WWC:-4

1.Check if a Binary Tree is Balanced.

Code:-

```
// Definition for a binary tree node
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int val) {
        this.val = val;
    }
}

public class BalancedBinaryTree {

    // Method to check if the tree is balanced
    public boolean isBalanced(TreeNode root) {
        return checkHeight(root) != -1;
    }

    // Helper method to check height and balance
    private int checkHeight(TreeNode node) {
        if (node == null) {
            return 0; // Base case: height of null node is 0
        }

        // Recursively calculate the height of left and right subtrees
        int leftHeight = checkHeight(node.left);
        if (leftHeight == -1) return -1; // Left subtree is unbalanced

        int rightHeight = checkHeight(node.right);
        if (rightHeight == -1) return -1; // Right subtree is unbalanced

        // Check the difference in height of subtrees
        if (Math.abs(leftHeight - rightHeight) > 1) {
            return -1; // Current node is unbalanced
        }

        // Return the height of the current node
        return Math.max(leftHeight, rightHeight) + 1;
    }
```

```
    // Main method for testing
    public static void main(String[] args) {
        BalancedBinaryTree treeChecker = new BalancedBinaryTree();

        // Example Tree: Balanced
        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.right = new TreeNode(6);

        System.out.println("Is the tree balanced? " + treeChecker.isBalanced(root)); // Output:
true

        // Example Tree: Unbalanced
        TreeNode unbalancedRoot = new TreeNode(1);
        unbalancedRoot.left = new TreeNode(2);
        unbalancedRoot.left.left = new TreeNode(3);

        System.out.println("Is the tree balanced? " + treeChecker.isBalanced(unbalancedRoot));
// Output: false
    }
}
```

1. Lowest Common Ancestor (LCA) of Two Nodes.

Code:-

```
// Definition for a binary tree node
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class LowestCommonAncestor {

    // Method to find the Lowest Common Ancestor
    public TreeNode findLCA(TreeNode root, TreeNode p, TreeNode q) {
        // Base case: if the root is null, or we find either p or q, return root
        if (root == null || root == p || root == q) {
```

```
            return root;
        }

        // Recur for left and right subtrees
        TreeNode leftLCA = findLCA(root.left, p, q);
        TreeNode rightLCA = findLCA(root.right, p, q);

        // If both leftLCA and rightLCA are non-null, current root is the LCA
        if (leftLCA != null && rightLCA != null) {
            return root;
        }

        // Otherwise, return the non-null child (either leftLCA or rightLCA)
        return (leftLCA != null) ? leftLCA : rightLCA;
    }

    // Helper method to create a sample binary tree for testing
    public static TreeNode createSampleTree() {
        TreeNode root = new TreeNode(3);
        root.left = new TreeNode(5);
        root.right = new TreeNode(1);
        root.left.left = new TreeNode(6);
        root.left.right = new TreeNode(2);
        root.right.left = new TreeNode(0);
        root.right.right = new TreeNode(8);
        root.left.right.left = new TreeNode(7);
        root.left.right.right = new TreeNode(4);
        return root;
    }

    // Main method to test the solution
    public static void main(String[] args) {
        LowestCommonAncestor lcaFinder = new LowestCommonAncestor();

        TreeNode root = createSampleTree();
        TreeNode p = root.left;       // Node 5
        TreeNode q = root.left.right.right; // Node 4

        TreeNode lca = lcaFinder.findLCA(root, p, q);
        System.out.println("Lowest Common Ancestor: " + (lca != null ? lca.val : "None"));
    }
}
```

2. Construct Binary Tree from Preorder and Inorder Traversals.

Code:-

```
import java.util.HashMap;
import java.util.Map;
```

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int val) {
        this.val = val;
    }
}

public class ConstructBinaryTree {
    private Map<Integer, Integer> inorderIndexMap;
    private int preorderIndex;

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        inorderIndexMap = new HashMap<>();
        preorderIndex = 0;

        // Store the index of each value in the inorder array
        for (int i = 0; i < inorder.length; i++) {
            inorderIndexMap.put(inorder[i], i);
        }

        return buildTreeRecursive(preorder, 0, inorder.length - 1);
    }

    private TreeNode buildTreeRecursive(int[] preorder, int inorderStart, int inorderEnd) {
        // Base case
        if (inorderStart > inorderEnd) {
            return null;
        }

