Experiment 5

Student Name: PRATEEK **UID: 22BCS13864**

Branch: UIE CSE 3rd Year **Section/Group: 22BCS_KPIT-901-'B'**

Semester: 6th **Date of Performance: 18th Feb 2025**

Subject Name: Advanced Programming – II **Subject Code: 22CSP-351**

1. Aim:

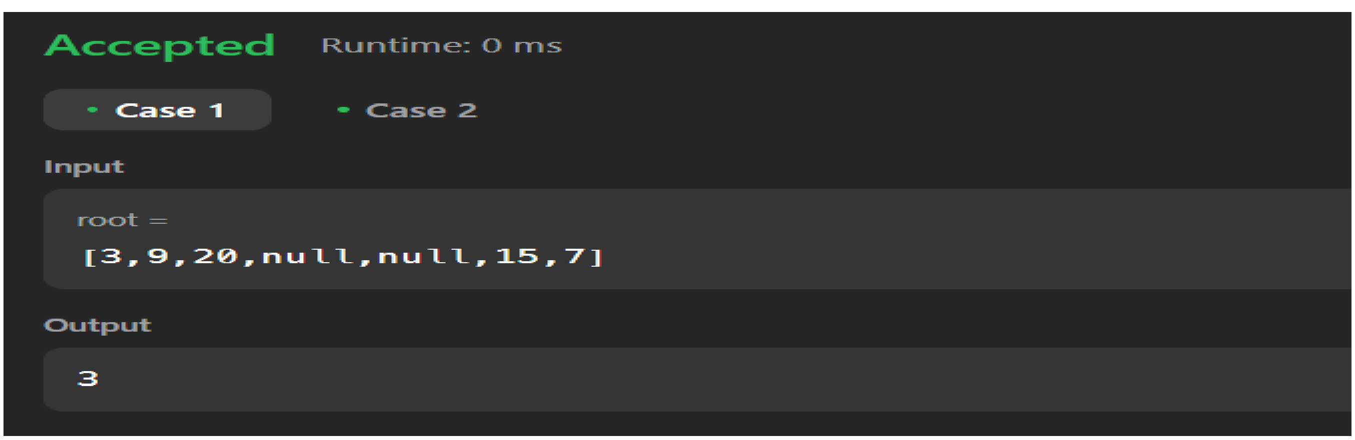
Given the root of a binary tree, return its maximum depth.

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. Implementation/Code:

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

3. Output:



QUES:2

1. **Aim:**

   Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

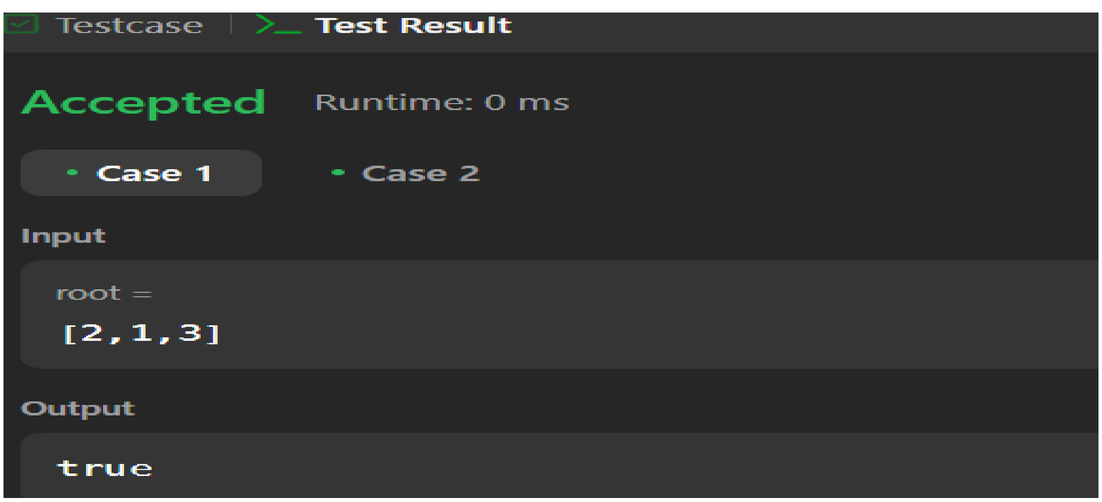
   A **valid BST** is defined as follows:

- 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.

