleteBinaryTree(TreeNode root) {**

**if (root == null) return true;**

**java.util.Queue<TreeNode> queue = new java.util.LinkedList<>();**

**queue.offer(root);**

**boolean flag = false;**

**while (!queue.isEmpty()) {**

**TreeNode current = queue.poll();**

**if (current.left != null) {**

**if (flag) return false; // Found a node after a missing node**

**queue.offer(current.left);**

**} else {**

**flag = true; // Mark that we found a missing left child**

**}**

**if (current.right != null) {**

**if (flag) return false; // Found a node after a missing node**

**queue.offer(current.right);**

**} else {**

**flag = true; // Mark that we found a missing right child**

**}**

**}**

**return true;**

**}**

**}**

**3. Perfect Binary Tree**

* **Definition: All internal nodes have exactly 2 children and all leaf nodes are at the same level**
* **Properties: Has exactly 2^h - 1 nodes where h is height**

**class PerfectBinaryTree {**

**// Example of Perfect Binary Tree**

**// 1**

**// / \**

**// 2 3**

**// / \ / \**

**// 4 5 6 7**

**public static boolean isPerfectBinaryTree(TreeNode root) {**

**int depth = getDepth(root);**

**return isPerfectRecursive(root, depth, 0);**

**}**

**private static int getDepth(TreeNode node) {**

**int depth = 0;**

**while (node != null) {**

**depth++;**

**node = node.left;**

**}**

**return depth;**

**}**

**private static boolean isPerfectRecursive(TreeNode node, int depth, int level) {**

**if (node == null) return true;**

**// If leaf node, check if it's at the correct depth**

**if (node.left == null && node.right == null) {**

**return depth == level + 1;**

**}**

**// If one child is missing, it's not perfect**

**if (node.left == null || node.right == null) {**

**return false;**

**}**

**return isPerfectRecursive(node.left, depth, level + 1) &&**

**isPerfectRecursive(node.right, depth, level + 1);**

**}**

**}**

**4. Balanced Binary Tree**

* **Definition: Height difference between left and right subtrees is at most 1 for every node**
* **Properties: Ensures O(log n) operations**

**class BalancedBinaryTree {**

**public static boolean isBalanced(TreeNode root) {**

**return checkBalance(root) != -1;**

**}**

**private static int checkBalance(TreeNode node) {**

**if (node == null) return 0;**

**int leftHeight = checkBalance(node.left);**

**if (leftHeight == -1) return -1;**

**int rightHeight = checkBalance(node.right);**

**if (rightHeight == -1) return -1;**

**if (Math.abs(leftHeight - rightHeight) > 1) {**

**return -1; // Not balanced**

**}**

**return Math.max(leftHeight, rightHeight) + 1;**

**}**

**}**

**5. Degenerate/Pathological Binary Tree**

* **Definition: Each parent node has only one child, essentially forming a linked list**
* **Properties: Performance degrades to O(n) for search operations**

**class DegenerateBinaryTree {**

**// Example of Degenerate Tree (Left-skewed)**

**// 1**

**// /**

**// 2**

**///**

**//3**

**public static boolean isDegenerateTree(TreeNode root) {**

**if (root == null) return true;**

**return isDegenerateHelper(root);**

**}**

**private static boolean isDegenerateHelper(TreeNode node) {**

**if (node == null) return true;**

**// If it's a leaf node**

**if (node.left == null && node.right == null) {**

**return true;**

**}**

**// If it has both children, it's not degenerate**

**if (node.left != null && node.right != null) {**

**return false;**

**}**

**// If it has only one child, continue checking**

**if (node.left != null) {**

**return isDegenerateHelper(node.left);**

**} else {**

**return isDegenerateHelper(node.right);**

**}**

**}**

**}**

**6. Binary Search Tree (BST)**

* **Definition: Left subtree contains values less than root, right subtree contains values greater than root**
* **Properties: Enables efficient searching, insertion, and deletion**

**// Already implemented in previous tasks**

**7. AVL Tree**

* **Definition: Self-balancing BST where height difference between subtrees is at most 1**
* **Properties: Guarantees O(log n) operations through rotations**

**8. Red-Black Tree**

* **Definition: Self-balancing BST with additional color property**
* **Properties: Used in Java's TreeMap and TreeSet**

