**DAY15 – July 5th**

**TASK1 - Create a node for a tree and include a constructor.**

class TreeNode {  
 int data;  
 TreeNode left, right;  
  
 // Constructor  
 TreeNode(int value) {  
 data = value;  
 left = null;  
 right = null;  
 }  
}

**TASK2 – Binary Search Tree with Insertion**

import java.util.\*;  
class TreeNode {  
 int data;  
 TreeNode left, right;  
  
 // Constructor  
 TreeNode(int value) {  
 data = value;  
 left = null;  
 right = null;  
 }  
}  
  
// BinarySearchTree class  
public class Day15\_BinaryTree {  
 TreeNode root;  
  
 // Constructor  
 public Day15\_BinaryTree() {  
 root = null;  
 }  
  
 public void insert(int value) {  
 if (root == null) {  
 root = new TreeNode(value);// if root is empty it will create a new node and assigns it to root  
 System.*out*.println("Inserted " + value + " as root node.");  
 } else {  
 insert(root, value);  
 }  
 }  
  
 // 2️⃣ Insert when tree is not empty  
 private void insert(TreeNode node, int value) {  
 if (value < node.data) {  
 if (node.left == null) {  
 node.left = new TreeNode(value);  
 System.*out*.println("Inserted " + value + " to the left of " + node.data);  
 } else {  
 insert(node.left, value);  
 }  
 } else if (value > node.data) {  
 if (node.right == null) {  
 node.right = new TreeNode(value);  
 System.*out*.println("Inserted " + value + " to the right of " + node.data);  
 } else {  
 insert(node.right, value);  
 }  
 } else {  
 // Duplicate value  
 System.*out*.println("Value " + value + " already exists in the tree.");  
 }  
 }  
  
 // Main method to test insertion  
 public static void main(String[] args) {  
 Day15\_BinaryTree bst = new Day15\_BinaryTree();  
  
 // Insert nodes  
 bst.insert(50); // root  
 bst.insert(30);  
 bst.insert(70);  
 bst.insert(20);  
 bst.insert(40);  
 bst.insert(60);  
 bst.insert(80);  
 bst.insert(70); // Duplicate  
 }  
}

**Output**

Inserted 50 as root node.

Inserted 30 to the left of 50

Inserted 70 to the right of 50

Inserted 20 to the left of 30

Inserted 40 to the right of 30

Inserted 60 to the left of 70

Inserted 80 to the right of 70

Value 70 already exists in the tree.

**TASK3 – InOrder Traversal**

class Node12  
{  
 int data;  
 Node12 left;  
 Node12 right;  
  
 Node12(int val)  
 {  
 data=val;  
 left=right=null;  
 }  
}  
public class Day15\_InOrderTraversal  
{  
 public static void InOrder(Node12 root)  
 {  
 if(root!=null)  
 {  
 *InOrder*(root.left);  
 System.*out*.println(root.data+" ");  
 *InOrder*(root.right);  
 }  
 }  
  
 public static void main(String[] args)  
 {  
 Node12 root = new Node12(10);  
 root.left = new Node12(20);  
 root.right= new Node12(30);  
 root.right.left=new Node12(40);  
 root.right.right=new Node12(50);  
 *InOrder*(root);  
  
 }  
}

**Output**

20

10

40

30

50

**TASK4**

class TreeNode1 {  
 int data;  
 TreeNode1 left, right;  
  
 // Constructor  
 TreeNode1(int value) {  
 data = value;  
 left = null;  
 right = null;  
 }  
}  
public class Day15\_BST {  
 TreeNode root; // root of the tree  
  
 // Constructor: creates an empty tree  
 public Day15\_BST() {  
 root = null;  
 }  
  
 // Insert method for empty tree or first insert  
 public void insert(int value) {  
 if (root == null) {  
 root = new TreeNode(value);  
 System.*out*.println("Inserted " + value + " as root node.");  
 } else {  
 insert(root, value); // delegate to recursive method  
 }  
 }  
  
 // Insert method when tree is not empty (recursive)  
 private void insert(TreeNode current, int value) {  
 if (value < current.data) {  
 if (current.left == null) {  
 current.left = new TreeNode(value);  
 System.*out*.println("Inserted " + value + " to the left of " + current.data);  
 } else {  
 insert(current.left, value);  
 }  
 } else if (value > current.data) {  
 if (current.right == null) {  
 current.right = new TreeNode(value);  
 System.*out*.println("Inserted " + value + " to the right of " + current.data);  
 } else {  
 insert(current.right, value);  
 }  
 } else {  
 System.*out*.println("Value " + value + " already exists. Not inserted.");  
 }  
 }  
  
