Experiment 6

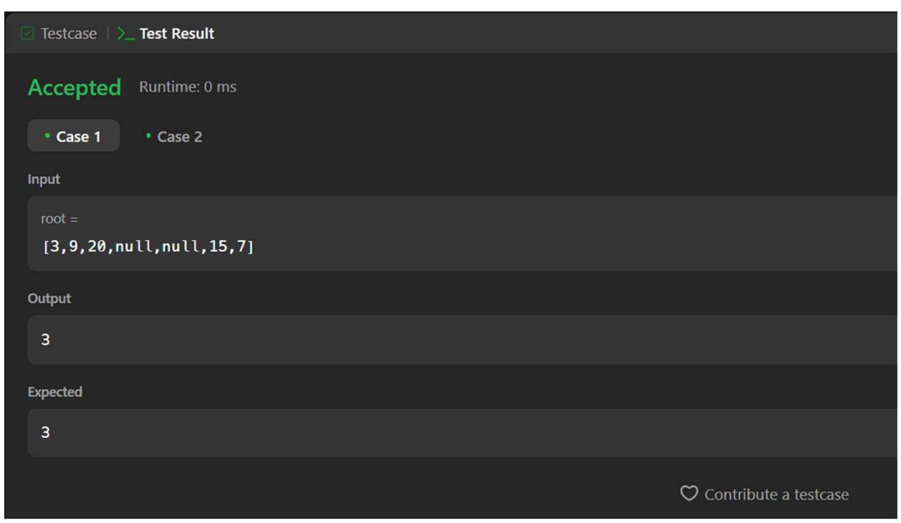
Student Name: Ayushi Gupta **UID: 22BET10133**
Branch: BE-IT **Section/Group: 22BET_IOT_702/B**
Semester: 6th **Date of Performance: 07-2-25**
Subject Name: Advanced Programming Lab-2 **Subject Code: 22ITP-351**

Problem 1

1. **Aim :** To finding the maximum depth of a binary tree is to determine the longest path from the root node to any leaf node, which helps assess the tree's structure and performance.
2. **Code:**

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        }
        int leftDepth = maxDepth(root->left);
        int rightDepth = maxDepth(root->right);
        return max(leftDepth, rightDepth) + 1;
    }
};
```

3. **Output:**



Problem 2

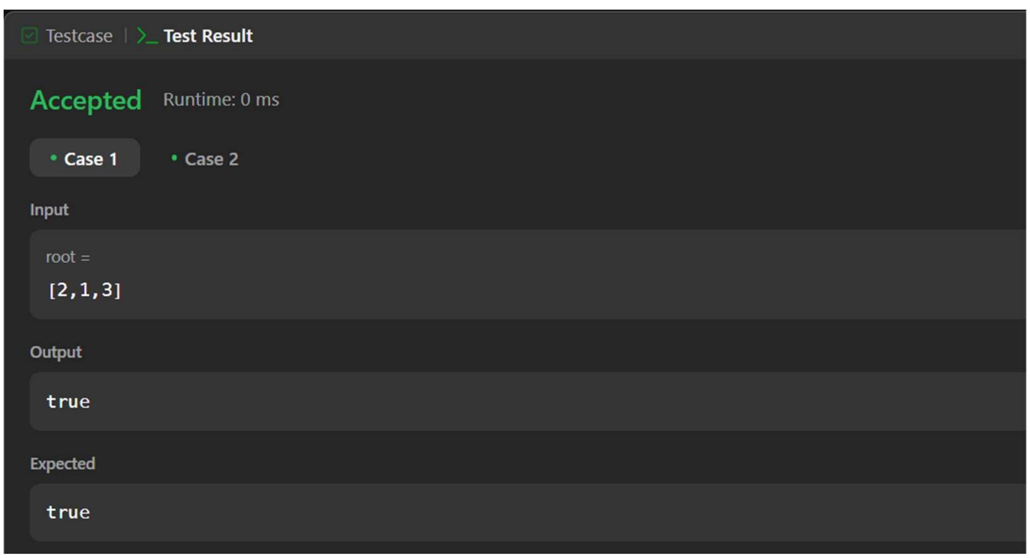
1. **Aim:** Validating a Binary Search Tree (BST) involves checking that for every node, its left subtree contains only nodes with values less than the node, and its right subtree contains only nodes with values greater than the node, recursively for all nodes.

2. **Code:**

