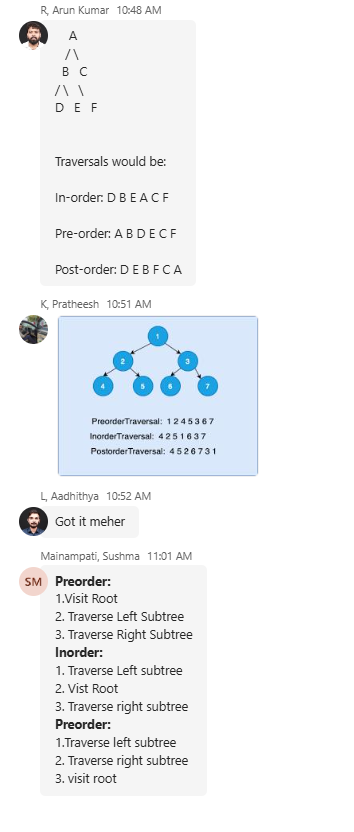
Day 15 - 5th july 2025



Preorder - ROOT  Left  Right  
Inorder - Left  ROOT  Right  
Postorder - Left  Right  ROOT

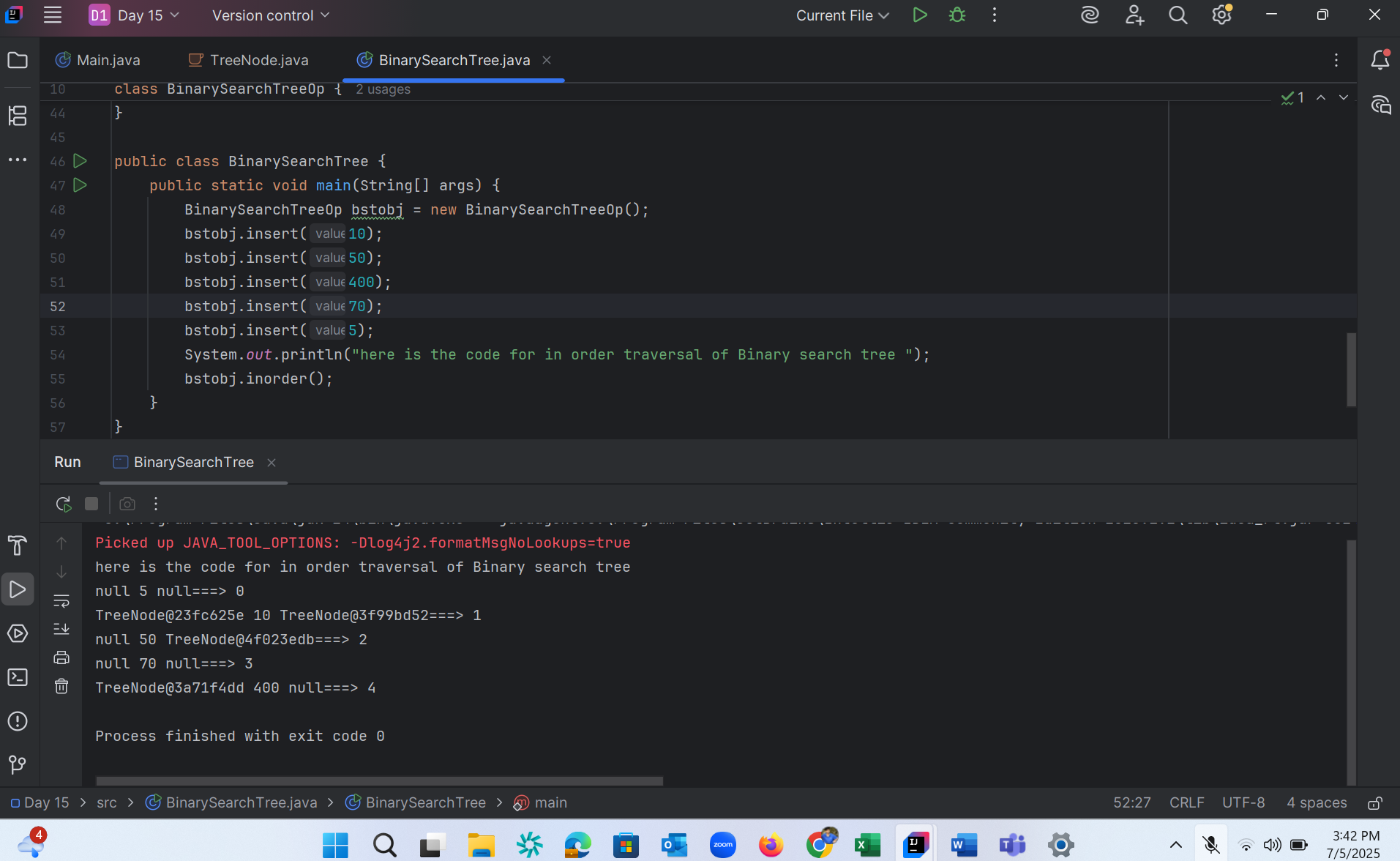
Code :

