Experiment 6

Student Name: Aditya Patel

UID: 22BCS11543

Branch: BE-CSE

Section/Group: 22BCS-IOT-640(B)

Semester: 6th

Date of Performance: 10/03/25

Subject Name: ADVANCED PROGRAMMING LAB - 2

Subject Code: 22CSP-351

PROGRAM-1

1) **Aim:** Maximum Depth of Binary Tree.

2) **Objective:** The objective of this code is to determine the maximum depth (or height) of a binary tree. The depth is defined as the number of nodes along the longest path from the root node to a leaf node. The code uses a recursive approach to traverse the tree and compute the depth.

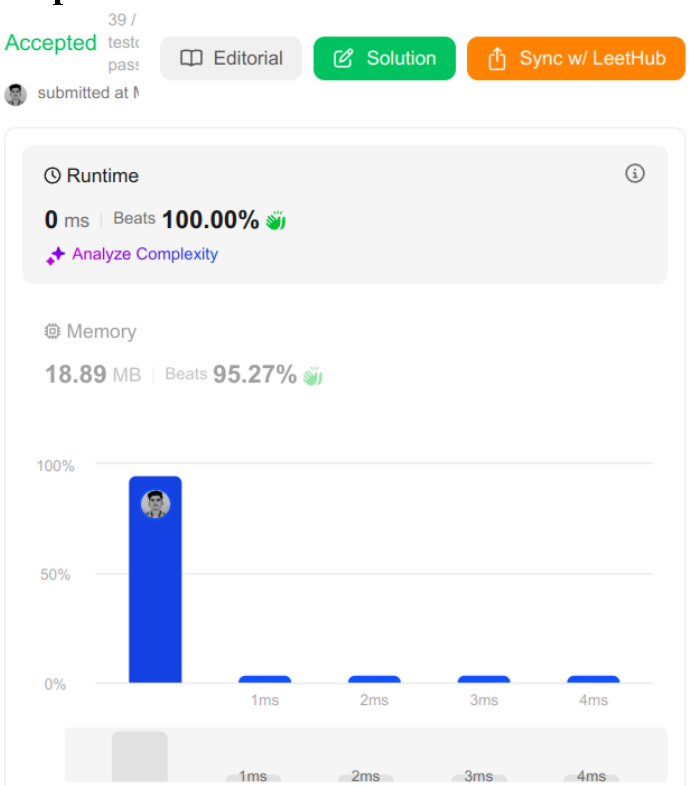
3) **Implementation/Code:**

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

// struct 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) return 0;
        int left_depth = maxDepth(root->left);
        int right_depth = maxDepth(root->right);
        return max(left_depth, right_depth) + 1;
    }
};
```

4) Output:



5) Learning Outcomes:

- Understanding Recursion in Tree Traversals.
- Binary Tree Depth Calculation.
- Use of Base Case in Recursion.
- Comparison of Left and Right Subtrees.
- Implementation of a Recursive Function in C++.

PROGRAM-2

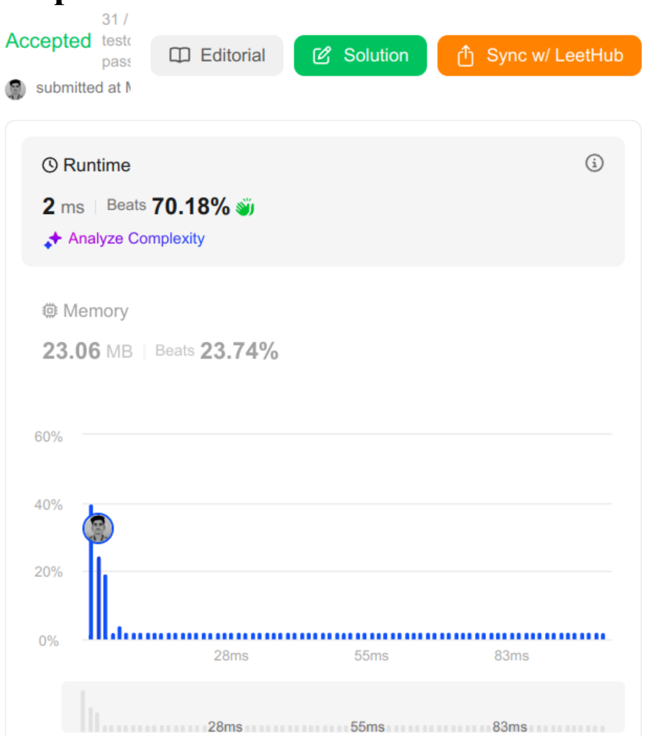
1) Aim: Convert Sorted Array to Binary Search Tree.

2) Objective: The objective of this code is to convert a sorted array into a height-balanced Binary Search Tree (BST). A height-balanced BST is a binary tree where the depth of the left and right subtrees of every node differs by at most one. The algorithm uses a recursive approach to construct the BST by selecting the middle element of the array as the root, ensuring balanced tree formation.

3) Implementation/Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return buildBST(nums, 0, nums.size() - 1);
    }
private:
    TreeNode* buildBST(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 = buildBST(nums, left, mid - 1);
        root->right = buildBST(nums, mid + 1, right);
        return root;
    }
};
```

4) Output:



5) Learning Outcomes:

- Understanding Binary Search Tree (BST) Construction.
- Recursive Divide-and-Conquer Approach.
- Optimal Root Selection for Balanced Trees.
- Efficient Tree Construction Using O(n) Time Complexity.
- Practical Implementation of TreeNode Class in C++.

PROGRAM-3

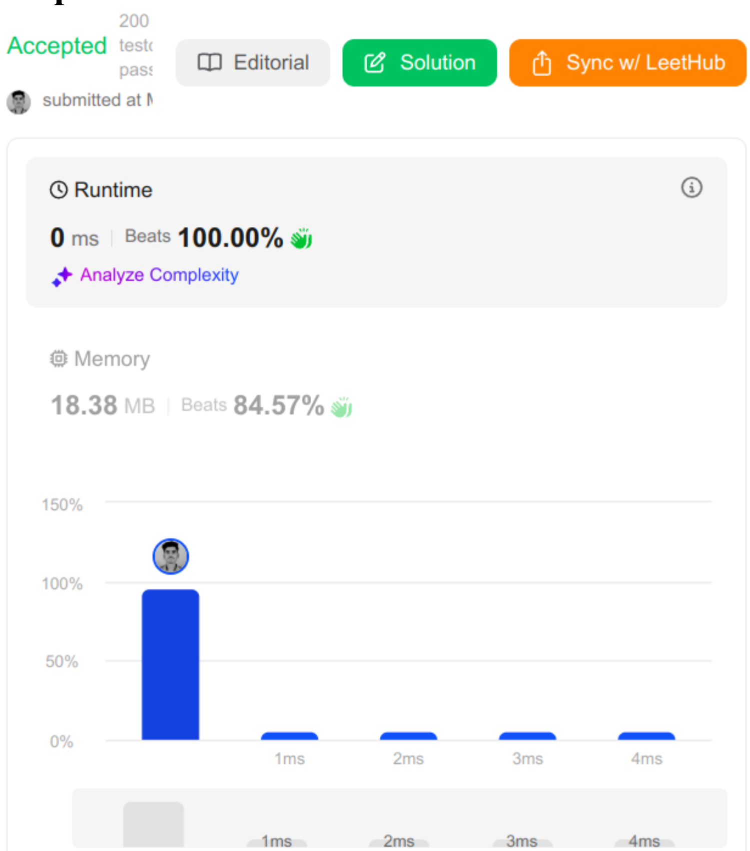
1) Aim: Symmetric Tree.

2) Objective: The objective of this code is to determine whether a given binary tree is symmetric. A binary tree is symmetric if it is a mirror image of itself around its center. The solution uses a recursive approach to check if the left and right subtrees of the root node are mirror images of each other.

3) Implementation/Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    bool isMirror(TreeNode* t1, TreeNode* t2) {
        if (!t1 && !t2) return true;
        if (!t1 || !t2) return false;
        return (t1->val == t2->val) &&
               isMirror(t1->left, t2->right) &&
               isMirror(t1->right, t2->left);
    }
    bool isSymmetric(TreeNode* root) {
        return isMirror(root, root);
    }
};
```

4) Output:



5) Learning Outcomes:

- Understanding Tree Symmetry.
- Recursive Tree Traversal.
- Concept of Mirror Trees.
- Handling Base Cases in Recursion.
- Efficient Checking Using O(n) Time Complexity.