```
class Solution {
  public:
     bool isValidBST(TreeNode* root) {
        return isValidBSTHelper(root, LONG_MIN, LONG_MAX);
     }

     bool isValidBSTHelper(TreeNode* root, long minVal, long maxVal) {
        if (root == nullptr) {
           return true;
        }
        if (root->val <= minVal || root->val >= maxVal) {
           return false;
        }
        return isValidBSTHelper(root->left, minVal, root->val) &&
             isValidBSTHelper(root->right, root->val, maxVal);
     }
  };
```

3. **Output:**



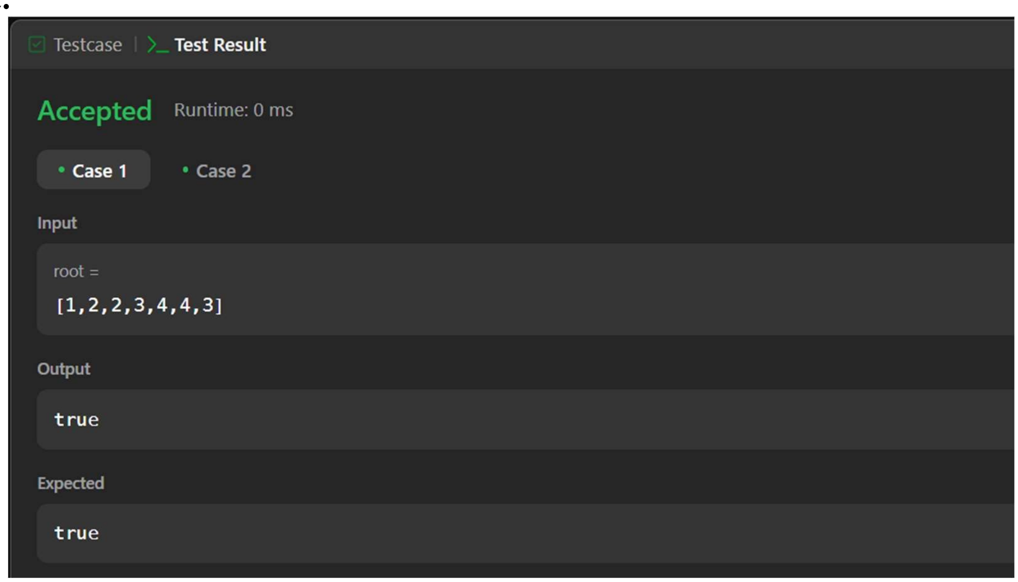
Problem 3

1. **Aim:** A symmetric tree is a binary tree that is a mirror image of itself, meaning its left and right subtrees are structurally identical and have matching values in corresponding positions.
2. **Code:**

```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if (root == nullptr) {
            return true;
        }
        return isMirror(root->left, root->right);
    }

    bool isMirror(TreeNode* left, TreeNode* right) {
        if (left == nullptr && right == nullptr) {
            return true;
        }
        if (left == nullptr || right == nullptr) {
            return false;
        }
        return (left->val == right->val) && isMirror(left->left, right->right) &&
            isMirror(left->right, right->left);
    }
};
```

3. **Output:**



Problem 4

1. **Aim:** Binary Tree Level Order Traversal is a breadth-first traversal method where nodes are visited level by level, starting from the root and moving down to each subsequent level.
2. **Code:**

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> result;
        if (root == nullptr) {
            return result;
        }
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            int levelSize = q.size();
            vector<int> currentLevel;
            for (int i = 0; i < levelSize; ++i) {
                TreeNode* node = q.front();
                q.pop();
                currentLevel.push_back(node->val);
                if (node->left) {
                    q.push(node->left);
                }
                if (node->right) {
                    q.push(node->right);
                }
            }
            result.push_back(currentLevel);
        }
        return result;
    }
};
```