        // Get the current root value from preorder traversal
        int rootValue = preorder[preorderIndex++];
        TreeNode root = new TreeNode(rootValue);

        // Get the index of the root value in the inorder array
        int inorderIndex = inorderIndexMap.get(rootValue);

        // Recursively build the left and right subtrees
        root.left = buildTreeRecursive(preorder, inorderStart, inorderIndex - 1);
        root.right = buildTreeRecursive(preorder, inorderIndex + 1, inorderEnd);

        return root;
    }

    // Helper method to print the tree (inorder traversal)
```

```
    public void printInorder(TreeNode root) {
        if (root != null) {
            printInorder(root.left);
            System.out.print(root.val + " ");
            printInorder(root.right);
        }
    }

    public static void main(String[] args) {
        int[] preorder = {3, 9, 20, 15, 7};
        int[] inorder = {9, 3, 15, 20, 7};

        ConstructBinaryTree treeBuilder = new ConstructBinaryTree();
        TreeNode root = treeBuilder.buildTree(preorder, inorder);

        System.out.println("Inorder traversal of the constructed tree:");
        treeBuilder.printInorder(root);
    }
}
```

3. Check if Two Trees are Identical.

Code:-

```
// Definition for a binary tree node
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class IdenticalTrees {

    // Function to check if two trees are identical
    public static boolean areIdentical(TreeNode root1, TreeNode root2) {
        // If both trees are null, they are identical
        if (root1 == null && root2 == null) {
            return true;
        }

        // If one of the trees is null and the other is not, they are not identical
        if (root1 == null || root2 == null) {
            return false;
```

```
    }

    // Check if the current nodes have the same value and
    // recursively check their left and right subtrees
    return (root1.val == root2.val) &&
        areIdentical(root1.left, root2.left) &&
        areIdentical(root1.right, root2.right);
  }

  public static void main(String[] args) {
    // Example trees
    TreeNode tree1 = new TreeNode(1);
    tree1.left = new TreeNode(2);
    tree1.right = new TreeNode(3);
    tree1.left.left = new TreeNode(4);
    tree1.left.right = new TreeNode(5);

    TreeNode tree2 = new TreeNode(1);
    tree2.left = new TreeNode(2);
    tree2.right = new TreeNode(3);
    tree2.left.left = new TreeNode(4);
    tree2.left.right = new TreeNode(5);

    // Check if the trees are identical
    if (areIdentical(tree1, tree2)) {
      System.out.println("The two trees are identical.");
    } else {
      System.out.println("The two trees are not identical.");
    }
  }
}
```

4. Serialize and Deserialize a Binary Tree.

Code:-

```
import java.util.*;

// Definition for a binary tree node
class TreeNode {
  int val;
  TreeNode left;
  TreeNode right;

  TreeNode(int val) {
    this.val = val;
  }
}
```

```java
public class SerializeDeserializeBinaryTree {

    // Serializes a tree to a single string
    public String serialize(TreeNode root) {
        if (root == null) {
            return "null";
        }

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

        while (!queue.isEmpty()) {
            TreeNode node = queue.poll();

            if (node == null) {
                sb.append("null,");
            } else {
                sb.append(node.val).append(",");
                queue.add(node.left);
                queue.add(node.right);
            }
        }

        // Remove the trailing comma
        sb.setLength(sb.length() - 1);
        return sb.toString();
    }

    // Deserializes a string to a tree
    public TreeNode deserialize(String data) {
        if (data.equals("null")) {
            return null;
        }

        String[] values = data.split(",");
        TreeNode root = new TreeNode(Integer.parseInt(values[0]));
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);

        int i = 1;
        while (!queue.isEmpty() && i < values.length) {
            TreeNode current = queue.poll();

            if (!values[i].equals("null")) {
                current.left = new TreeNode(Integer.parseInt(values[i]));
                queue.add(current.left);
            }
```

```
            i++;

            if (i < values.length && !values[i].equals("null")) {
                current.right = new TreeNode(Integer.parseInt(values[i]));
                queue.add(current.right);
            }
            i++;
        }

        return root;
    }

    public static void main(String[] args) {
        SerializeDeserializeBinaryTree codec = new SerializeDeserializeBinaryTree();

