Experiment-6

Student Name: Garv Kumar **UID:** 22BET10103
Branch: BE-IT **Section/Group:** 22BET_IOT-702/A
Semester: 6th **Date of Performance:** 07/03/2025
Subject Name: Advanced Programming Lab-2 **Subject Code:** 22ITP-351

Problem-1

1. **Aim:**

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

2. **Objective:**
   - Develop an algorithm to determine the maximum depth of a given binary tree.
   - Implement a function that traverses the tree and returns the longest path from the root to a leaf node.

3. **Implementation:**

```
class Solution {

public:

   int maxDepth(TreeNode* root) {

      if (!root) return 0;

      return 1 + max(maxDepth(root->left), maxDepth(root->right));

   }

};
```

4. Output:

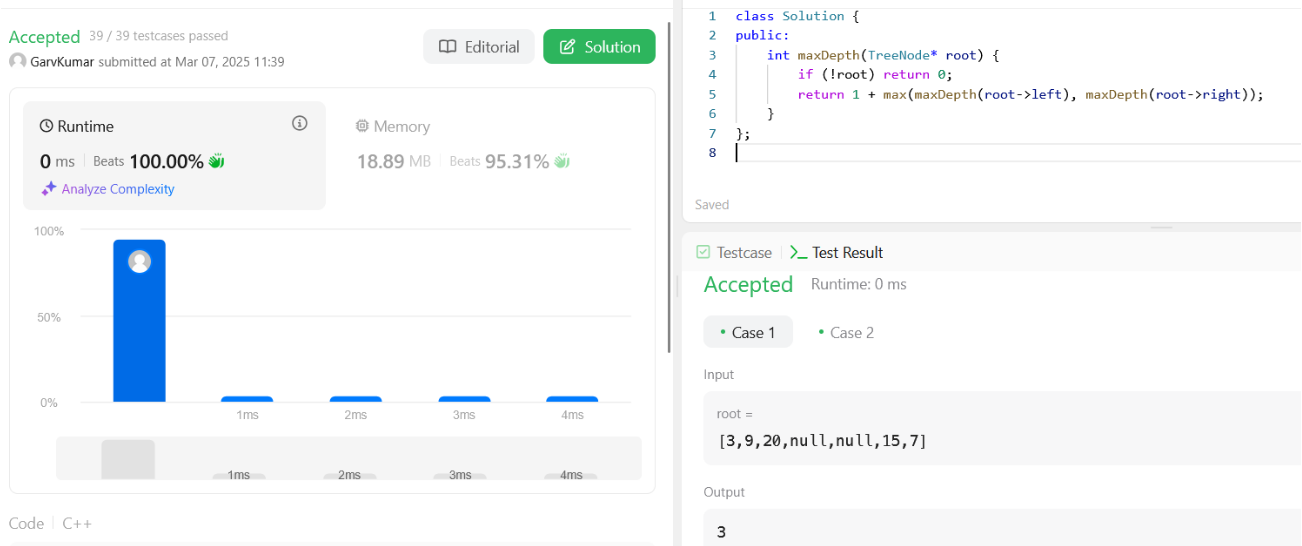

Fig: Maximum Depth of Binary Tree.

Problem-2

1. Aim:

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

2. Objective:

1 Develop an algorithm to check whether a given binary tree satisfies the properties of a Binary Search Tree (BST).

2 Implement a function that verifies if all nodes follow the BST rules: left subtree nodes are smaller, right subtree nodes are larger, and there are no violations in the entire tree.

