



## Experiment 6

**Student Name:** Rahul Prasad

**UID:** 22BET10167

**Branch:** IT

**Section/Group:** 701/A

**Semester:** 6<sup>th</sup>

**Date :** 05/03/2025

**Subject Name:** Advanced Programming Lab - 2

**Subject Code:** 22ITP-351

### 1. Problem 1:

#### ➤ **Validate Binary Search Tree:**

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.

#### ➤ **Code:**

```
class Solution {  
  
    public boolean isValidBST(TreeNode root) {  
  
        return isValidBST(root, Long.MIN_VALUE, Long.MAX_VALUE);  
  
    }  
  
    private boolean isValidBST(TreeNode node, long min, long max) {
```



# DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
        if (node == null) return true;

        if (node.val <= min || node.val >= max) return false;

        return isValidBST(node.left, min, node.val) && isValidBST(node.right,
            node.val, max);

    }

}
```

## ➤ Output

☒ Testcase | [➤ Test Result](#)

**Accepted** Runtime: 0 ms

- Case 1
- Case 2

**Input**  
root =  
[5,1,4,null,null,3,6]

**Output**  
false

**Expected**  
false

## 2. Problem 2:

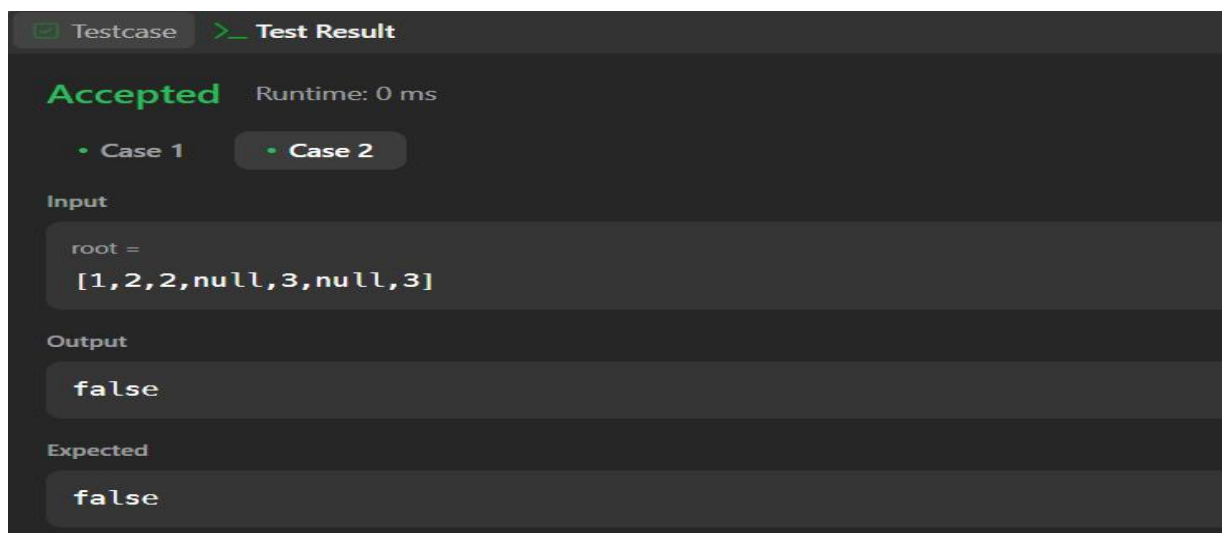
### ➤ Symmetric Tree:

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

## ➤ Code:

```
class Solution {  
  
    public boolean isSymmetric(TreeNode root) {  
  
        if (root == null) return true;  
  
        return isMirror(root.left, root.right);  
  
    }  
  
    private boolean isMirror(TreeNode t1, TreeNode t2) {  
  
        if (t1 == null && t2 == null) return true;  
  
        if (t1 == null || t2 == null) return false;  
  
        return (t1.val == t2.val) && isMirror(t1.left, t2.right)  
  
    }  
  
}
```

## ➤ Output:



The screenshot shows a test result interface with a dark theme. At the top, there are two tabs: 'Testcase' (selected) and 'Test Result'. Below the tabs, the status 'Accepted' is displayed in green, followed by 'Runtime: 0 ms'. There are two buttons for test cases: 'Case 1' and 'Case 2' (selected). Under the 'Input' section, the text 'root =' is followed by the array '[1,2,2,null,3,null,3]'. The 'Output' section shows the result 'false'. The 'Expected' section also shows 'false'.