3. **Output:**

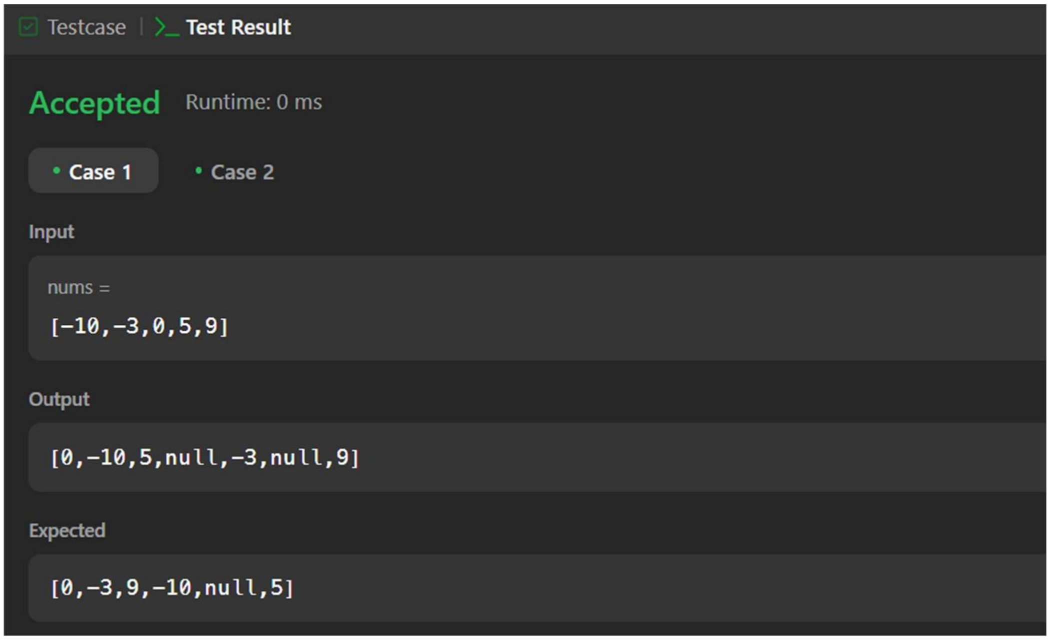
Problem 5

1. **Aim:** Converting a sorted array to a Binary Search Tree involves recursively selecting the middle element of the array as the root, and then applying the same process to the left and right halves of the array to form the left and right subtrees.
2. **Code:**

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return sortedArrayToBSTHelper(nums, 0, nums.size() - 1);
    }