Task 1-4

public class BinarySearchTree {  
  
 // Node class  
 private static class Node {  
 int value;  
 Node left, right;  
  
 Node(int v) {  
 value = v;  
 left = right = null;  
 }  
 }  
  
 private Node root;  
  
 public BinarySearchTree() {  
 root = null;  
 }  
  
 // Insert into an empty tree  
 public void insertFirst(int value) {  
 if (root != null) {  
 System.*out*.println("Tree is not empty — use insert(value) instead.");  
 return;  
 }  
 root = new Node(value);  
 }  
  
 // Insert into a non-empty tree  
 public void insert(int value) {  
 if (root == null) {  
 System.*out*.println("Tree is empty — using insertFirst instead.");  
 insertFirst(value);  
 } else {  
 root = insertRec(root, value);  
 }  
 }  
  
 // Recursive insert helper  
 private Node insertRec(Node node, int value) {  
 if (node == null) return new Node(value);  
  
 if (value < node.value) {  
 node.left = insertRec(node.left, value);  
 } else if (value > node.value) {  
 node.right = insertRec(node.right, value);  
 }  
 // Skip duplicate  
 return node;  
 }  
  
 // Inorder Traversal (Left → Root → Right)  
 public void inorderTraversal() {  
 System.*out*.print("Inorder traversal: ");  
 inorderRec(root);  
 System.*out*.println();  
 }  
  
 private void inorderRec(Node node) {  
 if (node != null) {  
 inorderRec(node.left);  
 System.*out*.print(node.value + " ");  
 inorderRec(node.right);  
 }  
 }  
  
 // Preorder Traversal (Root → Left → Right)  
 public void preorderTraversal() {  
 System.*out*.print("Preorder traversal: ");  
 preorderRec(root);  
 System.*out*.println();  
 }  
  
 private void preorderRec(Node node) {  
 if (node != null) {  
 System.*out*.print(node.value + " ");  
 preorderRec(node.left);  
 preorderRec(node.right);  
 }  
 }  
  
 // Postorder Traversal (Left → Right → Root)  
 public void postorderTraversal() {  
 System.*out*.print("Postorder traversal: ");  
 postorderRec(root);  
 System.*out*.println();  
 }  
  
 private void postorderRec(Node node) {  
 if (node != null) {  
 postorderRec(node.left);  
 postorderRec(node.right);  
 System.*out*.print(node.value + " ");  
 }  
 }  
  
  
 public static void main(String[] args) {  
 BinarySearchTree bst = new BinarySearchTree();  
  
  
 bst.insertFirst(50);  
 bst.insert(30);  
 bst.insert(70);  
 bst.insert(20);  
 bst.insert(40);  
 bst.insert(60);  
 bst.insert(80);  
  
  
 bst.inorderTraversal();  
 bst.preorderTraversal();  
 bst.postorderTraversal();  
 }  
}

Task 1-4

class TreeNode {  
 int value;  
 TreeNode left, right;  
  
 TreeNode(int item) {  
 value = item;  
 left = right = null;  
 }  
}  
class BinarySearchTreeOp {  
 TreeNode root;  
 int i=0;  
  
 void insert(int value) { // 10  
 root = insertVal(root, value); //root = null  
 }  
 TreeNode insertVal(TreeNode node, int value) { // null, 10 //  
 if (node == null) {  
 node = new TreeNode(value);  
 return node;  
 }  
 if (value < node.value) {  
 node.left = insertVal(node.left, value);  
 } else if (value > node.value) {  
 node.right = insertVal(node.right, value);  
 }  
 return node;  
 }  
  
 void inorder() {  
 inorderVal(root);  
 }  
  
 void inorderVal(TreeNode node) {  
 if (node != null) {  
 inorderVal(node.left);  
// System.out.print(node.value + " node.left ");  
 System.*out*.println(node.left + " "+ node.value + " " + node.right + "===> "+ i++);  
  