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

        // Serialize the tree
        String serializedTree = codec.serialize(root);
        System.out.println("Serialized Tree: " + serializedTree);

        // Deserialize the string back to a tree
        TreeNode deserializedTree = codec.deserialize(serializedTree);
        System.out.println("Deserialized Tree (root value): " + deserializedTree.val);
    }
}
```

5. Diameter of a Binary Tree.

Code:-

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

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

public class BinaryTreeDiameter {
    // Class variable to store the diameter
    private int diameter = 0;
```

```
    // Main function to calculate the diameter of a binary tree
    public int diameterOfBinaryTree(TreeNode root) {
        calculateHeight(root);
        return diameter;
    }

    // Helper function to calculate the height of the tree
    private int calculateHeight(TreeNode node) {
        if (node == null) {
            return 0;
        }

        // Recursively find the height of the left and right subtrees
        int leftHeight = calculateHeight(node.left);
        int rightHeight = calculateHeight(node.right);

        // Update the diameter (longest path between any two nodes)
        diameter = Math.max(diameter, leftHeight + rightHeight);

        // Return the height of the current node
        return Math.max(leftHeight, rightHeight) + 1;
    }

    // Main method for testing
    public static void main(String[] args) {
        // Example binary tree:
        //        1
        //       / \
        //      2   3
        //     / \
        //    4   5

        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);

        BinaryTreeDiameter btd = new BinaryTreeDiameter();
        System.out.println("Diameter of the tree: " + btd.diameterOfBinaryTree(root));
    }
}
```

7.K-th Smallest Element in a BST.

Code:-

```
class TreeNode {
```

```
    int val;
    TreeNode left, right;

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

public class KthSmallestElement {

    private static int count = 0;
    private static int result = -1;

    public static int kthSmallest(TreeNode root, int k) {
        count = 0; // Reset the count for each call
        result = -1; // Reset the result for each call
        inOrderTraversal(root, k);
        return result;
    }

    private static void inOrderTraversal(TreeNode node, int k) {
        if (node == null) return;

        // Traverse the left subtree
        inOrderTraversal(node.left, k);

        // Visit the current node
        count++;
        if (count == k) {
            result = node.val;
            return; // Stop traversal once we find the k-th element
        }

        // Traverse the right subtree
        inOrderTraversal(node.right, k);
    }

    public static void main(String[] args) {
        // Example: Constructing a BST
        TreeNode root = new TreeNode(5);
        root.left = new TreeNode(3);
        root.right = new TreeNode(6);
        root.left.left = new TreeNode(2);
        root.left.right = new TreeNode(4);
        root.left.left.left = new TreeNode(1);

        int k = 3;
```

```
        System.out.println("The " + k + "-th smallest element is: " + kthSmallest(root, k));
    }
}
```

6. Level Order Traversal (Spiral Form).

Code:-

```
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int val) {
        this.val = val;
    }
}

public class ZigzagLevelOrder {

    // Method to print level order traversal in spiral form (zigzag)
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) {
            return result;
        }

        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        boolean leftToRight = true;

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            List<Integer> levelNodes = new ArrayList<>();

            for (int i = 0; i < levelSize; i++) {
                TreeNode node = queue.poll();

                // Add node's value to the list based on the direction
                if (leftToRight) {
                    levelNodes.add(node.val);
                } else {
                    levelNodes.add(0, node.val); // Add to the front for right-to-left order
                }
```

```
                // Add child nodes to the queue
                if (node.left != null) queue.add(node.left);
                if (node.right != null) queue.add(node.right);
            }

            // Toggle the direction for the next level
            leftToRight = !leftToRight;
            result.add(levelNodes);
        }

        return result;
    }

    // Main method for testing
    public static void main(String[] args) {
        ZigzagLevelOrder treeTraverser = new ZigzagLevelOrder();

        // Example Tree
        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);

        // Printing zigzag level order
        List<List<Integer>> result = treeTraverser.zigzagLevelOrder(root);
        for (List<Integer> level : result) {
            System.out.println(level);
        }
    }
}
```