    TreeNode* sortedArrayToBSTHelper(vector<int>& nums, int start, int end) {
        if (start > end) {
            return nullptr;
        }
        int mid = start + (end - start) / 2;
        TreeNode* root = new TreeNode(nums[mid]);
        root->left = sortedArrayToBSTHelper(nums, start, mid - 1);
        root->right = sortedArrayToBSTHelper(nums, mid + 1, end);
        return root;
    }
};
```

3. **Output:**



Problem 6

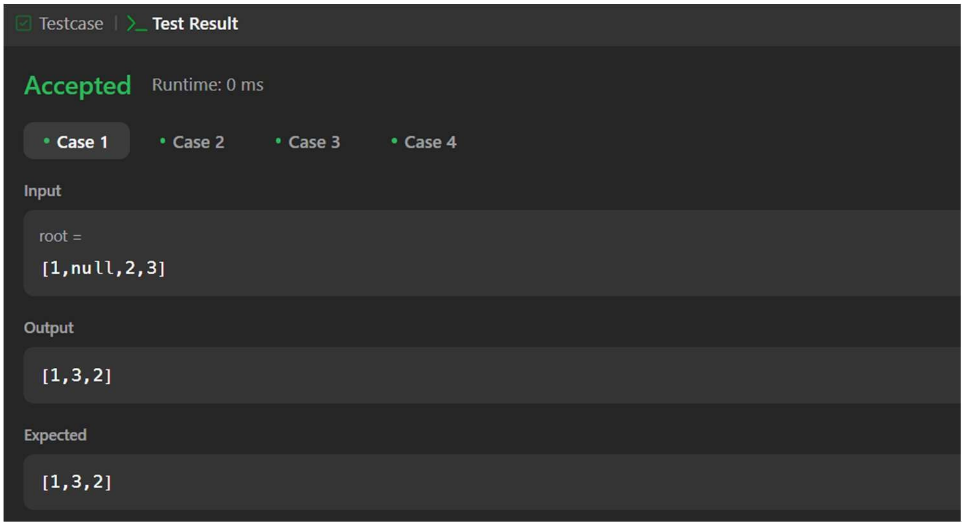
1. **Aim:** Binary Tree Inorder Traversal is a depth-first traversal method where the nodes are visited in the following order: left subtree, root node, and right subtree**.**

2. **Code:**

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        inorderTraversalHelper(root, result);
        return result;
    }

    void inorderTraversalHelper(TreeNode* root, vector<int>& result) {
        if (root == nullptr) {
            return;
        }
        inorderTraversalHelper(root->left, result);
        result.push_back(root->val);
        inorderTraversalHelper(root->right, result);
    }
};
```

3. Output:



Problem 7

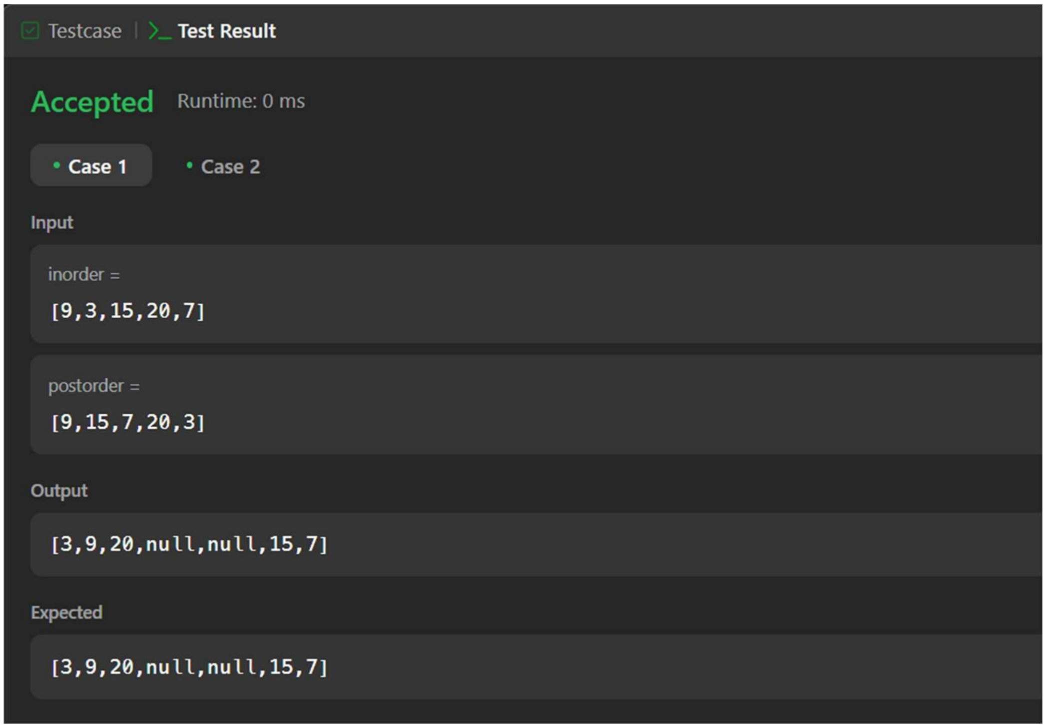
1. **Aim:** Constructing a binary tree from inorder and postorder traversal involves recursively using the last element of the postorder array as the root, finding its index in the inorder array to divide the tree into left and right subtrees, and repeating this process for each subtree.
2. **Code:**

```
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        int postIndex = postorder.size() - 1;
        unordered_map<int, int> inMap;
        for (int i = 0; i < inorder.size(); ++i) {
            inMap[inorder[i]] = i;
        }
        return buildTreeHelper(inorder, postorder, 0, inorder.size() - 1, postIndex, inMap);
    }

    TreeNode* buildTreeHelper(vector<int>& inorder, vector<int>& postorder, int inStart, int inEnd,
int& postIndex, unordered_map<int, int>& inMap) {
        if (inStart > inEnd) {
            return nullptr;
        }
        TreeNode* root = new TreeNode(postorder[postIndex]);
        postIndex--;
        int inRoot = inMap[root->val];
        root->right = buildTreeHelper(inorder, postorder, inRoot + 1, inEnd, postIndex, inMap);
```

```
        root->left = buildTreeHelper(inorder, postorder, inStart, inRoot - 1, postIndex, inMap);
        return root;
    }
};
```