```
Testcase > Test Result  
  
Accepted Runtime: 0 ms  
  
• Case 1 • Case 2  
  
Input  
root =  
[1,2,2,null,3,null,3]  
  
Output  
false  
  
Expected  
false
```

### 3. Problem - 3:

#### ➤ Binary Tree Level Order Traversal:

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).

#### ➤ Code:

```
import java.util.*;

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.add(root);

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            List<Integer> currentLevel = new ArrayList<>();

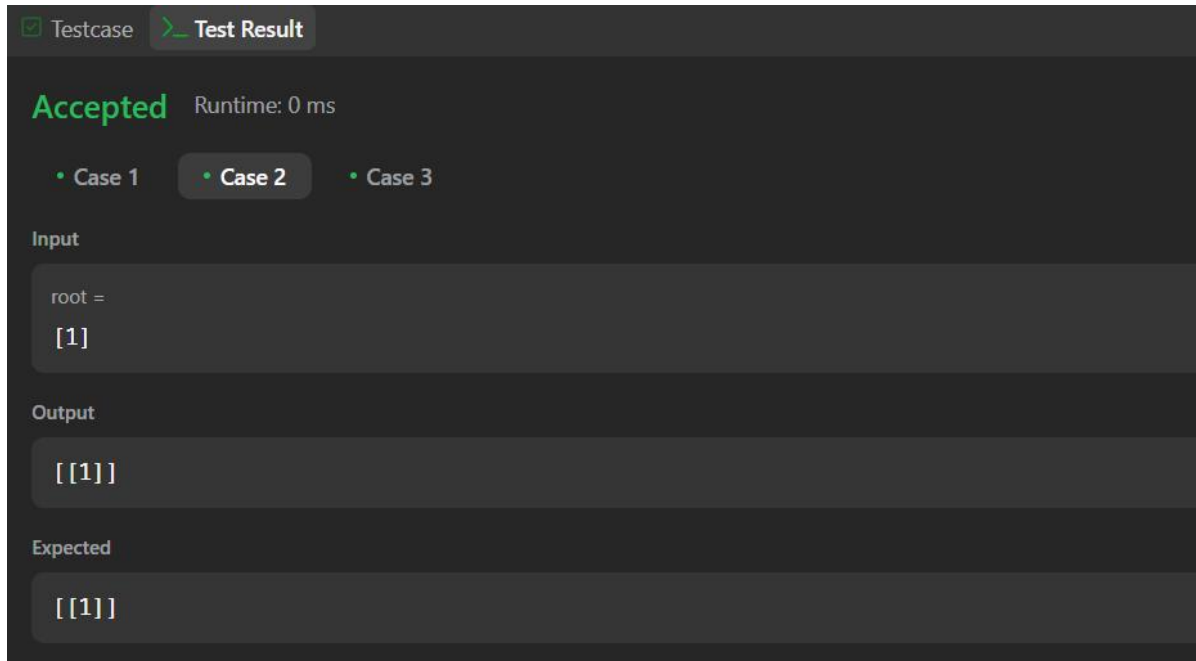
            for (int i = 0; i < levelSize; i++) {
                TreeNode node = queue.poll();
                currentLevel.add(node.val);

                if (node.left != null) queue.add(node.left);
                if (node.right != null) queue.add(node.right);
            }

            result.add(currentLevel);
        }

        return result;
    }
}
```

## ➤ Output:



## 4. Problem - 4:

### ➤ Convert Sorted Array to Binary Search Tree:

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

### ➤ Code:

```
class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        return buildBST(nums, 0, nums.length - 1);
    }

    private TreeNode buildBST(int[] nums, int left, int right) {
        if (left > right) return null;

        int mid = left + (right - left) / 2;
        TreeNode root = new TreeNode(nums[mid]);

        root.left = buildBST(nums, left, mid - 1);
```



# DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
        root.right = buildBST(nums, mid + 1, right);

    return root;
}
```

## ➤ Output:

Testcase | Test Result

**Accepted** Runtime: 0 ms

• Case 1 • Case 2

Input

nums =  
[-10, -3, 0, 5, 9]

Output

[0, -10, 5, null, -3, null, 9]

Expected

[0, -3, 9, -10, null, 5]

## 5. Problem - 5:

### ➤ Binary Tree Inorder Traversal:

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

### ➤ Code:

```
import java.util.*;

class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        inorder(root, result);
        return result;
    }
}
```

```
private void inorder(TreeNode node, List<Integer> result) {  
    if (node == null) return;  
    inorder(node.left, result);  
    result.add(node.val);  
    inorder(node.right, result);  
}  
}
```