**Demonstration of Different Tree Types**

**public class BinaryTreeTypes {**

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

**System.out.println("=== Binary Tree Types Demonstration ===\n");**

**// Full Binary Tree**

**System.out.println("--- Full Binary Tree ---");**

**TreeNode fullTree = FullBinaryTree.createFullBinaryTree();**

**System.out.println("Is Full Binary Tree: " + FullBinaryTree.isFullBinaryTree(fullTree));**

**// Complete Binary Tree Test**

**System.out.println("\n--- Complete Binary Tree ---");**

**TreeNode completeTree = new TreeNode(1);**

**completeTree.left = new TreeNode(2);**

**completeTree.right = new TreeNode(3);**

**completeTree.left.left = new TreeNode(4);**

**completeTree.left.right = new TreeNode(5);**

**completeTree.right.left = new TreeNode(6);**

**System.out.println("Is Complete Binary Tree: " +**

**CompleteBinaryTree.isCompleteBinaryTree(completeTree));**

**// Perfect Binary Tree Test**

**System.out.println("\n--- Perfect Binary Tree ---");**

**System.out.println("Is Perfect Binary Tree: " +**

**PerfectBinaryTree.isPerfectBinaryTree(fullTree));**

**// Balanced Binary Tree Test**

**System.out.println("\n--- Balanced Binary Tree ---");**

**TreeNode balancedTree = new TreeNode(1);**

**balancedTree.left = new TreeNode(2);**

**balancedTree.right = new TreeNode(3);**

**balancedTree.left.left = new TreeNode(4);**

**balancedTree.left.right = new TreeNode(5);**

**System.out.println("Is Balanced Binary Tree: " +**

**BalancedBinaryTree.isBalanced(balancedTree));**

**// Degenerate Tree Test**

**System.out.println("\n--- Degenerate Binary Tree ---");**

**TreeNode degenerateTree = new TreeNode(1);**

**degenerateTree.left = new TreeNode(2);**

**degenerateTree.left.left = new TreeNode(3);**

**System.out.println("Is Degenerate Tree: " +**

**DegenerateBinaryTree.isDegenerateTree(degenerateTree));**

**}**

**}**

**===============================================================================================================================================**

**Graphs**

**===============================================================================================================================================**

**Task 8:**

**Applications of Graphs**

**Answer:**

**Applications of Graphs**

**1. Social Networks**

* **Facebook: Friend connections, mutual friends**
* **LinkedIn: Professional networks, connections**
* **Twitter: Follow relationships, recommendation systems**
* **Instagram: User interactions, hashtag networks**

**2. Transportation and Navigation**

* **GPS Navigation: Road networks, shortest path algorithms**
* **Flight Networks: Airlines route planning, connecting flights**
* **Public Transport: Bus/train route optimization**
* **Traffic Management: Traffic flow analysis, congestion control**

**3. Computer Networks**

* **Internet Topology: Router connections, data packet routing**
* **Network Security: Firewall rules, intrusion detection**
* **Load Balancing: Server network optimization**
* **P2P Networks: BitTorrent, file sharing systems**

**4. Web and Internet**

* **Web Crawling: Search engine indexing (Google PageRank)**
* **Hyperlink Structure: Website linking and SEO**
* **Content Delivery Networks: Optimal server selection**
* **Web Graph Analysis: Link analysis algorithms**

**5. Biological Networks**

* **Protein Interaction: Molecular biology research**
* **Gene Networks: Genetic research and analysis**
* **Neural Networks: Brain structure modeling**
* **Epidemic Modeling: Disease spread simulation**

**6. Economics and Finance**

* **Trade Networks: International trade relationships**
* **Financial Networks: Banking system connections**
* **Supply Chain: Manufacturing and distribution networks**
* **Market Analysis: Stock correlation networks**

**7. Game Development**

* **Pathfinding: AI movement in games (A\* algorithm)**
* **Game State Representation: Chess, checkers game trees**
* **Level Design: Game world connectivity**
* **NPC Behavior: Non-player character interactions**

**8. Database and Data Structures**

* **Database Relationships: Foreign key connections**
* **Dependency Graphs: Software module dependencies**
* **Version Control: Git commit history, branching**
* **Data Lineage: Data flow tracking in ETL processes**

**9. Scheduling and Resource Allocation**

* **Task Scheduling: Project management (Critical Path Method)**
* **Resource Allocation: CPU scheduling, memory management**
* **Course Scheduling: University timetabling**
* **Job Scheduling: Manufacturing process optimization**

