Experiment-6

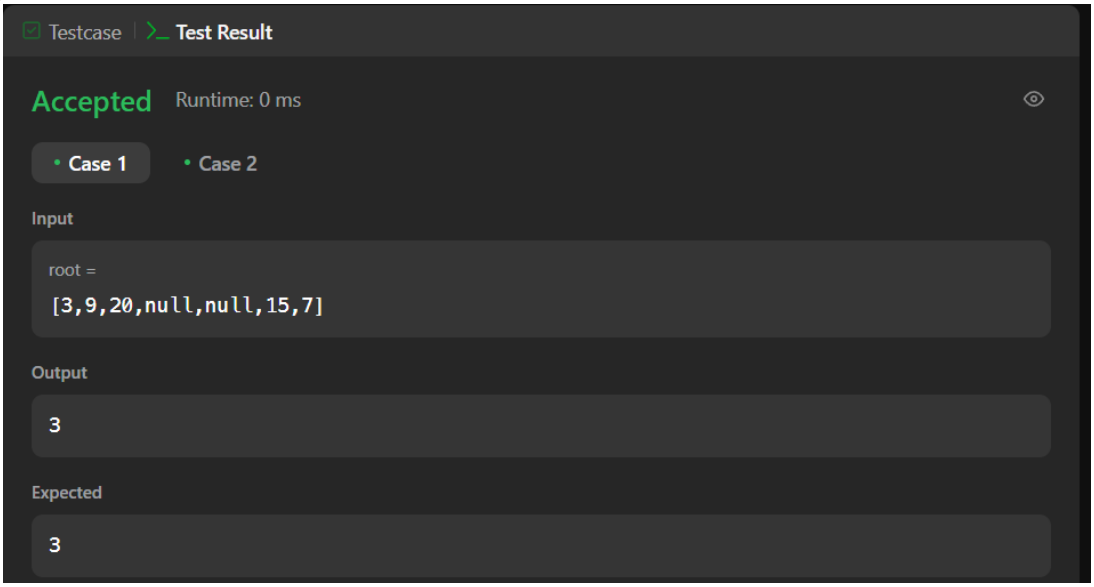
Name: Jatin Gautam **UID: 22BET10252**
Branch: BE-IT **Section/Group: 22BET_702-B**
Semester: 6 **Date of Performance: 07-03-25**
Subject Name: Advanced Programming Lab-2 **Subject Code: 22ITP-351**

Problem 1 -Maximum Depth of Binary Tree - To determine the maximum depth (or height) of a binary tree, which represents the longest path from the root node to a leaf node.

Code:

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

Output:

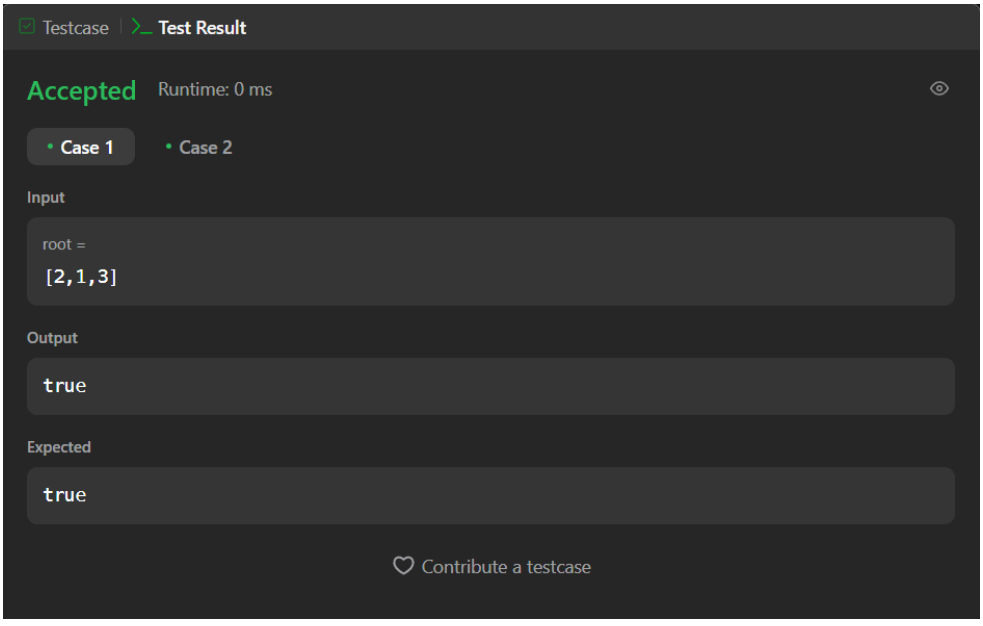


Problem 2 :- Validate Binary Search Tree- To verify if a given binary tree is a valid Binary Search Tree (BST) where the left subtree nodes are smaller, and the right subtree nodes are larger than the root.