 // ✅ Inorder Traversal Method (recursive)  
 public void inorderTraversal() {  
 System.*out*.print("Inorder Traversal: ");  
 inorderTraversal(root);  
 System.*out*.println(); // for newline after traversal  
 }  
  
 private void inorderTraversal(TreeNode node) {  
 if (node != null) {  
 inorderTraversal(node.left); // Visit left  
 System.*out*.print(node.data + " "); // Visit root  
 inorderTraversal(node.right); // Visit right  
 }  
 }  
  
 // Main method to test the insertions and traversal  
 public static void main(String[] args) {  
 Day15\_BST bst = new Day15\_BST();  
  
 // Insert some values into the BST  
 bst.insert(50);  
 bst.insert(30);  
 bst.insert(70);  
 bst.insert(20);  
 bst.insert(40);  
 bst.insert(60);  
 bst.insert(80);  
 bst.insert(70); // Duplicate test  
  
 // 🔍 Perform inorder traversal  
 bst.inorderTraversal();  
 }  
}

**Output**

Inserted 50 as root node.

Inserted 30 to the left of 50

Inserted 70 to the right of 50

Inserted 20 to the left of 30

Inserted 40 to the right of 30

Inserted 60 to the left of 70

Inserted 80 to the right of 70

Value 70 already exists. Not inserted.

Inorder Traversal: 20 30 40 50 60 70 80

**TASK5 – Application of Trees**

**TASK6 – Searching BST**

class TreeNode3 {  
 int data;  
 TreeNode3 left, right;  
  
 // Constructor  
 TreeNode3(int value) {  
 data = value;  
 left = null;  
 right = null;  
 }  
}  
  
// BinarySearchTree class: handles insert, inorder, and search  
class Day15\_BSTSearch {  
 TreeNode root;  
  
 // Constructor  
 public Day15\_BSTSearch() {  
 root = null;  
 }  
  
 // Insert method (handles root and delegates)  
 public void insert(int value) {  
 if (root == null) {  
 root = new TreeNode(value);  
 System.*out*.println("Inserted " + value + " as root node.");  
 } else {  
 insert(root, value);  
 }  
 }  
  
 // Recursive insert  
 private void insert(TreeNode current, int value) {  
 if (value < current.data) {  
 if (current.left == null) {  
 current.left = new TreeNode(value);  
 System.*out*.println("Inserted " + value + " to the left of " + current.data);  
 } else {  
 insert(current.left, value);  
 }  
 } else if (value > current.data) {  
 if (current.right == null) {  
 current.right = new TreeNode(value);  
 System.*out*.println("Inserted " + value + " to the right of " + current.data);  
 } else {  
 insert(current.right, value);  
 }  
 } else {  
 System.*out*.println("Value " + value + " already exists. Not inserted.");  
 }  
 }  
  
 // Inorder traversal (left, root, right)  
 public void inorderTraversal() {  
 System.*out*.print("Inorder Traversal: ");  
 inorderTraversal(root);  
 System.*out*.println();  
 }  
  
 private void inorderTraversal(TreeNode node) {  
 if (node != null) {  
 inorderTraversal(node.left);  
 System.*out*.print(node.data + " ");  
 inorderTraversal(node.right);  
 }  
 }  
  
 // Search method (returns true/false)  
 public boolean search(int key) {  
 return search(root, key);  
 }  
  
 private boolean search(TreeNode node, int key) {  
 if (node == null) {  
 return false;  
 }  
  
 if (key == node.data) {  
 return true;  
 } else if (key < node.data) {  
 return search(node.left, key);  
 } else {  
 return search(node.right, key);  
 }  
 }  
  
 // Main method to test the BST  
 public static void main(String[] args) {  
 Day15\_BSTSearch bst = new Day15\_BSTSearch();  
  
 // Insert values  
 bst.insert(50);  
 bst.insert(30);  
 bst.insert(70);  
 bst.insert(20);  
 bst.insert(40);  
 bst.insert(60);  
 bst.insert(80);  
 bst.insert(70); // Duplicate test  
  
 // Traversal  
 bst.inorderTraversal();  
  
 // Search  
 System.*out*.println("Search for 30: " + bst.search(30)); // true  
 System.*out*.println("Search for 25: " + bst.search(25)); // false  
 }  
}

**Output**

Inserted 50 as root node.

Inserted 30 to the left of 50

Inserted 70 to the right of 50

Inserted 20 to the left of 30

Inserted 40 to the right of 30

Inserted 60 to the left of 70

Inserted 80 to the right of 70

Value 70 already exists. Not inserted.

