Experiment 5

Student Name: Shristi Rai **UID: 22BCS15330**
Branch: CSE **Section/Group: NTPP-602-A**
Semester: 6th **Date of Performance: 17-02-25**
Subject Name: AP-II LAB **Subject Code: 22CSP-352**

PROBLEM 1

1. Aim:

Given the root of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

2. Objective:

- Understand how to determine the maximum depth of a binary tree.
- Develop an efficient approach using recursion for optimal performance.
- Learn to handle edge cases such as empty trees.

3. Algorithm:

1. If the root is null, return 0 (base case).
2. Recursively compute the depth of the left and right subtrees.
3. Return the maximum of these two depths plus 1 (to count the current node).

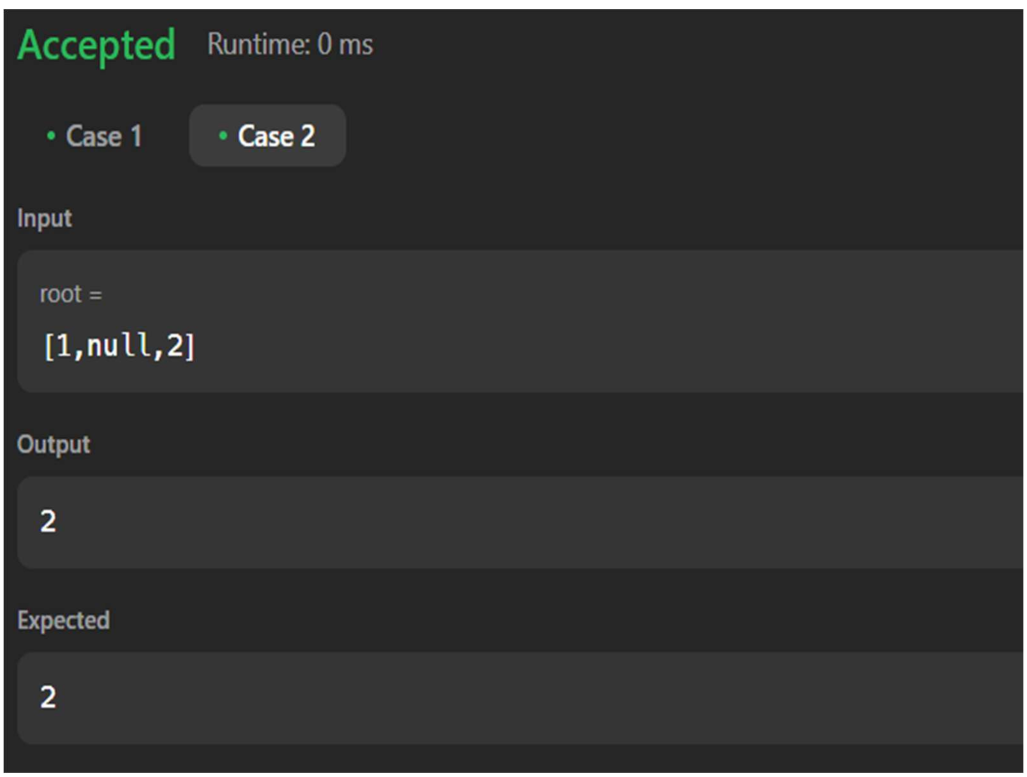
4. Implementation/Code:

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

```
    TreeNode(int x) {
        val = x;
        left = null;
        right = null;
    }
}

class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }

        int leftDepth = maxDepth(root.left);
        int rightDepth = maxDepth(root.right);

        return Math.max(leftDepth, rightDepth) + 1;
    }

    public static void main(String[] args) {
        TreeNode root = new TreeNode(3);
        root.left = new TreeNode(9);
        root.right = new TreeNode(20);
        root.right.left = new TreeNode(15);
        root.right.right = new TreeNode(7);



        Solution sol = new Solution();
        System.out.println("Maximum Depth: " + sol.maxDepth(root)); //
    }
 }
```

5. Output:


Accepted Runtime: 0 ms
• Case 1
• Case 2
Input
root =
[3,9,20,null,null,15,7]
Output
3
Expected
3


Accepted Runtime: 0 ms
• Case 1
• Case 2
Input
root =
[1,null,2]
Output
2
Expected
2

6. **Time Complexity:** O(n)

7. **Space Complexity:** O(n)

8. **Learning Outcomes:**

- Ability to implement a recursive solution for tree traversal.
- Understanding of depth calculation in a binary tree.
- Improved skills in handling recursion-based problems.

PROBLEM 2

1. **Aim:**
   Construct Binary Tree from Inorder and Postorder Traversal
   Given two integer arrays inorder and postorder where inorder is the inorder traversal of a binary tree and postorder is the postorder traversal of the same tree, construct and return *the binary tree*.

2. **Objective:**

   a. Understand how to construct a binary tree from inorder and postorder traversals.
   b. Develop an efficient approach using recursion and hash maps for fast lookups.
   c. Learn to handle constraints like unique values and guaranteed valid traversals.