**10. Recommendation Systems**

* **E-commerce: Product recommendations (Amazon, eBay)**
* **Streaming Services: Content recommendation (Netflix, Spotify)**
* **News Feed: Social media content curation**
* **Collaborative Filtering: User-item relationship analysis**

**// Example: Social Network Graph**

**import java.util.\*;**

**class SocialNetwork {**

**private Map<String, List<String>> adjacencyList;**

**public SocialNetwork() {**

**adjacencyList = new HashMap<>();**

**}**

**// Add user to network**

**public void addUser(String user) {**

**adjacencyList.putIfAbsent(user, new ArrayList<>());**

**}**

**// Add friendship (bidirectional)**

**public void addFriendship(String user1, String user2) {**

**addUser(user1);**

**addUser(user2);**

**adjacencyList.get(user1).add(user2);**

**adjacencyList.get(user2).add(user1);**

**}**

**// Find mutual friends**

**public List<String> findMutualFriends(String user1, String user2) {**

**List<String> friends1 = adjacencyList.get(user1);**

**List<String> friends2 = adjacencyList.get(user2);**

**List<String> mutualFriends = new ArrayList<>();**

**if (friends1 != null && friends2 != null) {**

**for (String friend : friends1) {**

**if (friends2.contains(friend)) {**

**mutualFriends.add(friend);**

**}**

**}**

**}**

**return mutualFriends;**

**}**

**// Find shortest path between users (degrees of separation)**

**public int shortestPath(String start, String end) {**

**if (!adjacencyList.containsKey(start) || !adjacencyList.containsKey(end)) {**

**return -1;**

**}**

**Queue<String> queue = new LinkedList<>();**

**Set<String> visited = new HashSet<>();**

**Map<String, Integer> distance = new HashMap<>();**

**queue.offer(start);**

**visited.add(start);**

**distance.put(start, 0);**

**while (!queue.isEmpty()) {**

**String current = queue.poll();**

**if (current.equals(end)) {**

**return distance.get(current);**

**}**

**for (String neighbor : adjacencyList.get(current)) {**

**if (!visited.contains(neighbor)) {**

**visited.add(neighbor);**

**distance.put(neighbor, distance.get(current) + 1);**

**queue.offer(neighbor);**

**}**

**}**

**}**

**return -1; // No connection found**

**}**

**// Display network**

**public void displayNetwork() {**

**System.out.println("Social Network:");**

**for (String user : adjacencyList.keySet()) {**

**System.out.println(user + " -> " + adjacencyList.get(user));**

**}**

**}**

**}**

**// Usage Example**

**public class SocialNetworkExample {**

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

**SocialNetwork network = new SocialNetwork();**

**// Add friendships**

**network.addFriendship("Alice", "Bob");**

**network.addFriendship("Bob", "Charlie");**

**network.addFriendship("Alice", "Diana");**

**network.addFriendship("Charlie", "Eve");**

**network.addFriendship("Diana", "Frank");**

**network.displayNetwork();**

**System.out.println("\nMutual friends of Alice and Bob: " +**

**network.findMutualFriends("Alice", "Bob"));**

**System.out.println("Degrees of separation between Alice and Eve: " +**

**network.shortestPath("Alice", "Eve"));**

**}**

**}**

**Task 9:**

**Types of Graphs:**

**Answer:**

**Types of Graphs**

**1. Based on Direction**

**A. Directed Graph (Digraph)**

* **Definition: Edges have direction (one-way)**
* **Examples: Web links, social media follows, traffic flow**

**class DirectedGraph {**

**private Map<Integer, List<Integer>> adjacencyList;**

**public DirectedGraph() {**

**adjacencyList = new HashMap<>();**

**}**

**public void addVertex(int vertex) {**

**adjacencyList.putIfAbsent(vertex, new ArrayList<>());**

**}**

**// Add directed edge from source to destination**

**public void addEdge(int src, int dest) {**

**addVertex(src);**

**addVertex(dest);**

**adjacencyList.get(src).add(dest);**

**}**

**public void display() {**

**System.out.println("Directed Graph:");**

**for (int vertex : adjacencyList.keySet()) {**

**System.out.println(vertex + " -> " + adjacencyList.get(vertex));**

**}**

**}**

**}**

**B. Undirected Graph**

* **Definition: Edges have no direction (two-way)**
* **Examples: Friendship networks, road networks**

