Experiment 6

Name: Aditya Soni
UID: 22BET10130
Branch: IT
Section: 22BET_IOT-702/B
Semester: 6th
Date of Performance: 07/03/25
Subject: Advanced Programming - 2
Subject Code: 22ITP-351

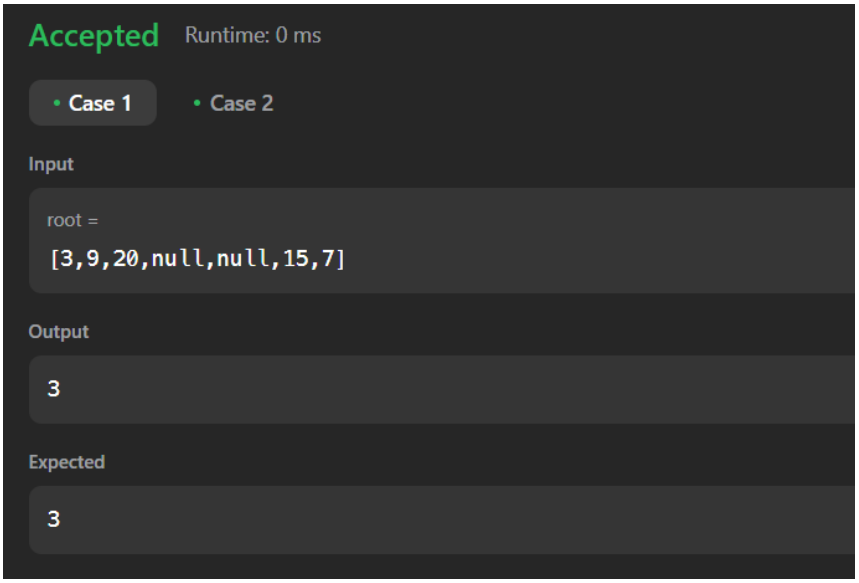
Problem 1

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.

Code:

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

Output:



Problem 2

Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*. A **valid BST** is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Code:

```
class Solution {

bool isPossible(TreeNode* root, long long l, long long r){
    if(root == nullptr)  return true;
    if(root->val < r and root->val > l)
        return isPossible(root->left, l, root->val) and
                          isPossible(root->right, root->val, r);
    else return false;
}

public:
    bool isValidBST(TreeNode* root) {
        long long int min = -1000000000000, max = 1000000000000;
        return isPossible(root, min, max);
    }
};
```

Output:

Problem 3

Given the root of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

Code:

```
class Solution {
public:
    bool isMirror(TreeNode* left, TreeNode* right) {
    if (!left && !right) return true;
    if (!left || !right) return false;
    return (left->val == right->val) && isMirror(left->left, right->right) && isMirror(left->right, right->left);
}

bool isSymmetric(TreeNode* root) {
    if (!root) return true;
    return isMirror(root->left, root->right);
}
};
```

Output:

Problem 4

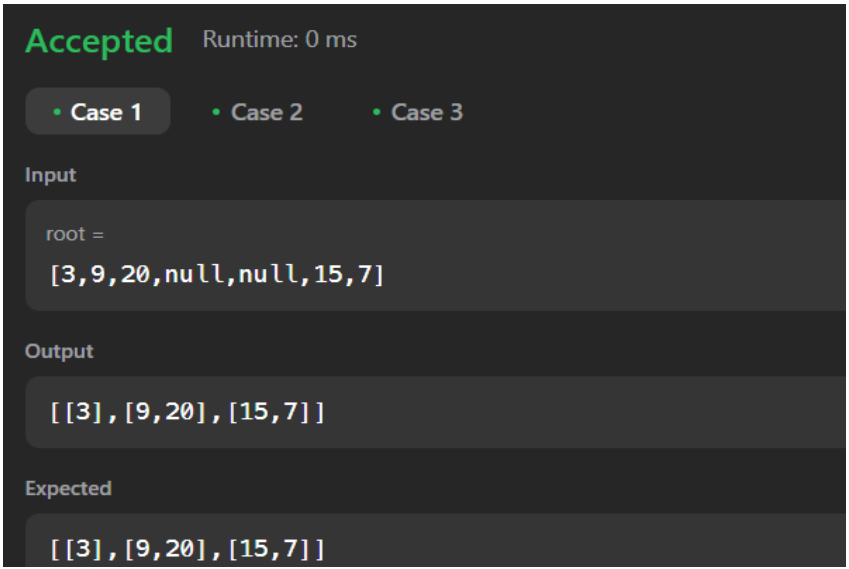
Given the root of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

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

Given an integer array nums where the elements are sorted in **ascending order**, convert *it to a* ***height-balanced*** *binary search tree*.