Inorder Traversal: 20 30 40 50 60 70 80

Search for 30: true

Search for 25: false

**TASK7 – Types of Binary Tree**

1. Full Binary Tree
2. Perfect Binary tree
3. Complete Binary Tree
4. Balanced Binary Tree
5. Skewed Binary Tree
6. AVL Binary Tree
7. Binary search Tree
8. Red-Black Binary tree

**TASK8 – Applications of Graphs**

**1. Computer Networks**

* Nodes = computers/routers, Edges = connections
* Used in designing and analyzing computer and mobile networks
* Routing algorithms like Dijkstra help find shortest paths

**2. Social Networks**

* Nodes = people, Edges = friendships/follows
* Helps in suggesting friends, finding influencers, and clustering users

**3. Web Page Linking**

* Web pages = nodes, hyperlinks = edges
* Search engines like Google use graph-based PageRank algorithm

**4. GPS & Navigation Systems**

* Locations = nodes, Roads = edges
* Used in route planning (Google Maps, Ola, Uber)
* Algorithms: A\*, Dijkstra, Floyd-Warshall

**5. Project Management (PERT/CPM)**

* Tasks = nodes, dependencies = edges
* Helps in scheduling, critical path detection, and avoiding delays

**6. Game Development**

* Game maps, level design use graphs
* Pathfinding for AI (e.g., enemy movements) uses A\* or BFS

**7. Circuit Design**

* Electrical circuits modeled using graphs
* Components = nodes, connections = edges
* Analyzed using DFS/BFS

**8. Transportation Systems**

* Airports, bus stops = nodes; routes = edges
* Used in public transport planning and logistics.

**TASK9 – Types of Graphs**

**1. Undirected Graph**

* Edges have **no direction**.
* Edge (u, v) means you can go both from u → v and v → u.

**Example**: Social networks, road maps.

**2. Directed Graph (Digraph)**

* Edges have a **direction**.
* Edge (u → v) means you can go **only from u to v**.

**Example**: Web page links, task scheduling.

**3. Weighted Graph**

* Each edge has a **weight** or **cost**.
* Used to represent distances, costs, time, etc.

**Example**: Google Maps, flight routes.

**4. Unweighted Graph**

* Edges do **not** have weights.
* Only show connectivity, not cost.

**5. Cyclic Graph**

* Contains at least one **cycle** (a path that starts and ends at the same node).

**6. Acyclic Graph**

* Contains **no cycles**.
* Common in task scheduling and dependency resolution.

**7. Connected Graph (Undirected)**

* There is **a path between every pair of nodes**.

**8. Disconnected Graph**

* Some nodes **cannot be reached** from others.

**9. Complete Graph**

* Every pair of distinct nodes is connected by a unique edge.
* For n vertices, there are n(n−1)/2 edges (undirected).

**10. Sparse Graph**

* Contains **very few edges** compared to the number of nodes.

**11. Dense Graph**

* Contains **many edges**, close to the maximum number of edges.

**12. Bipartite Graph**

* Vertices can be divided into **two sets** such that:
  + No two nodes within the same set are connected.

**Example**: Job assignment problems, matching.

**13. Directed Acyclic Graph (DAG)**

* A **directed** graph with **no cycles**.
* Used in:
  + Task scheduling
  + Course prerequisites
  + Git commit history

**14. Tree (Special Type of Graph)**

* A **connected**, **acyclic**, and **undirected** graph with n-1 edges and n vertices.

**15. Subgraph**

* A portion of a graph containing a subset of its vertices and edges.

**TASK10 – Write a program to display a graph redges in the below order no od edges 8 and no of vertex 5**

1 - 2

1 - 3

1 - 4

2 - 4

2 - 5

3 - 4

3 - 5

4 - 5

public class Day15\_EdgesOfGraph {  
  
 // Inner class to represent an edge  
 static class Edge {  
 int src;  
 int dest;  
  
 // Constructor  
 Edge(int src, int dest) {  
 this.src = src;  
 this.dest = dest;  
 }  
 }  
  
 public static void main(String[] args) {  
 // Number of vertices  
 int vertices = 5;  
  
 // Create an array of edges directly (8 edges)  
 Edge[] edges = {  
 new Edge(1, 2),  
 new Edge(1, 3),  
 new Edge(1, 4),  
 new Edge(2, 4),  
 new Edge(2, 5),  
 new Edge(3, 4),  
 new Edge(3, 5),  
 new Edge(4, 5)  
 };  
  
 // Display all edges  
 System.*out*.println("Graph edges:");  
 for (Edge edge : edges) {  
 System.*out*.println(edge.src + " - " + edge.dest);  
 }  
 }  
}

**Output**

Graph edges:

1 - 2

1 - 3

1 - 4

2 - 4

2 - 5

3 - 4

3 - 5

4 – 5