Code:

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

private:
    bool valid(TreeNode* node, long minimum, long maximum) {
        if (!node) return true;

        if (!(node->val > minimum && node->val < maximum)) return false;

        return valid(node->left, minimum, node->val) && valid(node->right, node->val,
maximum);
    }
};
```

Output:

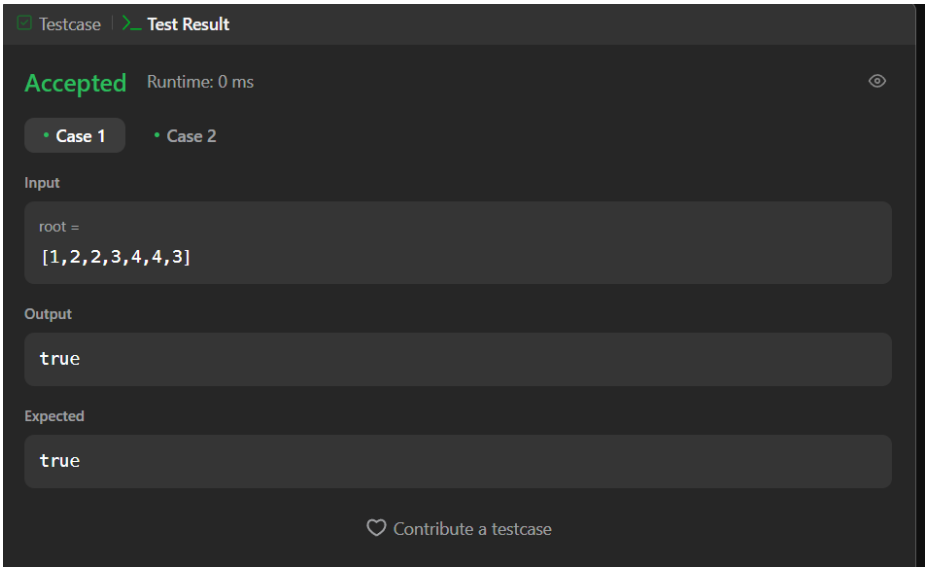


Problem 3:- Symmetric Tree- To determine if a binary tree is symmetric, meaning it is a mirror image of itself around its center.

Code:

```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        return isMirror(root->left, root->right);
    }
private:
    bool isMirror(TreeNode* n1, TreeNode* n2) {
        if (n1 == nullptr && n2 == nullptr) {
            return true;
        }

        if (n1 == nullptr || n2 == nullptr) {
            return false;
        }

        return n1->val == n2->val && isMirror(n1->left, n2->right) && isMirror(n1->right, n2->left);
    }
};
```

Output:



Problem 4:- Binary Tree Level Order Traversal- To perform a level-order traversal of a binary tree, returning nodes level by level from top to bottom.

Code:

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        if (!root) return ans;

        queue<TreeNode*> q;
        q.push(root);

        while (!q.empty()) {
            int level_size = q.size();
            vector<int> level;

            for (int i = 0; i < level_size; ++i) {
                TreeNode* node = q.front();
                q.pop();
                level.push_back(node->val);

                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }

            ans.push_back(level);
        }

        return ans;
    }
};
```

Output:

Problem 5. Convert Sorted Array to Binary Search Tree- To convert a sorted array into a height-balanced Binary Search Tree (BST).