**class UndirectedGraph {**

**private Map<Integer, List<Integer>> adjacencyList;**

**public UndirectedGraph() {**

**adjacencyList = new HashMap<>();**

**}**

**public void addVertex(int vertex) {**

**adjacencyList.putIfAbsent(vertex, new ArrayList<>());**

**}**

**// Add undirected edge between two vertices**

**public void addEdge(int vertex1, int vertex2) {**

**addVertex(vertex1);**

**addVertex(vertex2);**

**adjacencyList.get(vertex1).add(vertex2);**

**adjacencyList.get(vertex2).add(vertex1);**

**}**

**public void display() {**

**System.out.println("Undirected Graph:");**

**for (int vertex : adjacencyList.keySet()) {**

**System.out.println(vertex + " -> " + adjacencyList.get(vertex));**

**}**

**}**

**}**

**2. Based on Weights**

**A. Weighted Graph**

* **Definition: Edges have weights/costs associated with them**
* **Examples: Road networks with distances, network latency**

**class WeightedGraph {**

**class Edge {**

**int destination;**

**int weight;**

**public Edge(int destination, int weight) {**

**this.destination = destination;**

**this.weight = weight;**

**}**

**@Override**

**public String toString() {**

**return "(" + destination + ", w:" + weight + ")";**

**}**

**}**

**private Map<Integer, List<Edge>> adjacencyList;**

**public WeightedGraph() {**

**adjacencyList = new HashMap<>();**

**}**

**public void addVertex(int vertex) {**

**adjacencyList.putIfAbsent(vertex, new ArrayList<>());**

**}**

**public void addEdge(int src, int dest, int weight) {**

**addVertex(src);**

**addVertex(dest);**

**adjacencyList.get(src).add(new Edge(dest, weight));**

**adjacencyList.get(dest).add(new Edge(src, weight)); // For undirected**

**}**

**public void display() {**

**System.out.println("Weighted Graph:");**

**for (int vertex : adjacencyList.keySet()) {**

**System.out.println(vertex + " -> " + adjacencyList.get(vertex));**

**}**

**}**

**}**

**B. Unweighted Graph**

* **Definition: All edges have equal weight (usually 1)**
* **Examples: Simple friendship networks, basic connectivity**

**3. Based on Connectivity**

**A. Connected Graph**

* **Definition: There's a path between every pair of vertices**
* **Property: All vertices are reachable from any vertex**

**B. Disconnected Graph**

* **Definition: At least one pair of vertices has no path between them**
* **Property: Contains isolated components**

**class ConnectivityChecker {**

**public static boolean isConnected(UndirectedGraph graph) {**

**Map<Integer, List<Integer>> adjList = graph.adjacencyList;**

**if (adjList.isEmpty()) return true;**

**Set<Integer> visited = new HashSet<>();**

**Queue<Integer> queue = new LinkedList<>();**

**// Start DFS from first vertex**

**Integer startVertex = adjList.keySet().iterator().next();**

**queue.offer(startVertex);**

**visited.add(startVertex);**

**while (!queue.isEmpty()) {**

**Integer vertex = queue.poll();**

**for (Integer neighbor : adjList.get(vertex)) {**

**if (!visited.contains(neighbor)) {**

**visited.add(neighbor);**

**queue.offer(neighbor);**

**}**

**}**

**}**

**// Check if all vertices were visited**

**return visited.size() == adjList.size();**

**}**

**}**

**4. Special Types**

**A. Cyclic Graph**

* **Definition: Contains at least one cycle**
* **Detection: Using DFS with color coding**

**class CycleDetector {**

**public static boolean hasCycle(DirectedGraph graph) {**

**Set<Integer> white = new HashSet<>(); // Unvisited**

**Set<Integer> gray = new HashSet<>(); // Currently processing**

**Set<Integer> black = new HashSet<>(); // Processed**

**// Add all vertices to white set**

**for (Integer vertex : graph.adjacencyList.keySet()) {**

**white.add(vertex);**

**}**

**// Check each unvisited vertex**

**for (Integer vertex : white) {**

**if (dfs(vertex, white, gray, black, graph)) {**

**return true;**

**}**

**}**

**return false;**

**}**

**private static boolean dfs(Integer vertex, Set<Integer> white,**

**Set<Integer> gray, Set<Integer> black,**

**DirectedGraph graph) {**

**white.remove(vertex);**

**gray.add(vertex);**

**for (Integer neighbor : graph.adjacencyList.get(vertex)) {**

**if (black.contains(neighbor)) {**

**continue; // Already processed**

**}**

**if (gray.contains(neighbor)) {**

**return true; // Back edge found - cycle detected**

**}**

**if (dfs(neighbor, white, gray, black, graph)) {**

**return true;**

**}**

**}**

**gray.remove(vertex);**

**black.add(vertex);**

**return false;**

**}**

**}**

**B. Acyclic Graph**

* **Definition: Contains no cycles**
* **DAG: Directed Acyclic Graph (used in scheduling, dependency resolution)**

