Experiment 6

Student Name: Vipul kumar **UID: 22BCS10023**
Branch: CSE **Section: IOT-637-B**
Semester: 6th **Date of Performance: 28-02-25**
Subject: AP LAB-II **Subject Code:22CSP-351**

1. Aim:

Problem 6.1: Maximum Depth of Binary Tree

- **Problem Statement:** 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).

Problem 6.2: Convert Sorted Array to Binary Search Tree

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

Problem 6.3: Binary Tree In-order Traversal

- **Problem Statement**: Given the root of a binary tree, return the in-order traversal of its nodes' values.

Problem 6.4: Kth Smallest element in a BST

- **Problem Statement:** 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. Algorithm:

1. Insertion

1. Start at the root node.
2. If the tree is empty, create a new node and set it as the root.
3. If the tree is not empty:
   - Compare the value to be inserted with the current node's value.
   - If the value is less than the current node's value, move to the left child.
   - If the value is greater than the current node's value, move to the right child.
4. Repeat steps 3 until you find a null position (where the child is null).
5. Insert the new node at the null position.

2. Deletion

1. Start at the root node and search for the node to be deleted.
2. If the node is found:
   - If the node has no children (leaf node), simply remove it.
   - If the node has one child, replace the node with its child.
   - If the node has two children:
     - Find the in-order predecessor (maximum value in the left subtree) or in-order successor (minimum value in the right subtree).

- Replace the value of the node to be deleted with the predecessor or successor's value.
- Delete the predecessor or successor node (which will now have at most one child).

3. If the node is not found, return an error or null.

3. Searching

1. Start at the root node.
2. If the tree is empty, return null.
3. Compare the target value with the current node's value:
   - If they are equal, return the current node.
   - If the target value is less, move to the left child.
   - If the target value is greater, move to the right child.
4. Repeat step 3 until you find the target node or reach a null position.

4. In-order Traversal

1. Start at the root node.
2. Recursively traverse the left subtree.
3. Visit the current node (process its value).
4. Recursively traverse the right subtree.
5. Collect the values in a list or print them.

3. Code:

(A) Problem 6.1 :

```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
      List<List<Integer>> result = new ArrayList<>();
        if (root == null) {
            return result;
        }
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            List<Integer> currentLevel = new ArrayList<>();
            for (int i = 0; i < levelSize; i++) {
                TreeNode node = queue.poll();
                currentLevel.add(node.val);
                if (node.left != null) {
                    queue.offer(node.left);
                }
                if (node.right != null) {
```

```
                queue.offer(node.right);
            }
        }
        result.add(currentLevel);
    }
    return result;
  }
}
```

(B) problem 6.2 :

```
class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
    return buildTree(nums, 0, nums.length - 1);
    }
    private TreeNode buildTree(int[] nums, int left, int right) {
        if (left > right) {
            return null;
        }
        int mid = left + (right - left) / 2;
        TreeNode node = new TreeNode(nums[mid]);
        node.left = buildTree(nums, left, mid - 1);
        node.right = buildTree(nums, mid + 1, right);
        return node;
    }
}
```

(C) Problem 6.3:

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
        inorderHelper(root, result);
        return result;
    }
    private void inorderHelper(TreeNode node, List<Integer> result) {
        if (node == null) {
```

```
            return;
        }
        inorderHelper(node.left, result);
        result.add(node.val);
        inorderHelper(node.right, result);
    }
}
```

(C) Problem 6.4:

```
class Solution {
    private int count = 0;
    private int result = 0;

    public int kthSmallest(TreeNode root, int k) {
        inorderTraversal(root, k);
        return result;
    }