Code:

```
#include <vector>
using namespace std;

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

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

Output:

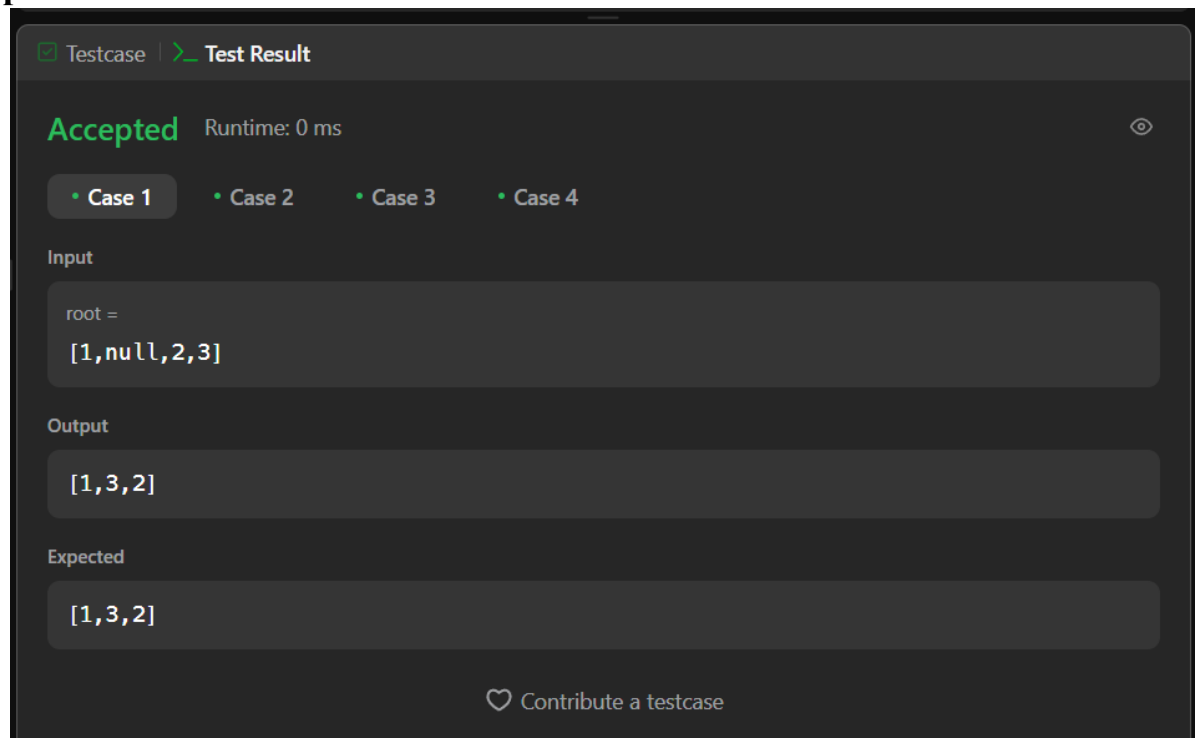
Problem 6. Binary Tree Inorder Traversal- To perform an in-order traversal of a binary tree and return the node values in left-root-right order.

Code:

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        stack<TreeNode*> stack;
        TreeNode* curr = root;

        while (curr != nullptr || !stack.empty()) {
            while (curr != nullptr) {
                stack.push(curr);
                curr = curr->left;
            }
            curr = stack.top();
            stack.pop();
            result.push_back(curr->val);
            curr = curr->right;
        }

        return result;
```

```
    }
};
```

Output:



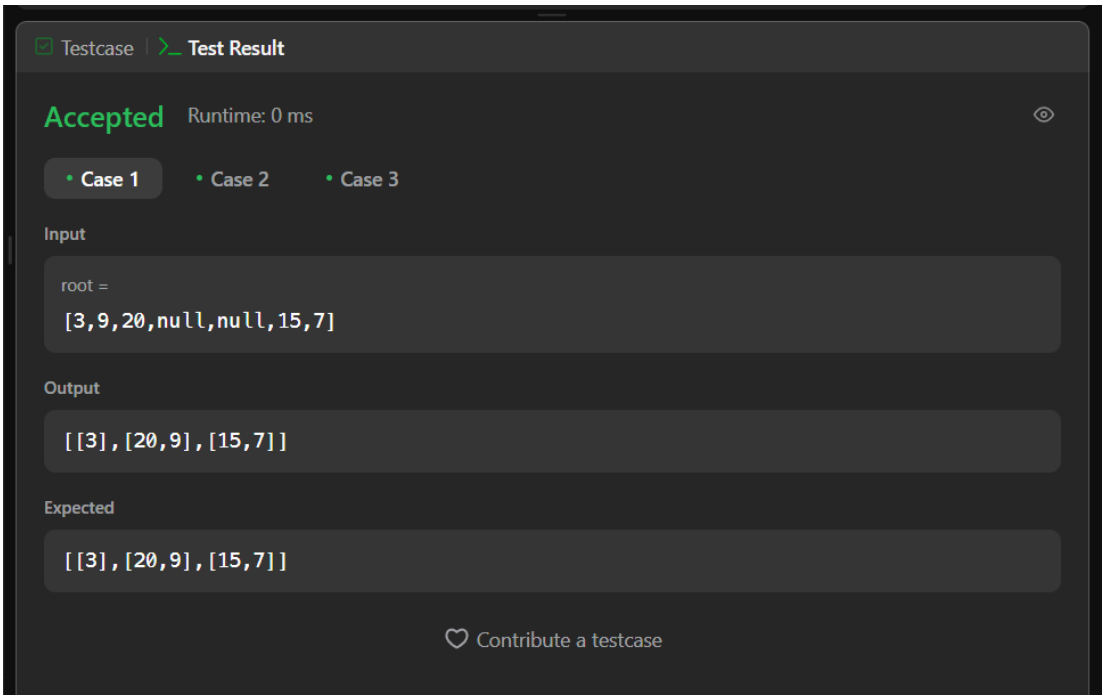
Problem 7:- Binary Zigzag Level Order Traversal-To perform a zigzag level order traversal of a binary tree, where nodes are traversed left-to-right on one level and right-to-left on the next.