3. Implementation:

```
class Solution {

public:

  bool isValidBST(TreeNode* root, long minVal = LONG_MIN, long maxVal = LONG_MAX) {

     if (!root) return true;
```

```
        if (root->val <= minVal || root->val >= maxVal) return false;

        return isValidBST(root->left, minVal, root->val) && isValidBST(root->right, root->val, maxVal);

    }

};
```

4. Output:

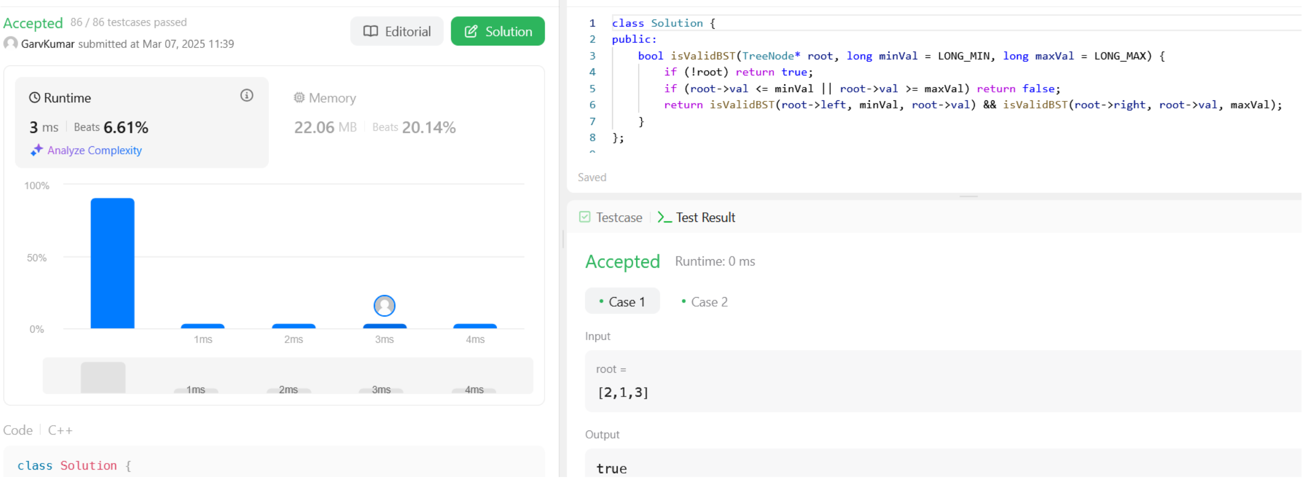

Fig: Validate Binary Search Tree.

Problem-3

1. Aim:

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

2. Objective:

1. Develop an algorithm to determine if a given binary tree is symmetric around its center.
2. Implement a function that checks whether the left and right subtrees are mirror images of each other.

3. Implementation:

```
class Solution {
```

```
public:
    bool isMirror(TreeNode* t1, TreeNode* t2) {
        if (!t1 || !t2) return t1 == t2;
        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:

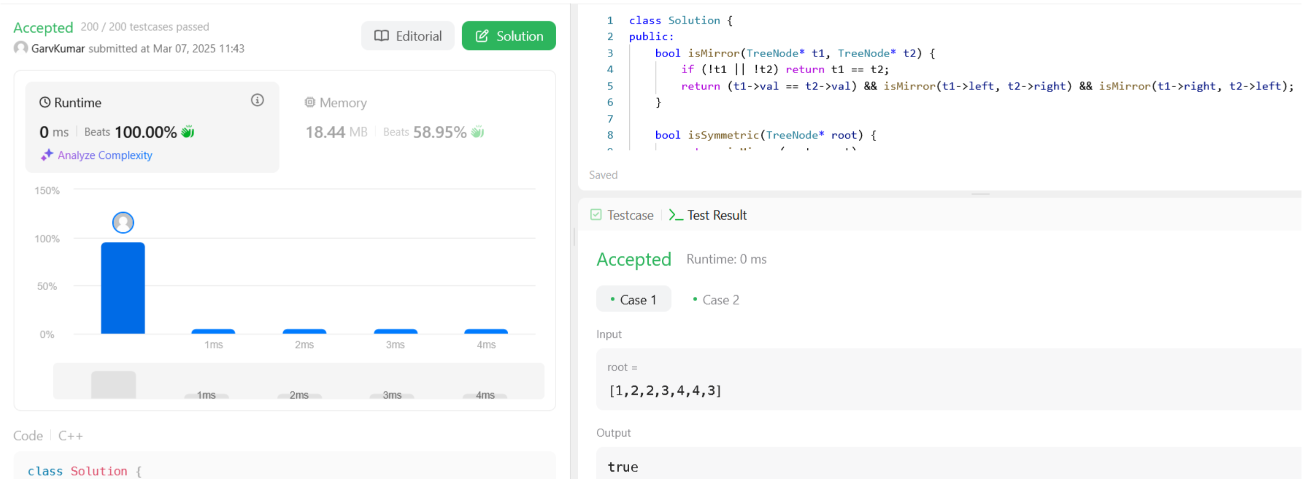

Fig: Symmetric Tree.

Problem-4

1. Aim:

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).

2. Objective:

1 Develop an algorithm to perform level order traversal of a given binary tree.

2 Implement a function that processes nodes level by level from left to right and returns their values.

3. Implementation:

```
class Solution {
public:
   vector<vector<int>> levelOrder(TreeNode* root) {
      vector<vector<int>> res;
      if (!root) return res;
      queue<TreeNode*> q;
      q.push(root);
      while (!q.empty()) {
         int size = q.size();
         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);
         }
         res.push_back(level);
      }
      return res;
   }
 };
```

4. Output:

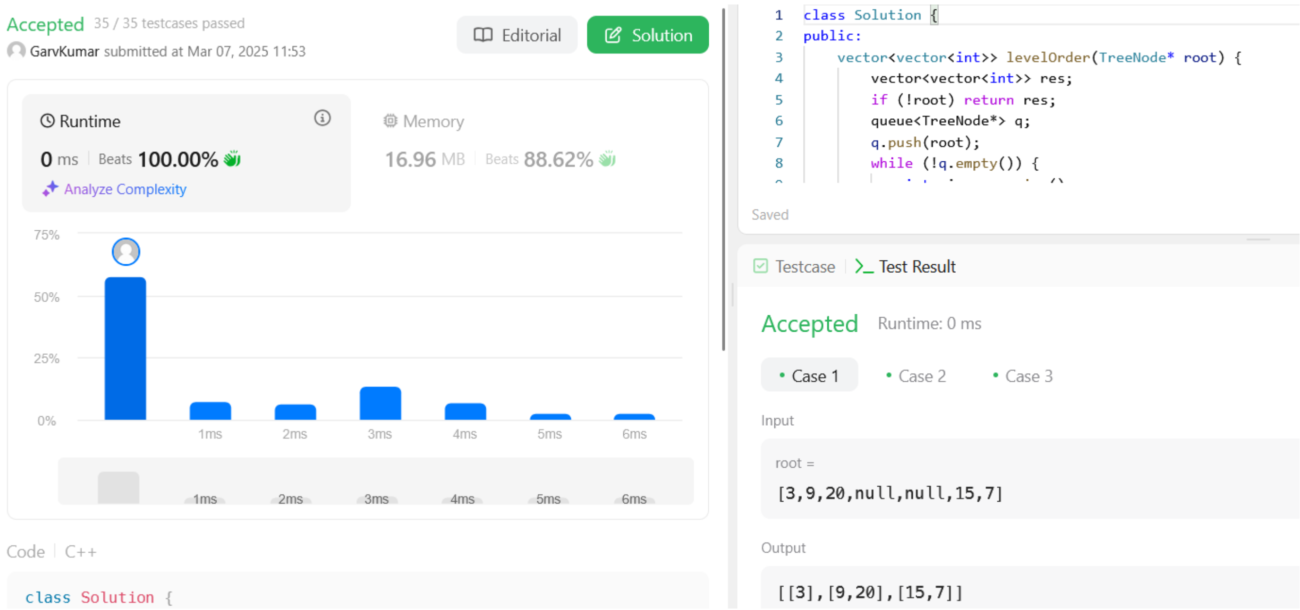

Fig: Binary Tree Level Order Traversal.

Problem-5

1. Aim:

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

2. Objective:

1 Develop an algorithm to convert a sorted integer array into a height-balanced Binary Search Tree (BST).

2 Implement a function that recursively selects the middle element as the root to ensure balanced tree construction.

3. Implementation:

```
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:

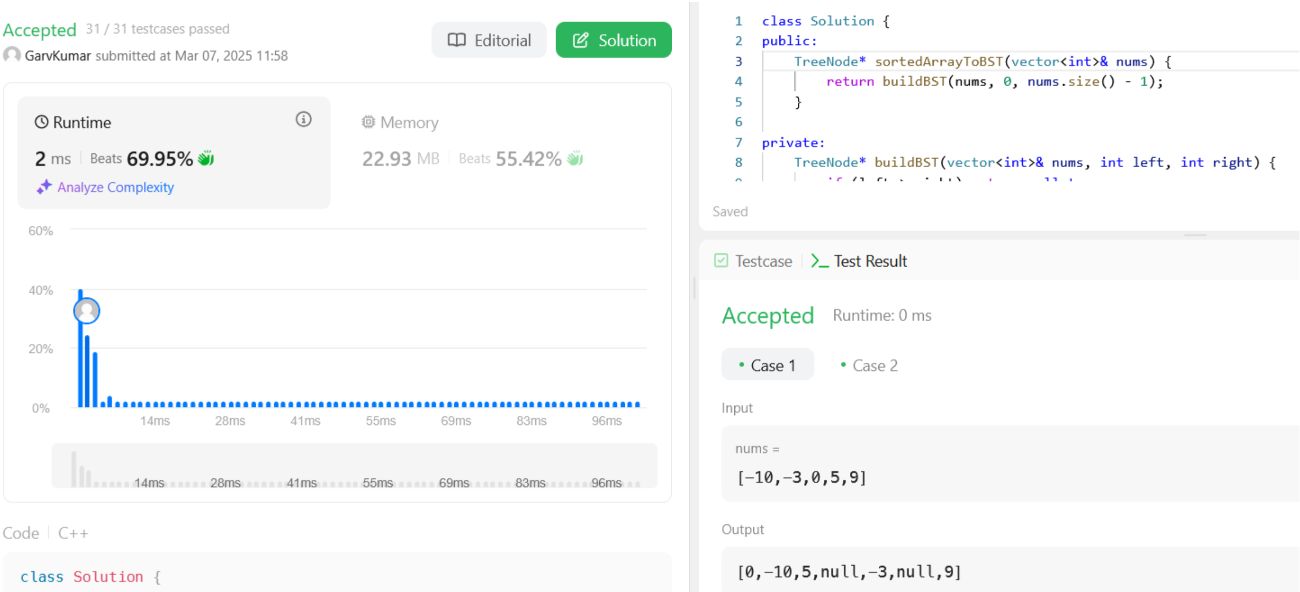

Fig: Convert Sorted Array to Binary Search Tree.

Problem-6

1. Aim:

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

2. Objective:

1 Develop an algorithm to perform inorder traversal of a given binary tree.

2 Implement a function that visits nodes in left-root-right order and returns their values.

3. Implementation:

```
class Solution {

public:

    vector<int> inorderTraversal(TreeNode* root) {

        vector<int> res;

        stack<TreeNode*> s;

        while (root || !s.empty()) {

            while (root) {

                s.push(root);

                root = root->left;

            }

            root = s.top();

            s.pop();

            res.push_back(root->val);

            root = root->right;

        }

        return res;

    }
```

};

4. Output:

Fig: Binary Tree Inorder Traversal.

Problem-7

1. Aim:

Given the root of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

2. Objective:

1 Develop an algorithm to perform zigzag level order traversal of a given binary tree.

2 Implement a function that alternates traversal direction at each level, switching between left-to-right and right-to-left.