 inorderVal(node.right);  
// System.out.print(node.value + " node.left ");  
 }  
 }  
}  
  
public class BinarySearchTree {  
 public static void main(String[] args) {  
 BinarySearchTreeOp bstobj = new BinarySearchTreeOp();  
 bstobj.insert(10);  
 bstobj.insert(50);  
 bstobj.insert(400);  
 bstobj.insert(70);  
 bstobj.insert(5);  
 System.*out*.println("here is the code for in order traversal of Binary search tree ");  
 bstobj.inorder();  
 }  
}



Task 5 :

Applications of Trees ?

Here are the key applications of Trees in computer science and real-world scenarios: 1. File System Organization - Directory structures in operating systems - Folder hierarchies on computers - File explorers in operating systems

2. Database Systems - B-Trees and B+ Trees for indexing - Binary Search Trees for efficient data retrieval - Decision trees for data classification

3. HTML DOM (Document Object Model) - Web browsers use tree structures to represent HTML pages - CSS selectors navigate this tree structure - XML parsing

4. Artificial Intelligence - Decision Trees for prediction models - Game Trees for strategic planning (like chess) - Mini-max trees in game theory

5. Network Routing - Network broadcast algorithms - Spanning trees in network topology - Routing tables in networking

6. Compiler Design - Expression trees - Abstract Syntax Trees (AST) - Parse trees for programming languages

7. Organization Charts - Company hierarchies - Family trees - Organizational structures

8. Computer Graphics - Scene graphs - Spatial partitioning (BSP trees, Quad trees) - Animation hierarchies

9. Search Algorithms - Binary Search Trees for efficient searching - Trie structures for string searching - Auto-complete features

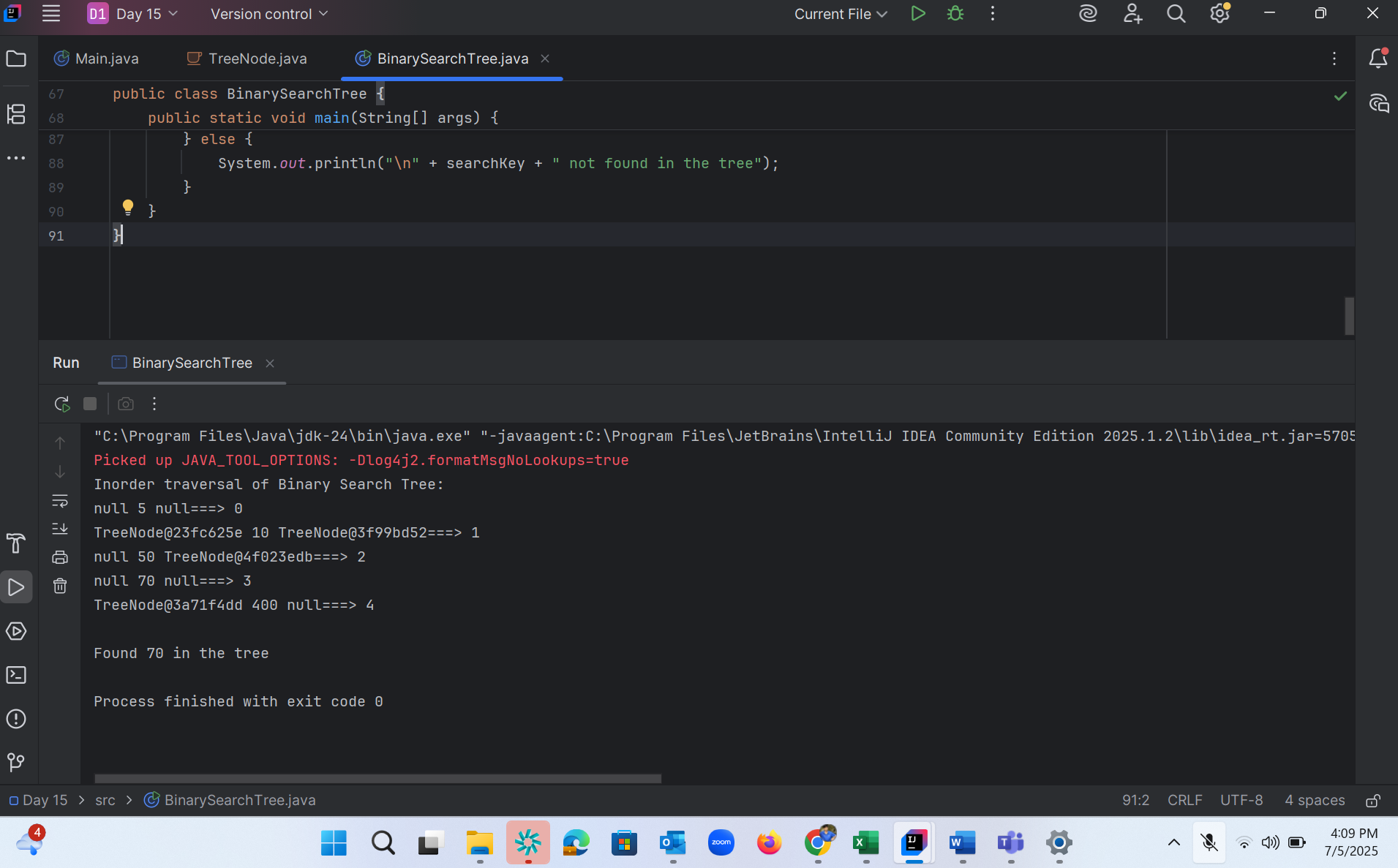
10. Mathematics - Expression evaluation - Mathematical formula representation - Probability trees