Code:

```
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        if (!root) return {};
        vector<vector<int>> result;
        queue<TreeNode*> q;
        q.push(root);
        bool leftToRight = true;

        while (!q.empty()) {
            int levelSize = q.size();
            vector<int> level(levelSize);
            for (int i = 0; i < levelSize; ++i) {
                TreeNode* node = q.front();
                q.pop();
```

```
                int index = leftToRight ? i : (levelSize - 1 - i);
                level[index] = node->val;

                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }
            leftToRight = !leftToRight;
            result.push_back(level);
        }

        return result;
    }
};
```

Output:

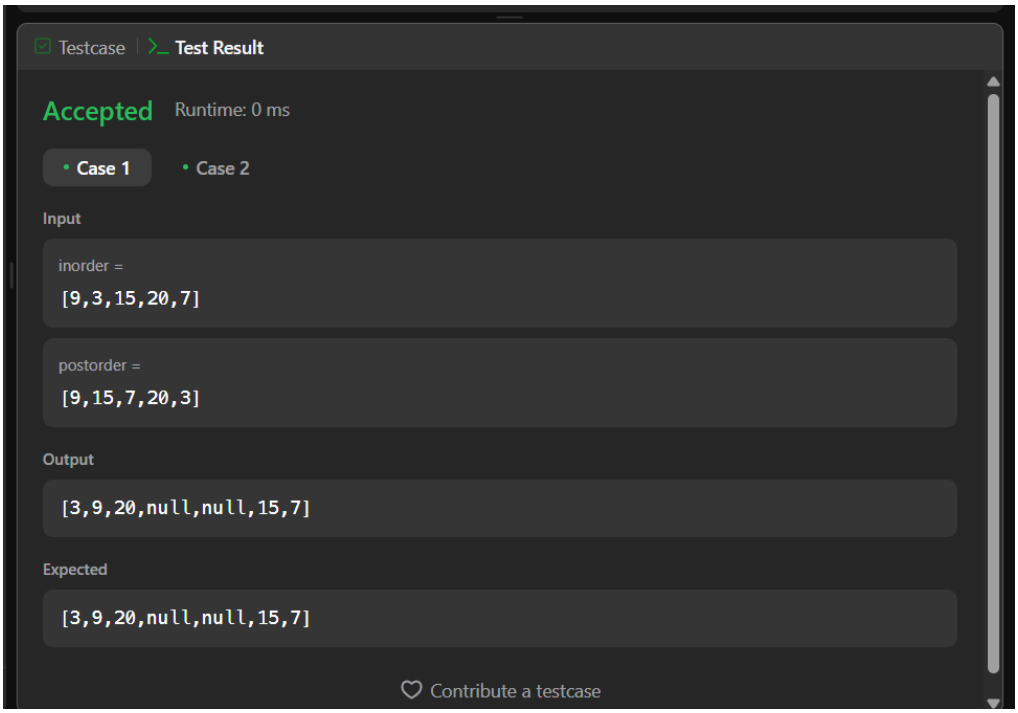


Problem 8:-Construct Binary Tree from Inorder and Postorder Traversal- To construct a binary tree from given inorder and postorder traversal sequences.

Code:-

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

    TreeNode* buildTreeHelper(vector<int>& inorder, vector<int>& postorder, int
inorderStart, int inorderEnd, int postorderStart, int postorderEnd, unordered_map<int, int>&
index) {
        if (inorderStart > inorderEnd || postorderStart > postorderEnd) {
            return nullptr;
        }
        int rootVal = postorder[postorderEnd];
        TreeNode* root = new TreeNode(rootVal);
        int inorderRootIndex = index[rootVal];
        int leftSubtreeSize = inorderRootIndex - inorderStart;
        root->left = buildTreeHelper(inorder, postorder, inorderStart, inorderRootIndex - 1,
postorderStart, postorderStart + leftSubtreeSize - 1, index);
        root->right = buildTreeHelper(inorder, postorder, inorderRootIndex + 1, inorderEnd,
postorderStart + leftSubtreeSize, postorderEnd - 1, index);
        return root;
    }
};
```

Output:-

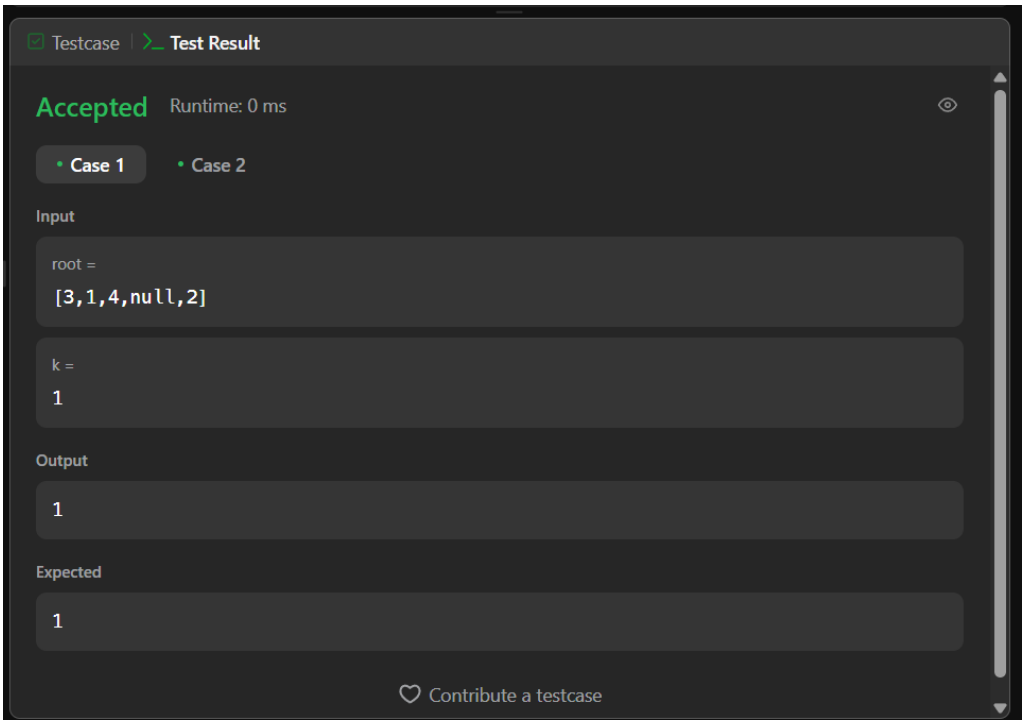


Problem 9:- Kth Smallest Element in a BST- To find the kth smallest element in a Binary Search Tree (BST).

Code:-

```
class Solution {
public:
    void preOrderTraversal(TreeNode* root, vector<int> &v){
        if(root == NULL)    return;

        //root, left, right
        v.push_back(root->val);
        preOrderTraversal(root->left, v);
        preOrderTraversal(root->right, v);
    }
    int kthSmallest(TreeNode* root, int k) {
        vector<int> v;
        preOrderTraversal(root, v);
        sort(v.begin(), v.end());
        return v[k-1];
    }
};
```

Output:-



Problem 10:-Populating Next Right Pointers in Each Node-To populate each node's next pointer to its right neighbor in a perfect binary tree.

Code:-

```
class Solution {
public:
    Node* connect(Node* root) {
        if(!root) return nullptr;
        queue<Node*> q;
        q.push(root);
        while(size(q)) {
            Node* rightNode = nullptr;
            for(int i = size(q); i; i--) {
                auto cur = q.front(); q.pop();
                cur -> next = rightNode;
                rightNode = cur;
                if(cur -> right)
                    q.push(cur -> right),
```

```
                q.push(cur -> left);
            }
        }
        return root;
    }
};
```

Output:-