2. **Implementation/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);
    }
};
```

3. **Output:**



QUESTION:3 Symmetric Tree

```
class Solution {
public:
    bool solve(TreeNode *p , TreeNode*q){
        if(p==NULL && q==NULL){
            return true;
        }
        if(p==NULL || q==NULL){
            return false;
        }
        if(p->val != q->val){
            return false;
        }
        if(solve(p->left,q->right) && solve(p->right,q->left)){
            return true;
        }
        return false;
    }
    bool isSymmetric(TreeNode* root) {
        if(root==NULL)return true;
        return solve(root,root);
    }
};
```

QUESTION:4 Binary Tree Level Order Traversal

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>>ans;
        if(root==NULL)return ans;
        queue<TreeNode*>q;
        q.push(root);
        while(!q.empty()){
            int s=q.size();
            vector<int>v;
            for(int i=0;i<s;i++){
                TreeNode *node=q.front();
                q.pop();
                if(node->left!=NULL)q.push(node->left);
```

```
                if(node->right!=NULL)q.push(node->right);
                v.push_back(node->val);
            }
            ans.push_back(v);
        }
        return ans;
    }
};
```

QUESTION:5 Convert Sorted Array to Binary Search Tree

```
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;
    }
};
```

QUESTION:6 Binary Tree Inorder Traversal

```
class Solution {
public:
    void inOrder(TreeNode* root ,vector<int>&ans){
        if(root!=nullptr){
            inOrder(root->left,ans);
            ans.push_back(root->val);
            inOrder(root->right,ans);
        }
    }
    vector<int> inorderTraversal(TreeNode* root) {
```

```
        vector<int>ans;
        inOrder(root,ans);
        return ans;
    }
};
```

QUESTION:7 Construct Binary Tree from Inorder and Postorder Traversal

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

    TreeNode* constructTree(vector<int>& inorder, vector<int>& postorder,
unordered_map<int, int>& inorderIndexMap, int& postIndex, int inStart, int inEnd)
{
        if (inStart > inEnd) return nullptr;

        int rootVal = postorder[postIndex--];
        TreeNode* root = new TreeNode(rootVal);
        int rootIndex = inorderIndexMap[rootVal];

        root->right = constructTree(inorder, postorder, inorderIndexMap,
postIndex, rootIndex + 1, inEnd);
        root->left = constructTree(inorder, postorder, inorderIndexMap, postIndex,
inStart, rootIndex - 1);

        return root;
    }
};
```

QUESTION:8 Kth Smallest Element in a BST

```
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];
    }
};
```

QUESTION:9 Populating Next Right Pointers in Each Node

```
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return nullptr;
        Node* leftMost = root;

        while (leftMost->left) {
            Node* currNode = leftMost;

            while (currNode) {
                currNode->left->next = currNode->right;
                if (currNode->next) {
                    currNode->right->next = currNode->next->left;
                }
                currNode = currNode->next;
            }

            leftMost = leftMost->left;
        }
```

```
        return root;
    }
};
```

QUESTION 10: Kth Smallest Element in a Sorted Matrix

```
class Solution {
public:
    int kthSmallest(vector<vector<int>>& matrix, int k) {
        priority_queue<int> maxHeap;

        for (const auto& row : matrix) {
            for (int num : row) {
                maxHeap.push(num);
                if (maxHeap.size() > k) {
                    maxHeap.pop();
                }
            }
        }

        return maxHeap.top();
    }
};
```