11. Machine Learning - Random Forest algorithms - Decision tree learning - Hierarchical clustering

12. Operating Systems - Process trees (parent-child relationships) - Memory allocation (buddy system) - File system implementation These applications demonstrate why trees are one of the most important and widely used data structures in computer science and software development.

Task 6 :

class TreeNode {  
 int value;  
 TreeNode left, right;  
  
 TreeNode(int value) {  
 this.value = value;  
 left = right = null;  
 }  
}  
  
class BinarySearchTreeOperations {  
 TreeNode root;  
 int i = 0;  
  
 // Constructor  
 public BinarySearchTreeOperations() {  
 this.root = null;  
 }  
  
 // Insert operations  
 void insert(int value) {  
 root = insertVal(root, value);  
 }  
  
 TreeNode insertVal(TreeNode node, int value) {  
 if (node == null) {  
 node = new TreeNode(value);  
 return node;  
 }  
 if (value < node.value) {  
 node.left = insertVal(node.left, value);  
 } else if (value > node.value) {  
 node.right = insertVal(node.right, value);  
 }  
 return node;  
 }  
  
 // Search operation  
 public TreeNode search(int key) {  
 TreeNode current = root;  
 while (current != null) {  
 if (key == current.value) {  
 return current;  
 } else if (key < current.value) {  
 current = current.left;  
 } else {  
 current = current.right;  
 }  
 }  
 return null;  
 }  
  
 // Traversal operations  
 void inorder() {  
 inorderVal(root);  
 }  
  
 void inorderVal(TreeNode node) {  
 if (node != null) {  
 inorderVal(node.left);  
 System.*out*.println(node.left + " " + node.value + " " + node.right + "===> " + i++);  
 inorderVal(node.right);  
 }  
 }  
}  
  
public class BinarySearchTree {  
 public static void main(String[] args) {  
 BinarySearchTreeOperations bst = new BinarySearchTreeOperations();  
  
 // Insert operations  
 bst.insert(10);  
 bst.insert(50);  
 bst.insert(400);  
 bst.insert(70);  
 bst.insert(5);  
  
 // Inorder traversal  
 System.*out*.println("Inorder traversal of Binary Search Tree:");  
 bst.inorder();  
  
 // Search operations  
 int searchKey = 70;  
 TreeNode result = bst.search(searchKey);  
 if (result != null) {  
 System.*out*.println("\nFound " + searchKey + " in the tree");  
 } else {  
 System.*out*.println("\n" + searchKey + " not found in the tree");  
 }  
 }  
}



Task 7 :

Types of binary trees?

Full Binary Tree (or Strict/Proper Binary Tree):

In a full binary tree, every node has either zero or two children. No node has only one child.

Complete Binary Tree:

A complete binary tree is a binary tree where all levels are completely filled except possibly the last level, and in the last level, all nodes are as far left as possible.

Perfect Binary Tree:

A perfect binary tree is a special case that is both a full and a complete binary tree. All internal nodes have two children, and all leaf nodes are at the same level (depth).