3. **Algorithm:**

   1. Create a hash map to store indices of inorder values for quick access.
   2. Define a recursive function to construct the tree:
      - Pick the last element from postorder as the root.
      - Find this root's index in inorder to determine left & right subtrees.
      - Recursively construct left and right subtrees.
   3. Return the constructed tree.

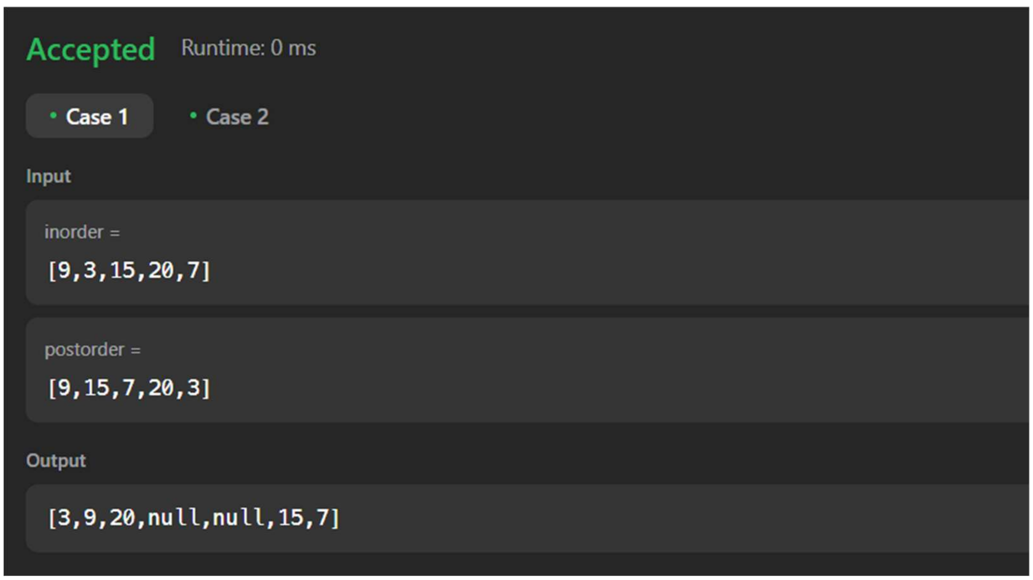
4. **Implementation/Code:**

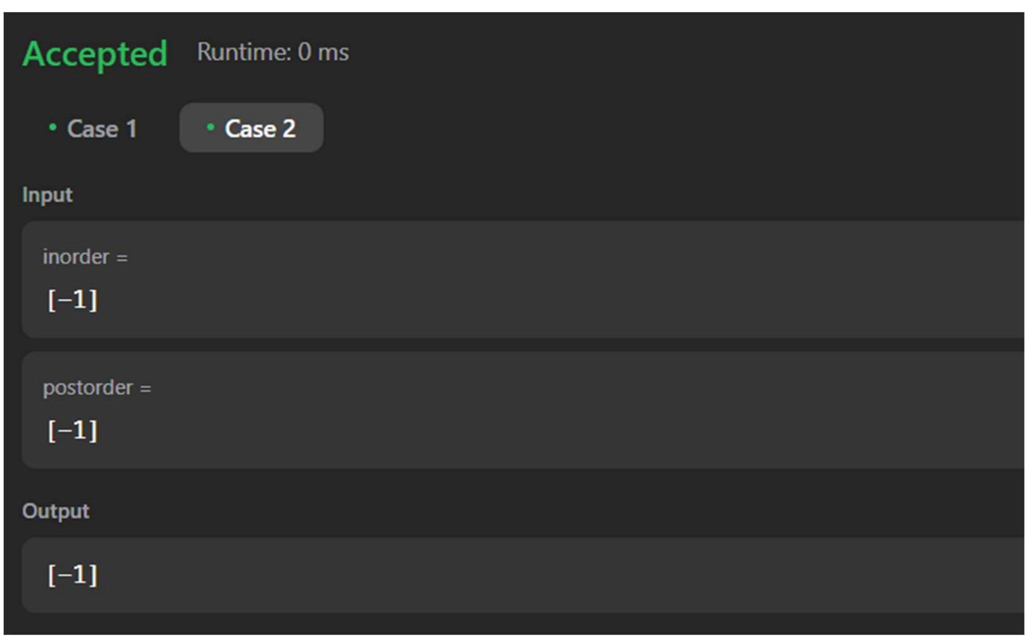
```
import java.util.*;

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

```
    TreeNode(int x) {
        val = x;
        left = null;
        right = null;
    }
}

class Solution {
    private Map<Integer, Integer> inorderMap;
    private int postIndex;

    public TreeNode buildTree(int[] inorder, int[] postorder) {
        inorderMap = new HashMap<>();
        postIndex = postorder.length - 1;

        for (int i = 0; i < inorder.length; i++) {
            inorderMap.put(inorder[i], i);
        }

        return constructTree(postorder, 0, inorder.length - 1);
    }

    private TreeNode constructTree(int[] postorder, int left, int right) {
        if (left > right) {
            return null;
        }

        int rootValue = postorder[postIndex--];
        TreeNode root = new TreeNode(rootValue);

        int inorderIndex = inorderMap.get(rootValue);

        root.right = constructTree(postorder, inorderIndex + 1, right);
        root.left = constructTree(postorder, left, inorderIndex - 1);

        return root;
    }

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

```
        Solution sol = new Solution();
        TreeNode root = sol.buildTree(inorder, postorder);

        System.out.println("Binary Tree constructed from inorder and postorder
      traversal");
    }
}
```

5. Output:





6. **Time Complexity:** O(n)

7. **Space Complexity:** O(n)

8. **Learning Outcomes:**

   a. Ability to implement a recursive solution for tree traversal.
   b. Understanding of recursive tree-building techniques.
   c. Improved skills in handling recursion-based problems.