## ➤ Output:



The screenshot shows a test result interface with a dark theme. At the top, there are tabs for 'Testcase' and 'Test Result', with 'Test Result' being the active tab. Below the tabs, the status 'Accepted' is displayed in green, followed by 'Runtime: 0 ms'. There are four test cases listed: 'Case 1', 'Case 2', 'Case 3', and 'Case 4'. 'Case 2' is selected and highlighted. Below the test cases, there are three sections: 'Input', 'Output', and 'Expected'. Each section contains a text box with the following content: 'Input' shows 'root = [1,2,3,4,5,null,8,null,null,6,7,9]'; 'Output' shows '[4,2,6,5,7,1,3,9,8]'; and 'Expected' shows '[4,2,6,5,7,1,3,9,8]'.

## 6. Problem - 6:

### ➤ Construct Binary Tree from Inorder and Postorder Traversal:

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.

**➤ Code:**

```
import java.util.*;

class Solution {
    private int postIndex;
    private Map<Integer, Integer> inorderMap;

    public TreeNode buildTree(int[] inorder, int[] postorder) {
        postIndex = postorder.length - 1;
        inorderMap = new HashMap<>();

        for (int i = 0; i < inorder.length; i++) {
            inorderMap.put(inorder[i], i);
        }

        return constructTree(postorder, 0, inorder.length - 1);
    }

    private TreeNode constructTree(int[] postorder, int left, int right) {
        if (left > right) return null;

        int rootValue = postorder[postIndex--];
        TreeNode root = new TreeNode(rootValue);

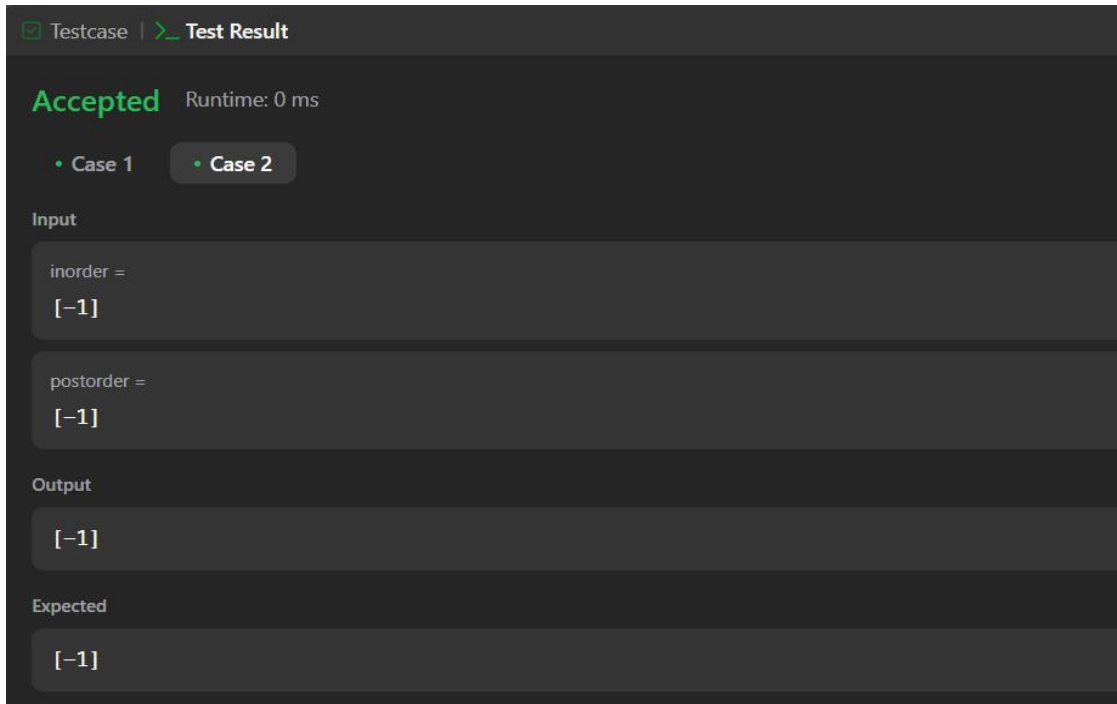
        int inorderIndex = inorderMap.get(rootValue);

        root.right = constructTree(postorder, inorderIndex + 1, right);
        root.left = constructTree(postorder, left, inorderIndex - 1);

        return root;
    }
}
```



## ➤ Output:



## 7. Problem - 7:

### ➤ Kth Smallest Element in a BST:

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

### ➤ Code:

```
import java.util.*;

class Solution {
    public int kthSmallest(TreeNode root, int k) {
        Stack<TreeNode> stack = new Stack<>();
        TreeNode curr = root;

        while (curr != null || !stack.isEmpty()) {
            while (curr != null) {
                stack.push(curr);
                curr = curr.left;
            }
```



# DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
        }  
  
        curr = stack.pop();  
        k--;  
  
        if (k == 0) return curr.val;  
  
        curr = curr.right;  
    }  
  
    return -1;  
}  
}
```

## ➤ OutPut:

☒ Testcase | [Test Result](#)

**Accepted** Runtime: 0 ms

- Case 1
- Case 2

Input

root =  
[5,3,6,2,4,null,null,1]

k =  
3

Output

3

Expected

3

## 8. Problem - 8:

### ➤ **Populating Next Right Pointers in Each Node:**

You are given a perfect binary tree where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {  
    int val;  
    Node *left;  
    Node *right;  
    Node *next;  
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

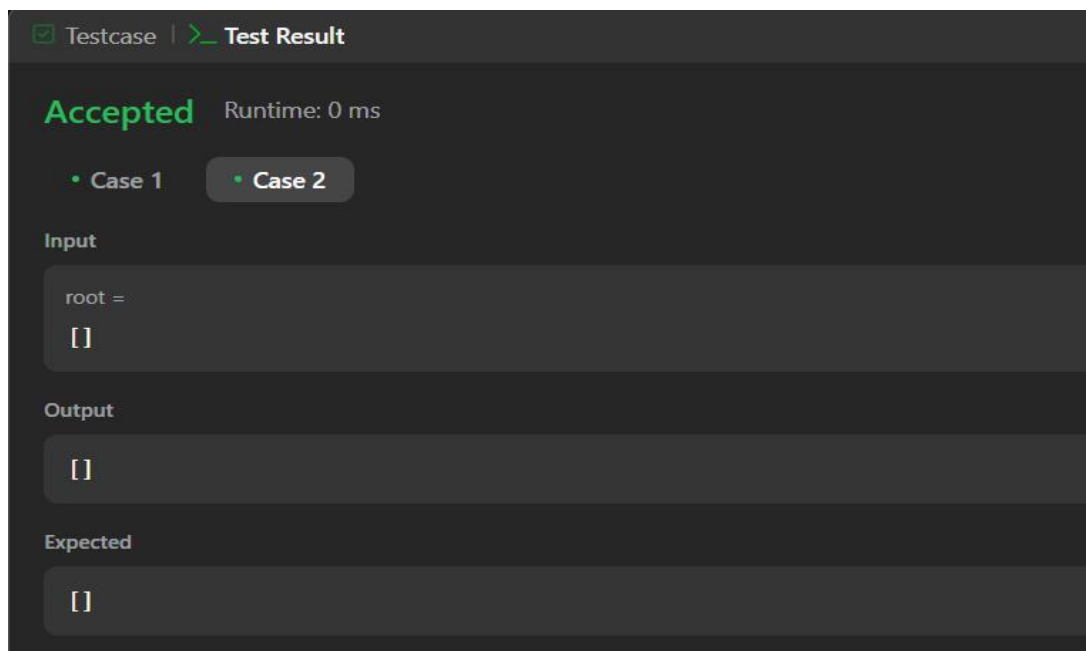
.

### ➤ **Code:**

```
import java.util.*;  
  
class Solution {  
    public Node connect(Node root) {  
        if (root == null) return null;  
  
        Queue<Node> queue = new LinkedList<>();  
        queue.add(root);  
  
        while (!queue.isEmpty()) {  
            int size = queue.size();  
            Node prev = null;  
  
            for (int i = 0; i < size; i++) {  
                Node node = queue.poll();  
  
                if (prev != null) prev.next = node;  
                prev = node;  
            }  
        }  
    }  
}
```

```
        if (node.left != null) queue.add(node.left);  
        if (node.right != null) queue.add(node.right);  
    }  
}  
  
return root;  
}  
}
```

## ➤ OutPut:



## ❖ Learning Outcomes:

- Understanding Tree Traversal – Learn how to traverse a binary tree level by level using BFS (Queue) or Next Pointers.
- Efficient Node Linking – Gain the ability to link sibling nodes at each level using either extra space (Queue) or constant space (Next Pointers).
- Optimizing Space Complexity – Understand how to eliminate extra space usage by leveraging already established next pointers.
- Handling Edge Cases – Learn to manage null nodes, leaf nodes, and tree boundaries while setting up next pointers correctly.
- Applying in Real-World Scenarios – Develop insights into how this approach is useful for networking (packet forwarding), multi-threading, and tree-based data structures.