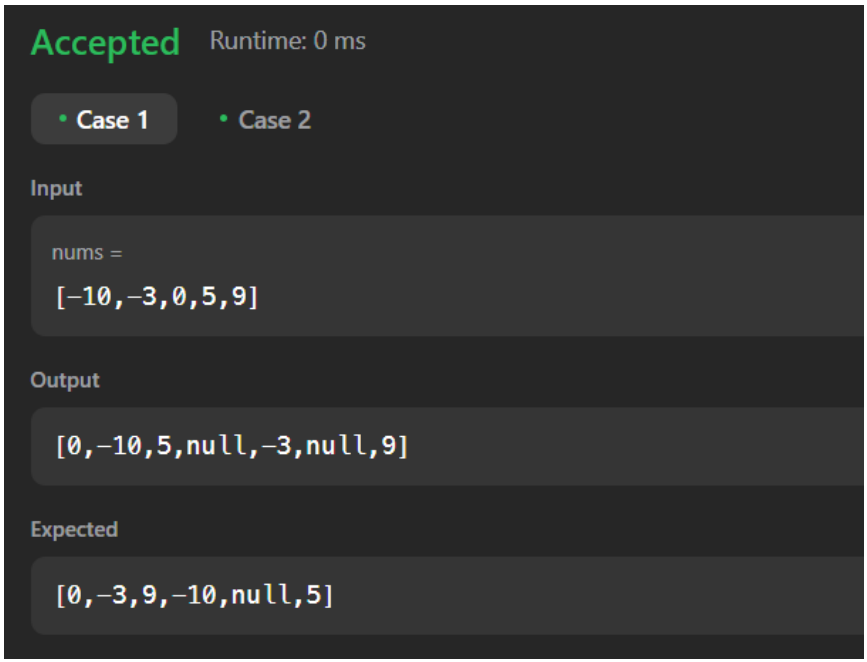
Code:

```
#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:



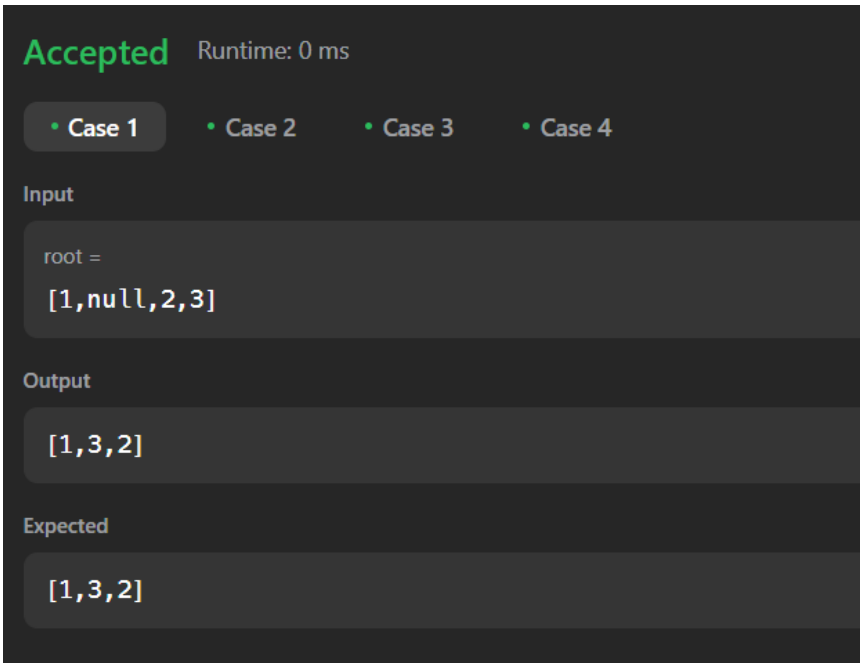
Problem 6

Given the root of a binary tree, return *the inorder traversal of its nodes' values*.

Code:

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
    vector<int> ans;
    if (root == NULL) return ans;
    vector<int> left = inorderTraversal(root->left);
    ans.insert(ans.end(), left.begin(), left.end());
    ans.push_back(root->val);
    vector<int> right = inorderTraversal(root->right);
    ans.insert(ans.end(), right.begin(), right.end());
    return ans;
}
};
```

Output:



Problem 7

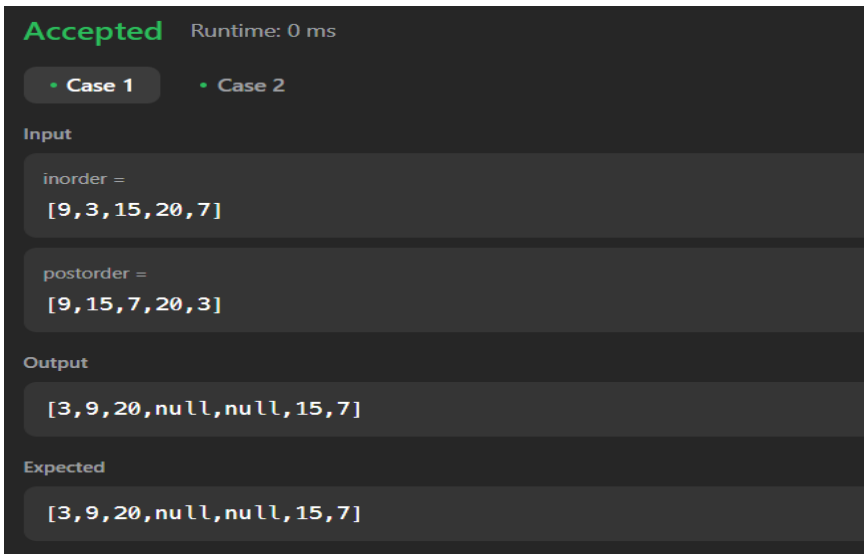
Given two integer arrays inorder and postorder where inorder is the inorder traversal of a binary tree and postorder is the postorder traversal of the same tree, construct and return *the binary tree*.

