



## Experiment-6

**Student Name:** Satyakam Tyagi

**Branch:** BE-CSE

**Semester:**06

**UID:** 22BET10013

**Section/Group:**TPP\_IOT\_701-A

**Date of Performance:**27-02-2025

**Subject Name:** AP LAB-II

**Subject Code:** 22ITP-351

### 1. Aim:

- a. **Maximum Depth of Binary Tree.**
- b. **Binary Tree Level Order Traversal.**
- c. **Convert Sorted Array to Binary Search Tree.**

### 2. Introduction to Binary Trees Problem:

A Binary Search Tree (BST) is a special type of binary tree where:

- The left subtree contains values less than the node.
- The right subtree contains values greater than the node.

This structure enables efficient searching, insertion, and deletion in  $O(\log N)$  time on average, making BSTs essential in algorithms, databases, and real-world applications.

### 3. Implementation/Code:

#### A. Maximum Depth of Binary Tree

```
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;  
  
    }  
};
```



## **B. Binary Tree Level Order Traversal**

```
class Solution { public:    vector<vector<int>>
levelOrder(TreeNode* root) {
vector<vector<int>> result;
    if (!root) return result; // If tree is empty, return empty
vector

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

    while (!q.empty()) {
        int size = q.size(); // Number of nodes at current level
vector<int> level;

        for (int i = 0; i < 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);
        }

        result.push_back(level);
    }

    return result;
}
};
```

### C. Convert Sorted Array to Binary Search Tree

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

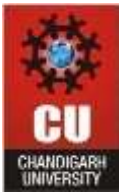
private:
    TreeNode* buildBST(vector<int>& nums, int left, int right) {
if (left > right) return nullptr; // Base case

        int mid = left + (right - left) / 2; // Find middle element
        TreeNode* root = new TreeNode(nums[mid]); // Create root node

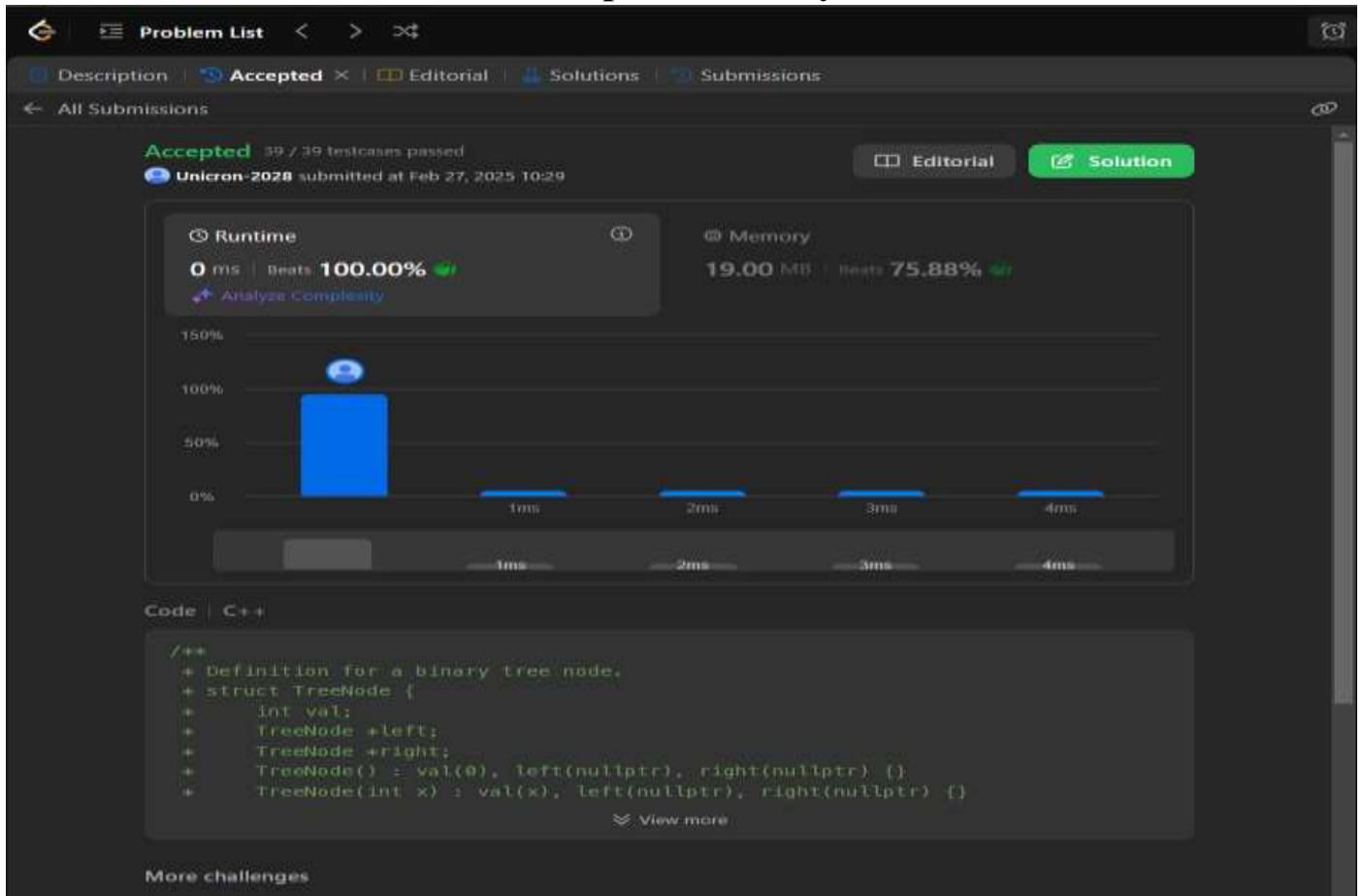
        root->left = buildBST(nums, left, mid - 1); // Recursively build left
subtree
        root->right = buildBST(nums, mid + 1, right); // Recursively build right
subtree