3. Implementation:

class Solution {

public:

```
vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
    vector<vector<int>> res;
    if (!root) return res;
    queue<TreeNode*> q;
    q.push(root);
    bool leftToRight = true;
    while (!q.empty()) {
        int size = q.size();
        vector<int> level(size);
        for (int i = 0; i < size; i++) {
            TreeNode* node = q.front();
            q.pop();
            int index = leftToRight ? i : size - 1 - i;
            level[index] = node->val;
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
        res.push_back(level);
        leftToRight = !leftToRight;
```

}

return res;

}

};

4. Output:

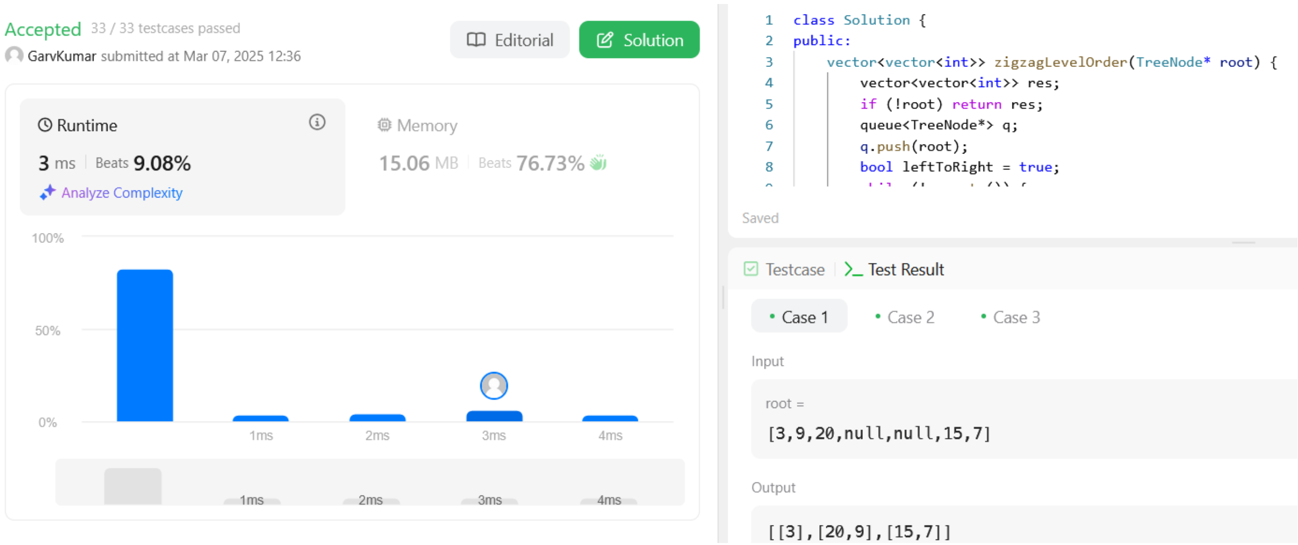

Fig: Binary Tree Zigzag Level Order Traversal.

Problem-8

5. Aim:

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.

6. Objective:

3 Develop an algorithm to reconstruct a binary tree from its inorder and postorder traversal sequences.

4 Implement a function that identifies the root from postorder, splits the inorder list into subtrees, and recursively builds the tree.

7. Implementation:

```
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        unordered_map<int, int> inMap;
        for (int i = 0; i < inorder.size(); i++)
            inMap[inorder[i]] = i;
        int postIdx = postorder.size() - 1;
        return build(inorder, postorder, inMap, postIdx, 0, inorder.size() - 1);
    }
private:
    TreeNode* build(vector<int>& inorder, vector<int>& postorder, unordered_map<int,
     int>& inMap, int& postIdx, int inLeft, int inRight) {
        if (inLeft > inRight) return nullptr;
        int rootVal = postorder[postIdx--];
        TreeNode* root = new TreeNode(rootVal);
        int inIdx = inMap[rootVal];
        root->right = build(inorder, postorder, inMap, postIdx, inIdx + 1, inRight);
        root->left = build(inorder, postorder, inMap, postIdx, inLeft, inIdx - 1);
        return root;
    }
 };
```