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 8

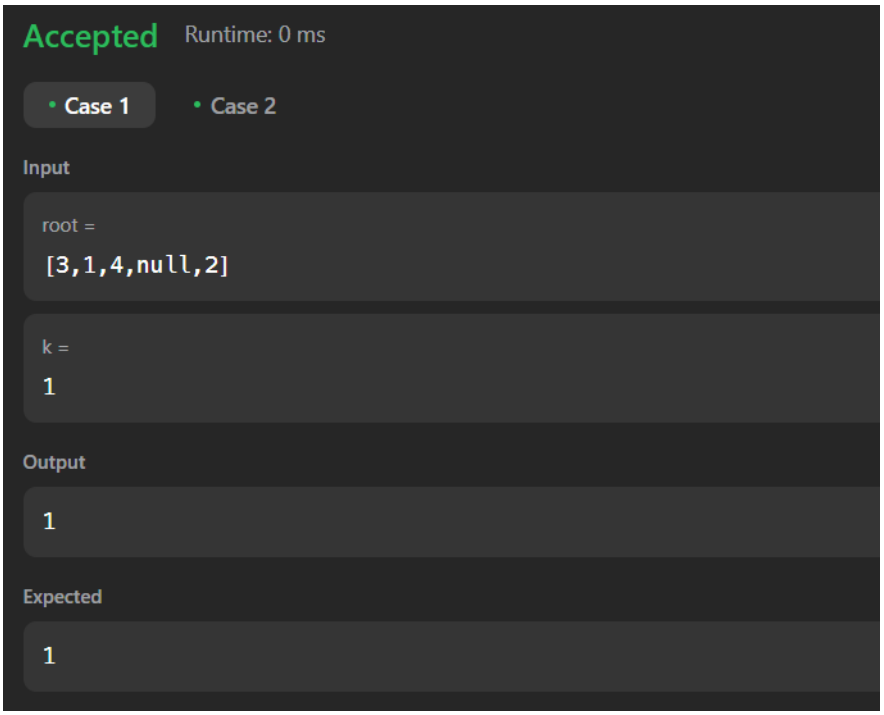
Given the root of a binary search tree, and an integer k, return *the* kth *smallest value (**1-indexed**) of all the values of the nodes in the tree*.

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:



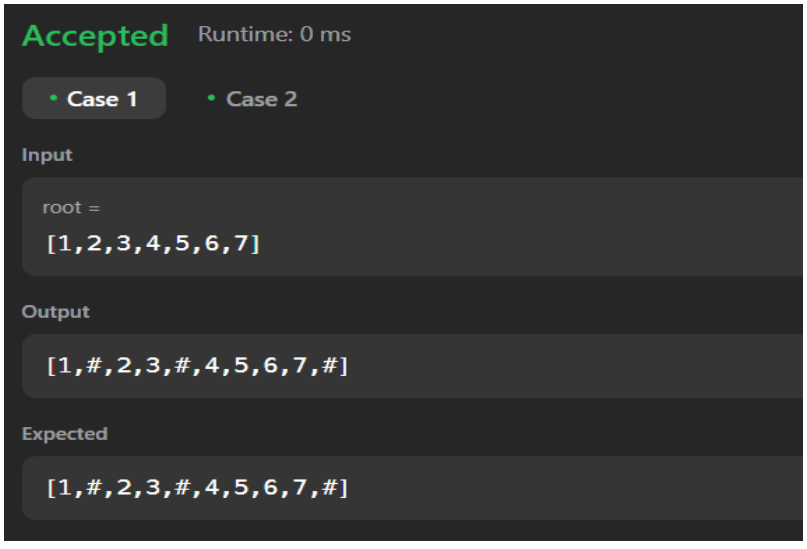
Problem 9

You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

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:



Problem 10

Given an n x n matrix where each of the rows and columns is sorted in ascending order, return *the* kth *smallest element in the matrix*.

Code:

```
class Solution {
public:
    int kthSmallest(vector<vector<int>>& matrix, int z) {
        int n = matrix.size(), m = matrix[0].size();
        int a[n*m], k=0;
        for(int i=0; i<n; i++){
            for(int j=0; j<m; j++){
                a[k] = matrix[i][j];
                k++;
            }

        }
        sort(a, a+(n*m));
        return a[z-1];
    }
};
```

Output: