[W4 - Data Structure & Algorithms]

Problem 1: Compute the height of a binary tree.

Overview: Detect the height of a binary tree.

Definition of important concepts

1. Binary tree: is defined as a tree data structure where each node has at most 2 children.
2. The height of a node in a binary tree is the largest number of edges in a path from a leaf node to a target node.
3. The height of a binary tree: is equal to the largest number of edges from the root to the most distant leaf node, or the height of the most distant node.

Algorithm Steps:

1. **Using recursion**
    - Since the tree is the binary tree, using recursion, we just check the height of the left subtree and the right subtree.
    - Take the maximum of the two numbers, and that is the height of the tree.
2. **Code**

```
class TreeNode {
   int value;
   TreeNode left, right;

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

public class BinaryTree {
   TreeNode root;

   // Method to calculate the height of a binary tree
   int height(TreeNode node) {
       if (node == null)
           return 0;
       else {
           // Compute the height of each subtree
           int leftHeight = height(node.left);
           int rightHeight = height(node.right);

           // Use the larger one
           return Math.max(leftHeight, rightHeight) + 1;
       }
   }

}
```

Problem 2: Implement Binary Search Tree (BST)

Overview: Implement a Binary Search Tree (BST) for integers. Provide operations for insertion, in-order traversal, and searching on the tree.

Algorithm Steps:

1. **BST Insertion**:
    - Insert nodes into the BST while maintaining its properties using recursion.
    - Each new node is inserted as a leaf.
2. **In-Order Traversal**:

    - Traverse the BST in in-order fashion (left, root, right) to confirm the ascending order of values.
3. **BST Search**:
    - Search for a value in the BST.
    - Keep track of the number of comparisons made during the search.
4. **Code**:

```
class Node {
   int data;
   Node left, right;

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

class BinarySearchTree {
   Node root;

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

   // Insert a new node
   void insert(int data) {
       root = insertRec(root, data);
   }

   // Recursive function to insert a new node
   Node insertRec(Node root, int data) {
       if (root == null) {
           root = new Node(data);
           return root;
       }
       if (data < root.data)
           root.left = insertRec(root.left, data);
       else if (data > root.data)
           root.right = insertRec(root.right, data);

       return root;
   }

   // In-order traversal
   void inOrder() {
       inOrderRec(root);
   }

   void inOrderRec(Node root) {
       if (root != null) {
           inOrderRec(root.left);
           System.out.print(root.data + " ");
           inOrderRec(root.right);
       }
   }

   // Search for a node
   boolean search(int data) {
```

```
        return searchRec(root, data, new int[1]);
    }

    boolean searchRec(Node root, int data, int[] comparisons) {
        if (root == null)
            return false;

        comparisons[0]++;
        if (root.data == data)
            return true;

        if (data < root.data)
            return searchRec(root.left, data, comparisons);
        else
            return searchRec(root.right, data, comparisons);
    }
}

// Test the BST
public class Main {
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();
        int[] values = {4, 2, 8, 3, 1, 7, 9, 6, 5};

        for (int value : values) {
            bst.insert(value);
        }

        System.out.println("In-order Traversal:");
        bst.inOrder();

        int searchValue = 5;
        int[] comparisons = new int[1];
        boolean found = bst.search(searchValue, comparisons);
        System.out.println("\nSearching for " + searchValue + ": " + found + ", Comparisons: " + comparisons[0]);
    }
}
```

Problem 3: Graph Representation and Traversal

Overview: Represent an undirected graph using an adjacency matrix. Perform and display the results of Depth-First Search (DFS) and Breadth-First Search (BFS) to traverse the graph and list the vertex labels.

Algorithm Steps:

1. **Graph Representation**:
    - Represent the graph using an adjacency matrix, where each cell (i, j) is 1 if there is an edge between vertices i and j, and 0 otherwise.
2. **Depth-First Search (DFS)**:
    - Traverse the graph deeply, exploring as far as possible along each branch before backtracking.
3. **Breadth-First Search (BFS)**:
    - Traverse the graph broadly, exploring all neighbor vertices at the present depth level before moving on to the vertices at the next depth level.
4. **Code**:

```
import java.util.*;
```

```
class Graph {
   private int V;   // Number of vertices
   private int[][] adjMatrix; // Adjacency Matrix

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

   // Function to add an edge into the graph
   void addEdge(int v, int w) {
       adjMatrix[v][w] = 1;
       adjMatrix[w][v] = 1; // For undirected graph
   }

   // DFS traversal of the vertices
   void DFS(int v, boolean visited[]) {
       visited[v] = true;
       System.out.print(v + " ");

       for (int i = 0; i < V; i++) {
           if (adjMatrix[v][i] == 1 && !visited[i]) {
               DFS(i, visited);
           }
       }
   }

   // Function to do DFS traversal
   void DFS(int v) {
       boolean visited[] = new boolean[V];
       DFS(v, visited);
   }

   // BFS traversal of the vertices
   void BFS(int s) {
       boolean visited[] = new boolean[V];
       LinkedList<Integer> queue = new LinkedList<Integer>();

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

       while (queue.size() != 0) {
           s = queue.poll();
           System.out.print(s + " ");

           for (int i = 0; i < V; i++) {
               if (adjMatrix[s][i] == 1 && !visited[i]) {
                   visited[i] = true;
                   queue.add(i);
               }
           }
       }
   }
}

// Test the Graph Traversal
public class Main {
   public static void main(String args[]) {
```

```
      // Test with the provided graph
        Graph g = new Graph(6);

        g.addEdge(1, 2);
        g.addEdge(2, 3);
        g.addEdge(3, 4);
        g.addEdge(4, 5);
        g.addEdge(5, 6);
        g.addEdge(6, 1);
        g.addEdge(2, 5);

        System.out.println("Depth First Traversal (starting from vertex 2):");
        g.DFS(1);

        System.out.println("\nBreadth First Traversal (starting from vertex 2):");
        g.BFS(1);
    }
}
```