        return root;
    } };
```

### 4. Output



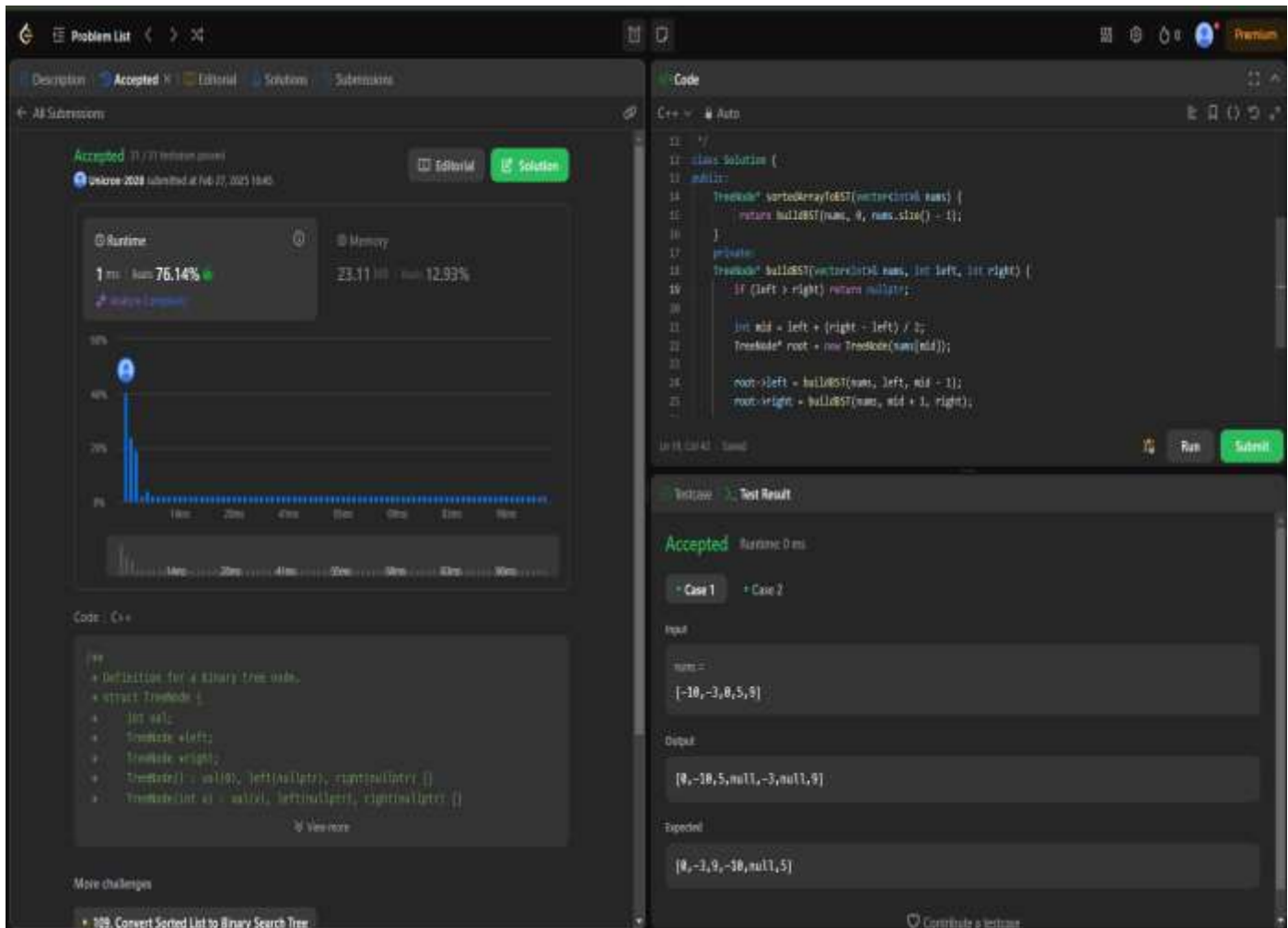
## A. Maximum Depth of Binary Tree



## B. Binary Tree Level Order Traversal



## C. Convert Sorted Array to Binary Search Tree



The screenshot displays a coding interface for the problem "Convert Sorted List to Binary Search Tree". The left panel shows the problem description, a graph of runtime and memory usage, and a code editor with a C++ solution. The right panel shows the test results, indicating the solution is accepted.

**Runtime and Memory Usage:**

- Runtime: 1 ms, 76.14%
- Memory: 23.11 MB, 12.93%

**Code (C++):**

```

1 //
2 // Definition for a binary tree node.
3 // struct TreeNode {
4 //     int val;
5 //     TreeNode *left;
6 //     TreeNode *right;
7 //     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8 //     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
9 // };
10
11 // Definition for a sorted list node.
12 // struct ListNode {
13 //     int val;
14 //     ListNode *next;
15 //     ListNode() : val(0), next(nullptr) {}
16 //     ListNode(int x) : val(x), next(nullptr) {}
17 // };
18
19 // Solution
20 // class Solution {
21 // public:
22 //     TreeNode* sortedArrayToBST(vector<int> &nums) {
23 //         return buildBST(nums, 0, nums.size() - 1);
24 //     }
25 // private:
26 //     TreeNode* buildBST(vector<int> &nums, int left, int right) {
27 //         if (left > right) return nullptr;
28 //
29 //         int mid = left + (right - left) / 2;
30 //         TreeNode* root = new TreeNode(nums[mid]);
31 //
32 //         root->left = buildBST(nums, left, mid - 1);
33 //         root->right = buildBST(nums, mid + 1, right);
34 //     }
35 // };

```

**Test Result:**

- Accepted
- Runtime: 0 ms
- Case 1: Input: [-10, -3, 0, 5, 9], Output: [0, -10, 5, null, -3, null, 9], Expected: [0, -10, 5, null, -3, null, 9]

## 5. Learning Outcomes:

### Understanding Binary Search Trees (BSTs)

- Learn how BSTs maintain the ordering property (left < root < right).
- Understand the importance of height-balancing in BSTs for optimal performance.

### Divide and Conquer Approach

- Learn how recursion can be used to break a problem into smaller subproblems. □ Understand how selecting the middle element ensures balanced tree formation.

### Recursive Tree Construction

- Develop the ability to construct trees dynamically using recursive functions.
- Learn how to create and link `TreeNode` objects in C++.



# DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

## Time & Space Complexity Analysis

- Understand why  $O(N)$  time complexity is achieved (each element processed once). □ Learn about  $O(\log N)$  space complexity due to recursion depth.

## Real-World Applications

- BSTs are used in databases, search operations, and hierarchical data storage.
- Understanding balanced trees helps in optimizing search, insert, and delete operations.