**C. Complete Graph**

* **Definition: Every pair of vertices is connected by an edge**
* **Property: Has n(n-1)/2 edges for n vertices**

**class CompleteGraph {**

**public static UndirectedGraph createCompleteGraph(int n) {**

**UndirectedGraph graph = new UndirectedGraph();**

**// Add all vertices**

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

**graph.addVertex(i);**

**}**

**// Connect every pair of vertices**

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

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

**graph.addEdge(i, j);**

**}**

**}**

**return graph;**

**}**

**}**

**D. Bipartite Graph**

* **Definition: Vertices can be divided into two disjoint sets such that edges only connect vertices from different sets**
* **Examples: Job matching, scheduling problems**

**class BipartiteChecker {**

**public static boolean isBipartite(UndirectedGraph graph) {**

**Map<Integer, Integer> colors = new HashMap<>();**

**for (Integer vertex : graph.adjacencyList.keySet()) {**

**if (!colors.containsKey(vertex)) {**

**if (!bfsColor(vertex, graph, colors)) {**

**return false;**

**}**

**}**

**}**

**return true;**

**}**

**private static boolean bfsColor(Integer start, UndirectedGraph graph,**

**Map<Integer, Integer> colors) {**

**Queue<Integer> queue = new LinkedList<>();**

**queue.offer(start);**

**colors.put(start, 0);**

**while (!queue.isEmpty()) {**

**Integer vertex = queue.poll();**

**int currentColor = colors.get(vertex);**

**for (Integer neighbor : graph.adjacencyList.get(vertex)) {**

**if (!colors.containsKey(neighbor)) {**

**colors.put(neighbor, 1 - currentColor);**

**queue.offer(neighbor);**

**} else if (colors.get(neighbor) == currentColor) {**

**return false; // Same color as neighbor - not bipartite**

**}**

**}**

**}**

**return true;**

**}**

**}**

**Demonstration of Graph Types**

**public class GraphTypesDemo {**

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

**System.out.println("=== Graph Types Demonstration ===\n");**

**// Directed Graph**

**System.out.println("--- Directed Graph ---");**

**DirectedGraph digraph = new DirectedGraph();**

**digraph.addEdge(1, 2);**

**digraph.addEdge(2, 3);**

**digraph.addEdge(3, 1);**

**digraph.display();**

**// Undirected Graph**

**System.out.println("\n--- Undirected Graph ---");**

**UndirectedGraph undirected = new UndirectedGraph();**

**undirected.addEdge(1, 2);**

**undirected.addEdge(2, 3);**

**undirected.addEdge(3, 4);**

**undirected.display();**

**// Weighted Graph**

**System.out.println("\n--- Weighted Graph ---");**

**WeightedGraph weighted = new WeightedGraph();**

**weighted.addEdge(1, 2, 10);**

**weighted.addEdge(2, 3, 20);**

**weighted.addEdge(1, 3, 15);**

**weighted.display();**

**// Complete Graph**

**System.out.println("\n--- Complete Graph (4 vertices) ---");**

**UndirectedGraph complete = CompleteGraph.createCompleteGraph(4);**

**complete.display();**

**// Connectivity Check**

**System.out.println("\n--- Connectivity Check ---");**

**System.out.println("Is undirected graph connected? " +**

**ConnectivityChecker.isConnected(undirected));**

**// Bipartite Check**

**System.out.println("\n--- Bipartite Check ---");**

**UndirectedGraph bipartite = new UndirectedGraph();**

**bipartite.addEdge(1, 3);**

**bipartite.addEdge(1, 4);**

**bipartite.addEdge(2, 3);**

**bipartite.addEdge(2, 4);**

**System.out.println("Is graph bipartite? " +**

**BipartiteChecker.isBipartite(bipartite));**

**}**

**}**

**Task 10:**

**WAP to display a graph edges in the below order no of edges 8 and no of vertex 5**

**Answer:**

**class Graph {**

**class Edge {**

**int src; // source/start vertex**

**int dest; // destination/end vertex**

**public Edge(int src, int dest) {**

**this.src = src;**

**this.dest = dest;**

**}**

**@Override**