    private void inorderTraversal(TreeNode node, int k) {
        if (node == null) {
            return;
        }
        inorderTraversal(node.left, k);
        count++;
        if (count == k) {
            result = node.val;
            return; // Early exit
        }
        inorderTraversal(node.right, k);
    }
}
```

4. Output:

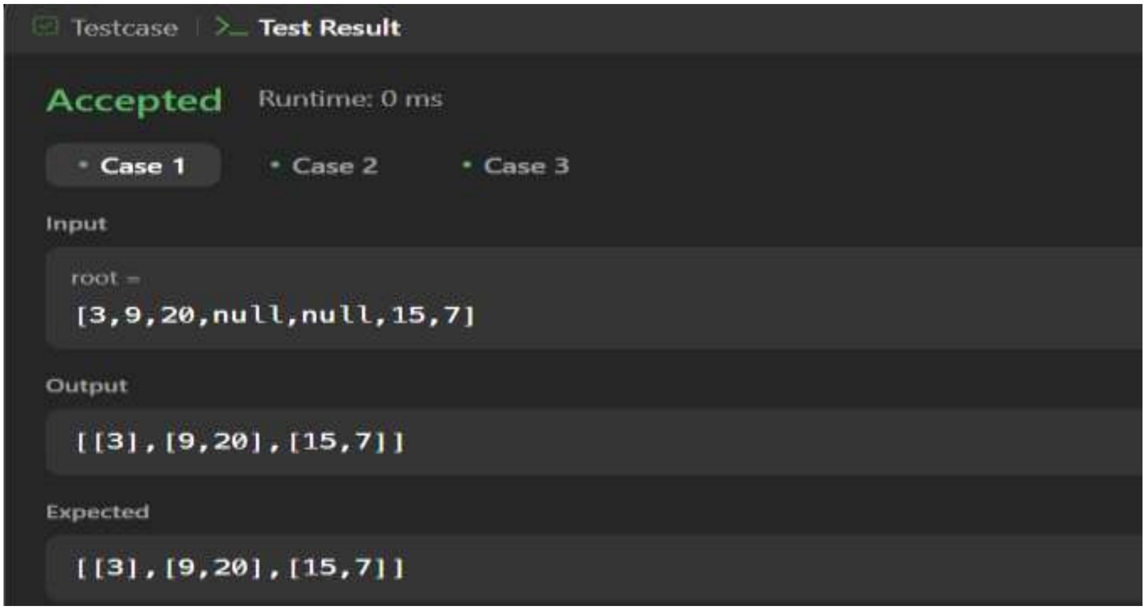

Figure 6.1

Figure 6.2

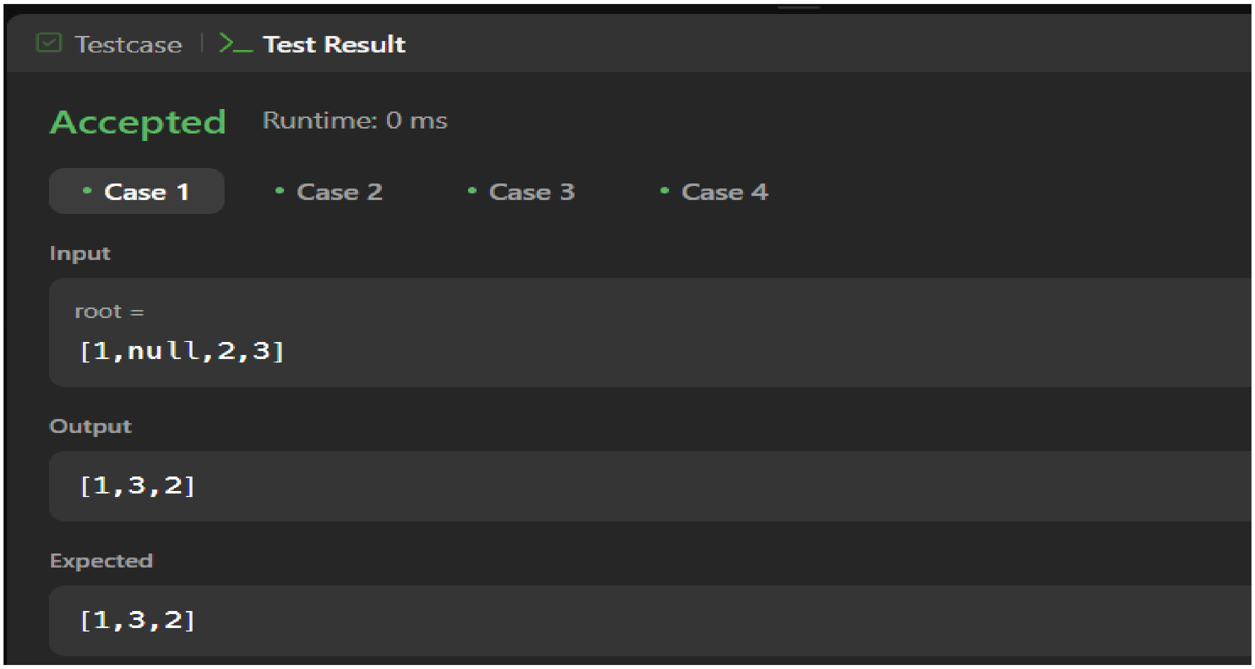

Figure 6.3

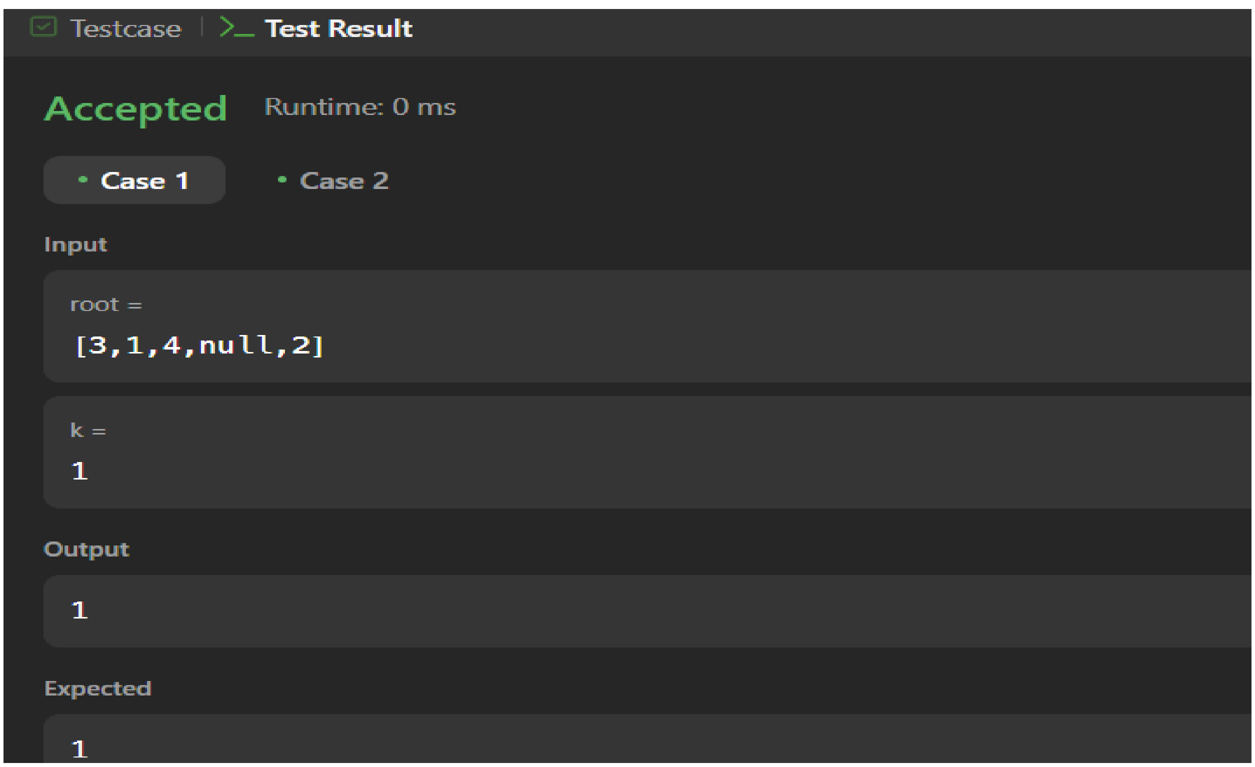

Figure 6.4

5. Learning Outcome :

- Gain familiarity with how intervals are represented in programming (as pairs of start and end points).
- Understand the importance of sorting in simplifying the problem of merging intervals.
- Learn to break down a problem into smaller, manageable parts (sorting, merging) and how to implement these parts in code.
- Learn how to convert between different data structures (e.g., from a list to a 2D array).
- Understand the implications of O(n log n) complexity due to sorting and O(n) for the merging process.