Degenerate (or Pathological) Binary Tree:

In a degenerate binary tree, each parent node has only one child node. This results in a tree that resembles a linked list, either left-skewed (all nodes lean left) or right-skewed (all nodes lean right).

Balanced Binary Tree:

A balanced binary tree is one where the heights of the left and right subtrees of any node differ by at most one. This property helps maintain efficient search, insertion, and deletion operations. Examples include AVL trees and Red-Black trees.

Skewed Binary Tree:

This is a specific type of degenerate tree where all nodes are either on the left side (left-skewed) or the right side (right-skewed).

Binary Search Tree (BST):

While not a structural type in the same sense as the above, a BST is a widely used type of binary tree where nodes are ordered. For every node, all values in its left subtree are less than the node's value, and all values in its right subtree are greater than the node's value.

Heap (Min-Heap/Max-Heap):

Heaps are tree-based data structures that satisfy the heap property, which dictates a specific ordering between parent and child nodes. In a max-heap, the parent node's value is greater than or equal to its children's, while in a min-heap, it's less than or equal to its children's.

Task 8 :

Here are the key applications of Graphs in various fields:

Social Networks

Representing friendships and connections

Analyzing social relationships

Influence mapping

Community detection

Transportation Systems

Road networks and navigation

Flight routes optimization

Public transit planning

Traffic flow analysis

Computer Networks

Internet topology

Network routing protocols

Data center architecture

Network security analysis

Biology and Chemistry

Molecular structures

Protein interaction networks

Gene regulatory networks

Ecosystem food webs

Task 9 :

Types of Graphs

1.Undirected Graph

A graph with only undirected edges is said to be undirected graph.

2.Directed Graph

A graph with only directed edges is said to be directed graph.

3.Complete Graph

A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An

undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in

the graph. The following figure shows a complete graph.

4.Regular Graph

Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other

node.

5.Cycle Graph

A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path

is a cycle.

3

6.Acyclic Graph

A graph without cycle is called acyclic graphs.

7. Weighted Graph

A graph is said to be weighted if there are some non negative value assigned to each edges of the graph. The

value is equal to the length between two vertices. Weighted graph is also called a network.

Task 10 :

class Graph {  
  
 class Edge {  
 int src;  
 int dest;  
  
  
 @Override  
 public String toString() {  
 return src + " - " + dest;  
 }  
 }  
  
 int vertices;  
 int edges; // Total number of edges in the graph (fixed size for the array)  
 Edge[] edge; // Array to store all the Edge objects  
  
  
 public Graph(int vertices, int edges) {  
 this.vertices = vertices;  
 this.edges = edges;  
  
 // Initialize the array to hold 'edges' number of Edge objects.  
 // Each element in the array must also be initialized with a new Edge instance.  
 edge = new Edge[edges];  
 for(int i = 0; i < edges; i++) {  
 edge[i] = new Edge(); // Create a new Edge object for each slot in the array  
 }  
 }  
  
  
 public static void main(String[] args) {  
 // Define the number of vertices and edges as per the requirement  
 int noVertices = 5;  
 int noEdges = 8;  
  
 // Create a new Graph object with 5 vertices and 8 edges  
 Graph gObj = new Graph(noVertices, noEdges);  
  
 // Assign the source and destination for each edge directly into the array  
 System.*out*.println("Assigning edges to the graph...");  
 gObj.edge[0].src = 1;  
 gObj.edge[0].dest = 2;  
 gObj.edge[1].src = 1;  
 gObj.edge[1].dest = 3;  
 gObj.edge[2].src = 1;  
 gObj.edge[2].dest = 4;  
 gObj.edge[3].src = 2;  
 gObj.edge[3].dest = 4;  
 gObj.edge[4].src = 2;  
 gObj.edge[4].dest = 5;  
 gObj.edge[5].src = 3;  
 gObj.edge[5].dest = 4;  
 gObj.edge[6].src = 3;  
 gObj.edge[6].dest = 5;  
 gObj.edge[7].src = 4;  
 gObj.edge[7].dest = 5;  
 System.*out*.println("Edges assigned successfully.");  
  
 // Display all the edges in the graph  
 System.*out*.println("\n--- Displaying Graph Edges ---");  
 System.*out*.println("Graph Edges (Total: " + noEdges + "):");  
 for(int i = 0; i < noEdges; i++) {  
 System.*out*.println(gObj.edge[i].src + " - " + gObj.edge[i].dest);  
 }  
 }  
}