**public String toString() {**

**return src + " - " + dest;**

**}**

**}**

**int vertices; // Number of vertices**

**int edges; // Number of edges**

**Edge[] edgeList; // Array to store edges**

**int edgeCount; // Current edge count**

**// Constructor**

**public Graph(int vertices, int edges) {**

**this.vertices = vertices;**

**this.edges = edges;**

**this.edgeList = new Edge[edges];**

**this.edgeCount = 0;**

**}**

**// Add edge to the graph**

**public void addEdge(int src, int dest) {**

**if (edgeCount < edges) {**

**edgeList[edgeCount] = new Edge(src, dest);**

**edgeCount++;**

**System.out.println("Added edge: " + src + " - " + dest);**

**} else {**

**System.out.println("Cannot add more edges. Maximum limit reached.");**

**}**

**}**

**// Display all edges**

**public void displayEdges() {**

**System.out.println("\n=== Graph Edge List ===");**

**System.out.println("Number of vertices: " + vertices);**

**System.out.println("Number of edges: " + edgeCount + "/" + edges);**

**System.out.println("\nEdges:");**

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

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

**}**

**}**

**// Display graph information**

**public void displayGraphInfo() {**

**System.out.println("\n=== Graph Information ===");**

**System.out.println("Vertices: " + vertices);**

**System.out.println("Edges: " + edgeCount);**

**System.out.println("Graph Type: " + (isComplete() ? "Complete" : "Incomplete"));**

**System.out.println("Maximum possible edges: " + getMaxPossibleEdges());**

**}**

**// Check if graph is complete**

**private boolean isComplete() {**

**int maxEdges = vertices \* (vertices - 1) / 2;**

**return edgeCount == maxEdges;**

**}**

**// Get maximum possible edges for undirected graph**

**private int getMaxPossibleEdges() {**

**return vertices \* (vertices - 1) / 2;**

**}**

**// Convert to adjacency list representation**

**public void displayAsAdjacencyList() {**

**System.out.println("\n=== Adjacency List Representation ===");**

**// Create adjacency list**

**java.util.Map<Integer, java.util.List<Integer>> adjList = new java.util.HashMap<>();**

**// Initialize adjacency list for all vertices**

**for (int i = 1; i <= vertices; i++) {**

**adjList.put(i, new java.util.ArrayList<>());**

**}**

**// Add edges to adjacency list (assuming undirected graph)**

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

**Edge edge = edgeList[i];**

**adjList.get(edge.src).add(edge.dest);**

**adjList.get(edge.dest).add(edge.src);**

**}**

**// Display adjacency list**

**for (int vertex = 1; vertex <= vertices; vertex++) {**

**System.out.println(vertex + " -> " + adjList.get(vertex));**

**}**

**}**

**// Convert to adjacency matrix representation**

**public void displayAsAdjacencyMatrix() {**

**System.out.println("\n=== Adjacency Matrix Representation ===");**

**// Create adjacency matrix**

**int[][] matrix = new int[vertices + 1][vertices + 1];**

**// Fill matrix with edges (assuming undirected graph)**

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

**Edge edge = edgeList[i];**

**matrix[edge.src][edge.dest] = 1;**

**matrix[edge.dest][edge.src] = 1; // For undirected graph**

**}**

**// Display matrix header**

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

**for (int i = 1; i <= vertices; i++) {**

**System.out.printf("%2d ", i);**

**}**

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

**// Display matrix**

**for (int i = 1; i <= vertices; i++) {**

**System.out.printf("%2d ", i);**

**for (int j = 1; j <= vertices; j++) {**

**System.out.printf("%2d ", matrix[i][j]);**

**}**

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