3. **Output:**



Problem 8

1. **Aim:** The Kth smallest element in a Binary Search Tree (BST) can be found by performing an inorder traversal and returning the Kth element visited, as the inorder traversal of a BST produces elements in ascending order.
2. **Code:**

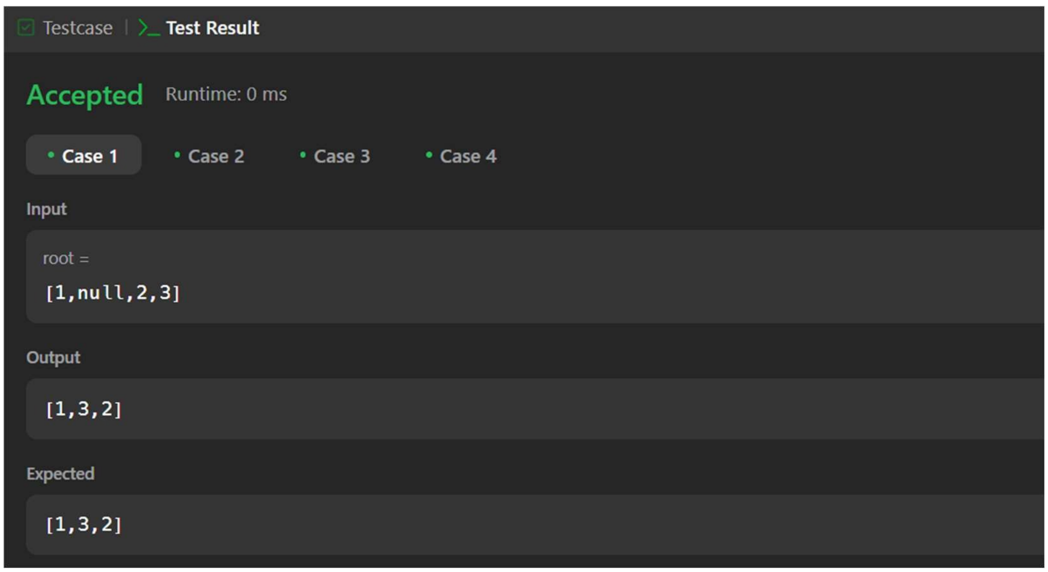
```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        int count = 0;
        int result = 0;
        kthSmallestHelper(root, k, count, result);
        return result;
    }

    void kthSmallestHelper(TreeNode* root, int k, int& count, int& result) {
        if (root == nullptr) {
            return;
        }
```

```
        kthSmallestHelper(root->left, k, count, result);
        count++;
        if (count == k) {
            result = root->val;
            return;
        }

        kthSmallestHelper(root->right, k, count, result);
    }
};
```