8. Output:

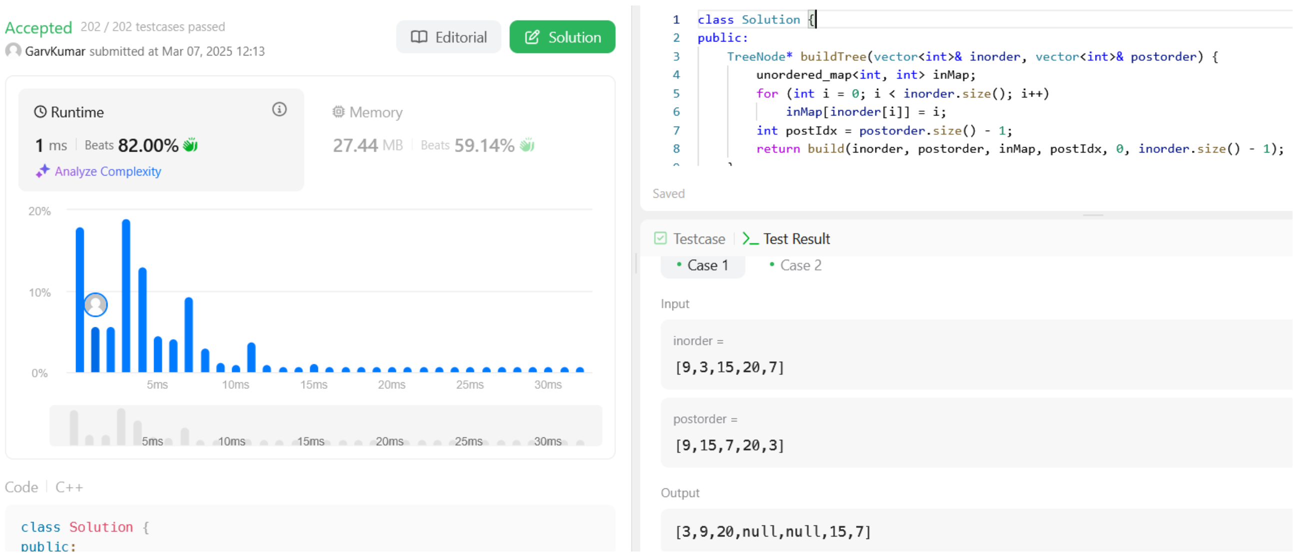

Fig: Construct Binary Tree from Inorder and Postorder Traversal.

Problem-9

1. Aim:

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

2. Objective:

1 Develop an algorithm to find the kth smallest element in a given Binary Search Tree (BST).

2 Implement an inorder traversal to retrieve elements in sorted order and return the kth smallest value.

3. Implementation:

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        stack<TreeNode*> s;
        while (true) {
            while (root) {