**}**

**}**

**// Find vertex degree**

**public void displayVertexDegrees() {**

**System.out.println("\n=== Vertex Degrees ===");**

**int[] degrees = new int[vertices + 1];**

**// Count degrees for each vertex**

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

**Edge edge = edgeList[i];**

**degrees[edge.src]++;**

**degrees[edge.dest]++;**

**}**

**// Display degrees**

**for (int i = 1; i <= vertices; i++) {**

**System.out.println("Vertex " + i + ": degree = " + degrees[i]);**

**}**

**}**

**}**

**// Main class to demonstrate the graph**

**public class GraphEdgeDisplay {**

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

**System.out.println("=== Graph Edge Display Demo ===\n");**

**// Create graph with 5 vertices and 8 edges**

**Graph graph = new Graph(5, 8);**

**System.out.println("Creating graph with 5 vertices and 8 edges...\n");**

**// Add the specified edges**

**System.out.println("Adding edges:");**

**graph.addEdge(1, 2);**

**graph.addEdge(1, 3);**

**graph.addEdge(1, 4);**

**graph.addEdge(2, 4);**

**graph.addEdge(2, 5);**

**graph.addEdge(3, 4);**

**graph.addEdge(3, 5);**

**graph.addEdge(4, 5);**

**// Display the graph in different representations**

**graph.displayEdges();**

**graph.displayGraphInfo();**

**graph.displayAsAdjacencyList();**

**graph.displayAsAdjacencyMatrix();**

**graph.displayVertexDegrees();**

**// Additional analysis**

**System.out.println("\n=== Additional Analysis ===");**

**analyzeGraph(graph);**

**}**

**// Additional graph analysis**

**public static void analyzeGraph(Graph graph) {**

**System.out.println("Graph Analysis:");**

**System.out.println("- This is an undirected graph");**

**System.out.println("- Each edge connects two vertices");**

**System.out.println("- The graph appears to be complete (all vertices connected to all others)");**

**System.out.println("- Total possible edges for 5 vertices: " + (5 \* 4 / 2) + " = 10");**

**System.out.println("- Current edges: 8");**

**System.out.println("- Missing edges: 2");**

**// Find missing edges**

**boolean[][] connected = new boolean[6][6];**

**// Mark existing edges**

**for (int i = 0; i < graph.edgeCount; i++) {**

**Graph.Edge edge = graph.edgeList[i];**

**connected[edge.src][edge.dest] = true;**

**connected[edge.dest][edge.src] = true;**

**}**

**System.out.println("\nMissing edges:");**

**for (int i = 1; i <= 5; i++) {**

**for (int j = i + 1; j <= 5; j++) {**

**if (!connected[i][j]) {**

**System.out.println(i + " - " + j);**

**}**

**}**

**}**

**}**

**}**

**/\* Output:**

**=== Graph Edge Display Demo ===**

**Creating graph with 5 vertices and 8 edges...**

**Adding edges:**

**Added edge: 1 - 2**

**Added edge: 1 - 3**

**Added edge: 1 - 4**

**Added edge: 2 - 4**

**Added edge: 2 - 5**

**Added edge: 3 - 4**

**Added edge: 3 - 5**

**Added edge: 4 - 5**

**=== Graph Edge List ===**

**Number of vertices: 5**

**Number of edges: 8/8**

**Edges:**

**1 - 2**

**1 - 3**

**1 - 4**

**2 - 4**

**2 - 5**

**3 - 4**

**3 - 5**

**4 - 5**

**=== Graph Information ===**

**Vertices: 5**

**Edges: 8**

**Graph Type: Incomplete**

**Maximum possible edges: 10**

**=== Adjacency List Representation ===**

**1 -> [2, 3, 4]**

**2 -> [1, 4, 5]**

**3 -> [1, 4, 5]**

**4 -> [1, 2, 3, 5]**

**5 -> [2, 3, 4]**

**=== Adjacency Matrix Representation ===**

**1 2 3 4 5**

**1 0 1 1 1 0**

**2 1 0 0 1 1**

**3 1 0 0 1 1**

**4 1 1 1 0 1**

**5 0 1 1 1 0**

**=== Vertex Degrees ===**

**Vertex 1: degree = 3**

**Vertex 2: degree = 3**

**Vertex 3: degree = 3**

**Vertex 4: degree = 4**

**Vertex 5: degree = 3**

**=== Additional Analysis ===**

**Graph Analysis:**

**- This is an undirected graph**

**- Each edge connects two vertices**

**- The graph appears to be complete (all vertices connected to all others)**

**- Total possible edges for 5 vertices: 10**

**- Current edges: 8**

**- Missing edges: 2**

**Missing edges:**

**1 - 5**

**2 - 3**

**\*/**