3. **Output:**



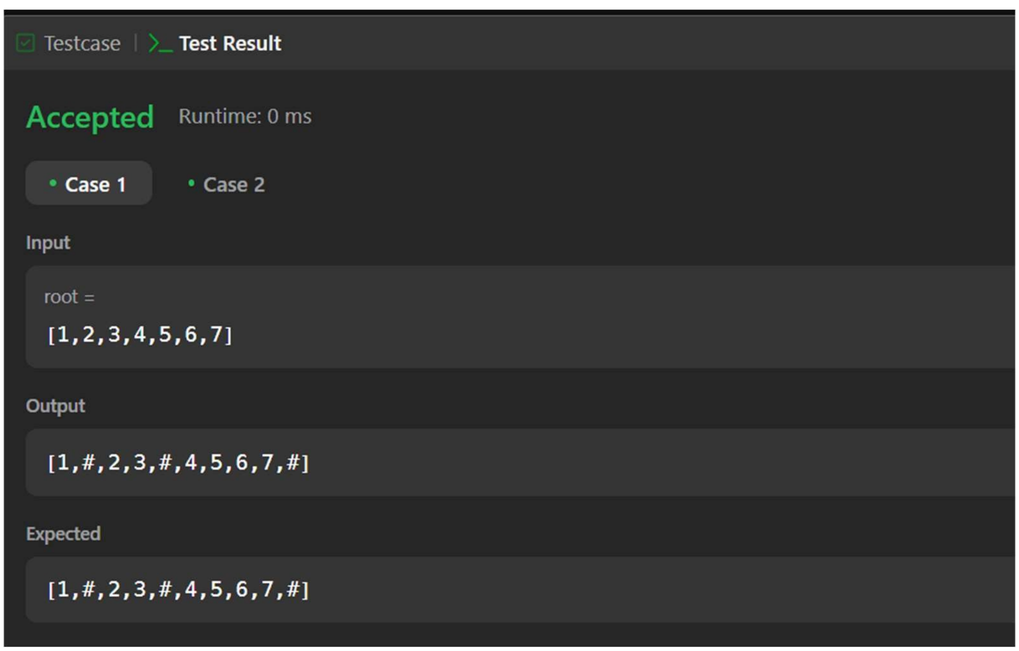
Problem 9

1. **Aim:** Populating next right pointers in each node involves setting the `next` pointer of each node to the node directly to its right at the same level, using level-order traversal or a modified depth-first search to connect the nodes level by level.
2. **Code:**

```
class Solution {
public:
    Node* connect(Node* root) {
        if (root == nullptr) {
            return nullptr;
        }
        queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            int levelSize = q.size();
            for (int i = 0; i < levelSize; ++i) {
                Node* node = q.front();
                q.pop();
```

```
                    if (i < levelSize - 1) {
                        node->next = q.front();
                    } else {
                        node->next = nullptr;
                    }
                    if (node->left) {
                        q.push(node->left);
                    }
                    if (node->right) {
                        q.push(node->right);
                    }
                }
            }
            return root;
        }
};
```

3. **Output:**



Problem 10

1. **Aim:** Binary Tree Inorder Traversal is a depth-first traversal method where nodes are visited in this specific order: traverse the left subtree, visit the root node, and then traverse the right subtree.
2. **Code:**

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        inorderTraversalHelper(root, result);
        return result;
    }

    void inorderTraversalHelper(TreeNode* root, vector<int>& result) {
        if (root == nullptr) {
            return;
        }
        inorderTraversalHelper(root->left, result);
        result.push_back(root->val);
        inorderTraversalHelper(root->right, result);
    }
};
```

3. **Output:**