PROBLEM 3

1. **Aim:**

   Binary Tree Level Order Traversal

   Given the root of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

   Example:

   Input: root = [1]

   Output: [[1]]

2. **Objective:**

   - Understand how to perform level order traversal of a binary tree.
   - Develop an efficient approach using a queue (BFS) for optimal performance.
   - Learn to handle edge cases such as empty trees.

3. **Algorithm:**

   1. If the root is null, return an empty list.
   2. Initialize a queue and add the root node.
   3. While the queue is not empty:
   4. Determine the number of nodes at the current level.
   5. Process each node, adding its value to the result list.
   6. Add the left and right children of each node to the queue.
   7. Return the list of levels.

4. Implementation/Code:

Code

Java Auto

```java
import java.util.*;
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int x) {
        val = x;
        left = null;
        right = null;
    }
}
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) {
            return result;
        }

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

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

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

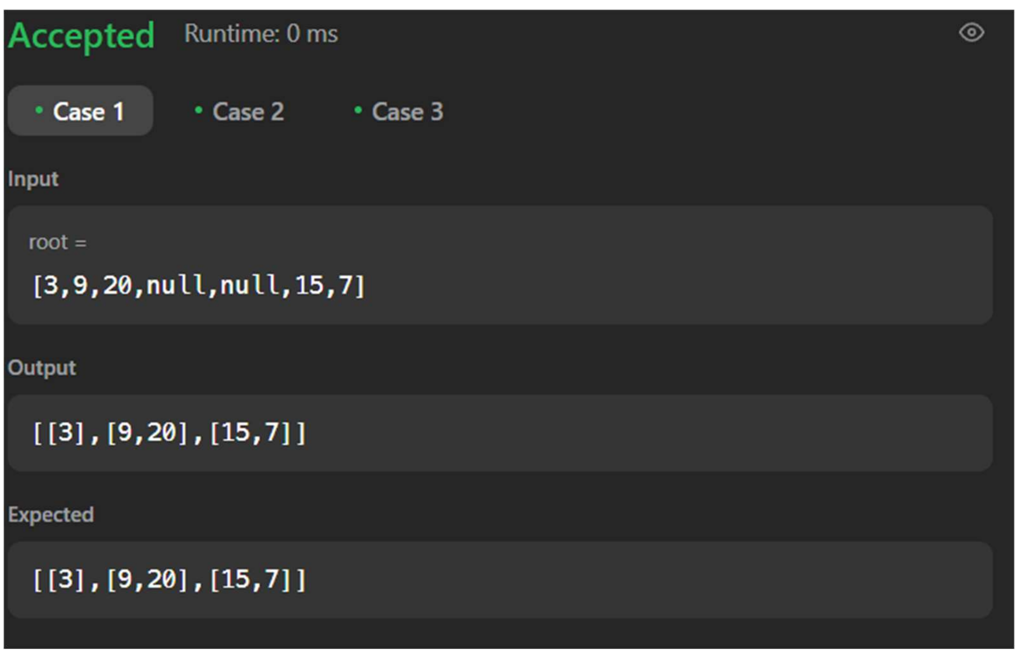
Code

Java Auto

```java
                TreeNode node = queue.poll();
                currentLevel.add(node.val);

                if (node.left != null) {
                    queue.add(node.left);
                }
                if (node.right != null) {
                    queue.add(node.right);
                }
            }
            result.add(currentLevel);
        }

        return result;
    }

    public static void main(String[] args) {
        TreeNode root = new TreeNode(3);
        root.left = new TreeNode(9);
        root.right = new TreeNode(20);
        root.right.left = new TreeNode(15);
        root.right.right = new TreeNode(7);

        Solution sol = new Solution();
        System.out.println("Level Order Traversal: " + sol.levelOrder(root)); // Output: [[3], [9, 20], [15, 7]]
    }
}
```

5. **Output:**


Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
root =
[1]
Output
[[1]]
Expected
[[1]]


Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
root =
[3,9,20,null,null,15,7]
Output
[[3],[9,20],[15,7]]
Expected
[[3],[9,20],[15,7]]

6. **Time Complexity:** O(n), where n is the number of nodes in the tree.

7. **Space Complexity:** O(n)

8. **Learning Outcomes:**

- Ability to implement level order traversal using a queue.
- Understanding of breadth-first search (BFS) in tree traversal.
- Improved problem-solving skills in handling binary tree structures.