                s.push(root);
                root = root->left;
            }
            root = s.top();
            s.pop();
            if (--k == 0) return root->val;
            root = root->right;
        }
    }
};
```

4. Output:

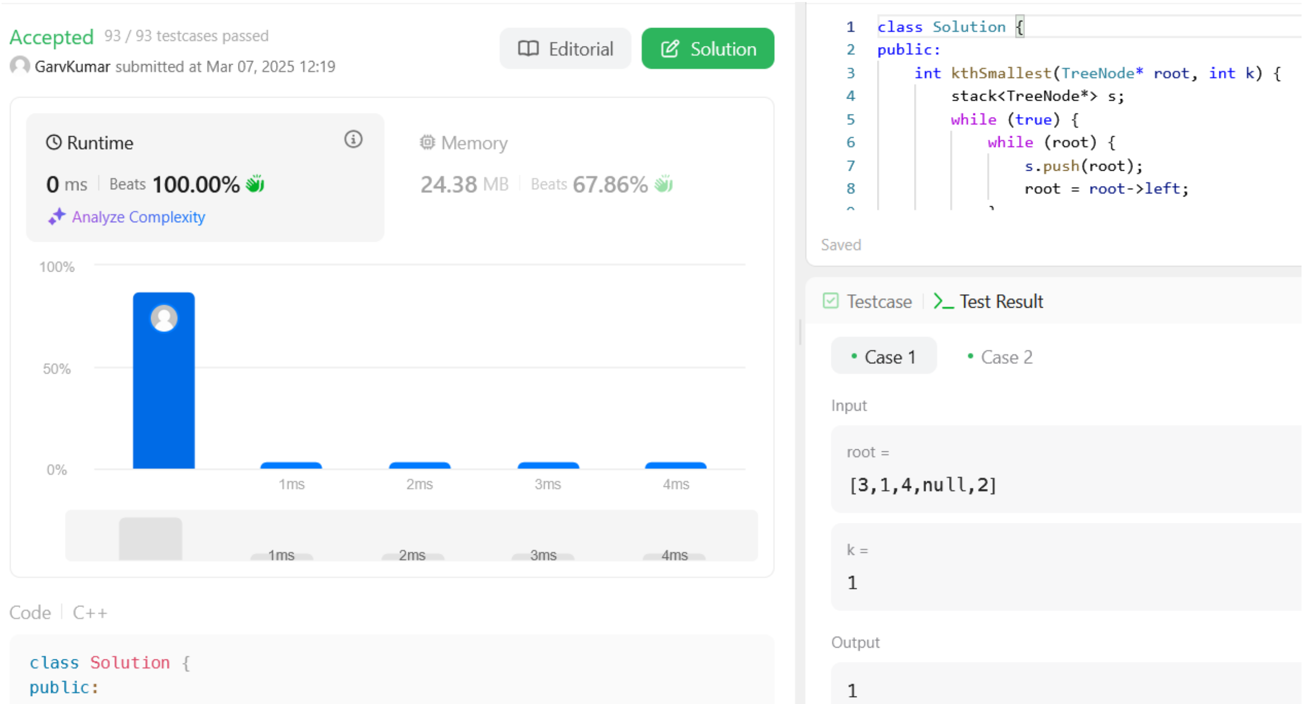

Fig: Kth Smallest Element in a BST.

Problem-10

1. Aim:

Develop an algorithm to connect the next pointers of each node in a perfect binary tree, ensuring every node points to its adjacent right node or NULL if no right neighbor exists.

2. Objective:

1 Design a method to traverse the perfect binary tree level by level and link each node to its next right node.

2 Implement an efficient approach that updates the next pointers without using extra space beyond recursion.

3. Implementation:

class Solution {

```
public:
    Node* connect(Node* root) {
        if (!root) return nullptr;
        queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            int size = q.size();
            for (int i = 0; i < size; i++) {
                Node* node = q.front();
                q.pop();
                if (i < size - 1) node->next = q.front();
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }
        }
        return root;
    }
};
```

4. Output:

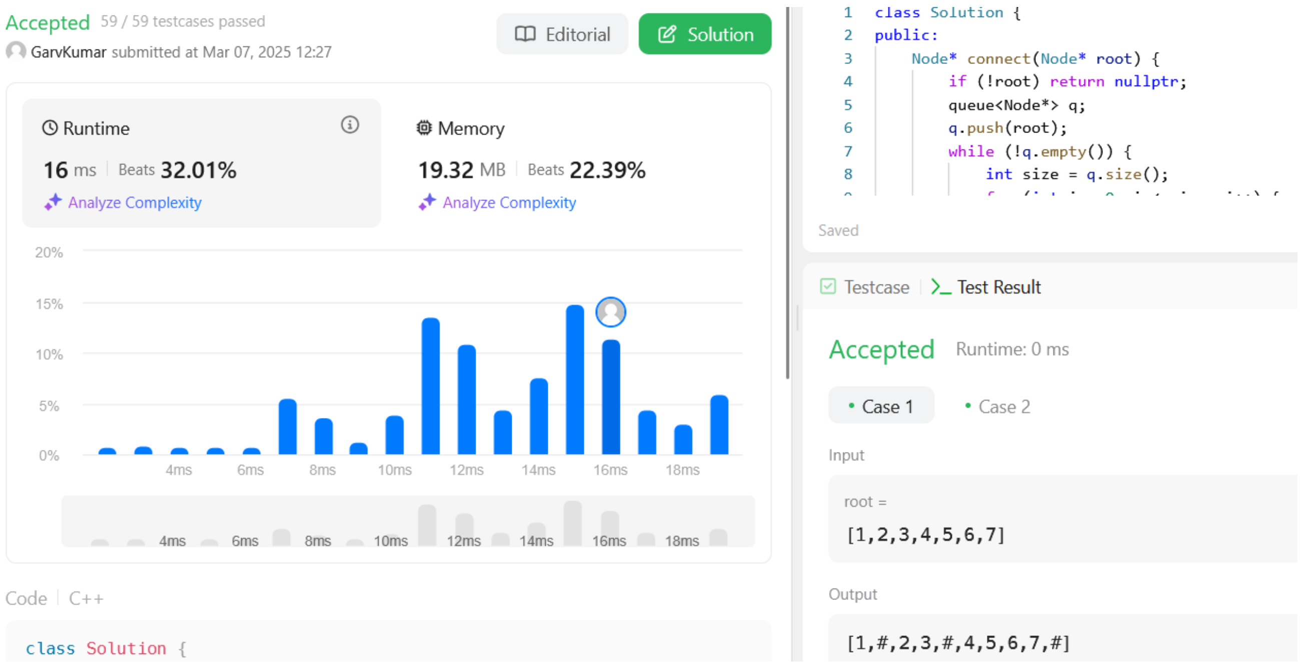

Fig: Populating Next Right Pointers in Each Node.

5. Learning Outcomes:

1 **Maximum Depth of a Binary Tree:**

1. Understand how to traverse a binary tree recursively to determine its depth.
2. Apply depth-first search (DFS) or breadth-first search (BFS) to compute the longest path from root to leaf.

2 **Validate a Binary Search Tree (BST):**

1. Learn how to check BST properties using inorder traversal.
2. Gain the ability to use recursion or iterative methods to validate node relationships.

3 **Check if a Binary Tree is Symmetric:**

1. Understand how to compare left and right subtrees for symmetry.
2. Apply recursion or iterative queue-based methods to check mirror properties.

4 **Level Order Traversal of a Binary Tree:**

1. Learn how to traverse a binary tree level by level using a queue.
2. Understand how to efficiently process nodes in a breadth-first manner.

5 **Convert Sorted Array to a Height-Balanced BST:**

1. Learn how to construct a balanced BST by selecting the middle element as the root.
2. Understand how to recursively build left and right subtrees while maintaining balance.

6 **Inorder Traversal of a Binary Tree:**

1. Understand how to perform left-root-right traversal of a binary tree.
2. Gain the ability to implement both recursive and iterative inorder traversal.

7 **Zigzag Level Order Traversal of a Binary Tree:**

1. Learn how to modify level order traversal to alternate between left-to-right and right-to-left.
2. Understand how to use a queue or deque to manage traversal direction dynamically.

8 **Construct Binary Tree from Inorder and Postorder Traversal:**

1. Learn how to reconstruct a binary tree by identifying roots and splitting traversal sequences.
2. Understand the recursive process of building subtrees based on given traversals.

9 **Find the kth Smallest Element in a BST:**

1. Learn how inorder traversal retrieves elements in sorted order.

2. Understand how to efficiently find the kth smallest element using recursion or iteration.

10 **Populate Next Right Pointers in a Perfect Binary Tree:**

1. Learn how to traverse a perfect binary tree level by level to set next pointers.
2. Understand space-efficient solutions that utilize existing pointers instead of extra storage.