



**DEPARTMENT OF**

**COMPUTER SCIENCE & ENGINEERING**

Discover. Learn. Empower.

### Experiment 6

**Student Name:** RIYA THAKUR

**UID:**22BET10171

**Branch:** BE -IT

**Section/Group:**22BET/IOT/702/A

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

**Date of Performance:**28/02/2025

**Subject Name:** Advanced Programming Lab-2 **Subject Code:** 22ITP-351

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

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

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

#### **2. 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:

- 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.

#### **3. Symmetric Tree:**

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

#### **4. 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).

#### **5. 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.

#### **6. Binary Tree Inorder Traversal:**

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

#### **7. Binary Tree Zigzag Level Order Traversal :**

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).



## **8. 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.

## **9. Kth Smallest Element in a BST :**

Given the root of a binary search tree, and an integer k, return the  $k^{\text{th}}$  smallest value (1-indexed) of all the values of the nodes in the tree.

## **10. 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;  
}
```

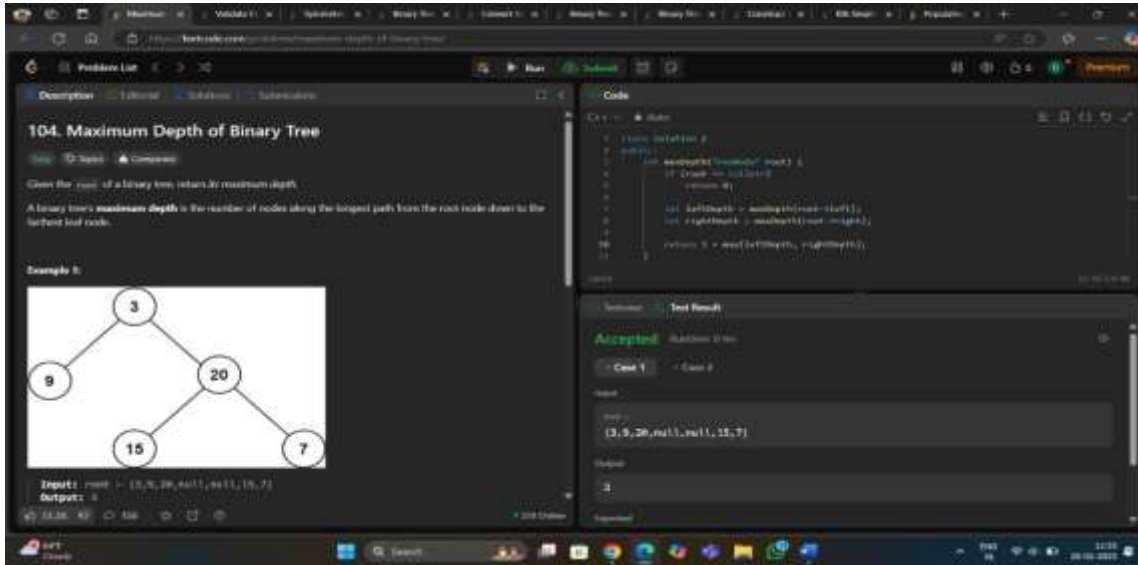
Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

## **11.Objective :**

1. Learn how to recursively or iteratively compute the height (or depth) of a binary tree.
2. Understand the properties of a Binary Search Tree (BST).
3. Understand mirror properties of a binary tree.
4. Learn Breadth-First Search (BFS) using queues.
5. Learn how to construct a height-balanced BST from a sorted array.
6. Understand inorder traversal (left-root-right).
7. Extend BFS level-order traversal to include zigzag ordering.
8. Understand tree reconstruction from traversal orders.
9. Use inorder traversal (sorted order) to efficiently find the kth smallest element.
10. Understand tree linking using level pointers.

## 12. Implementation of Code/Output 1 :



**104. Maximum Depth of Binary Tree**

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

A binary tree's *maximum depth* is the number of nodes along the longest path from the root node down to the farthest leaf node.

**Example 1:**

```

graph TD
    3((3)) --- 9((9))
    3 --- 20((20))
    20 --- 15((15))
    20 --- 7((7))
  
```

**Input:** root = [3,9,20,null,null,15,7]  
**Output:** 3

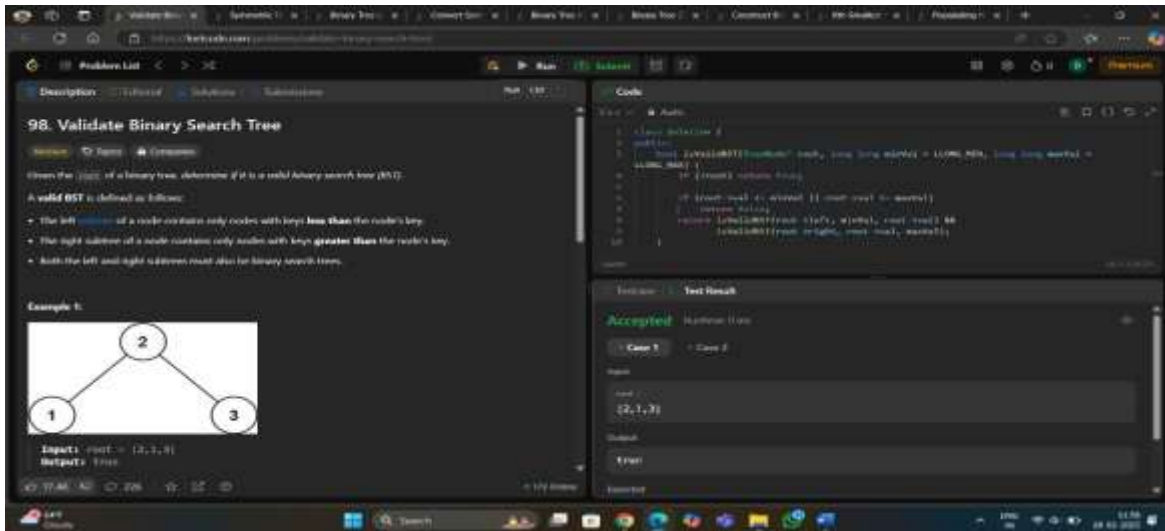
```

1 int maxDepth(TreeNode* root) {
2     if (!root) return 0;
3     return 1 + max(maxDepth(root->left),
4                     maxDepth(root->right));
5 }
  
```

**Test Result:** Accepted

**Case 1:** Input: root = [3,9,20,null,null,15,7], Output: 3

## 13.Code 2 :



**98. 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*.

**Example 1:**

```

graph TD
    2((2)) --- 1((1))
    2 --- 3((3))
  
```

**Input:** root = [2,1,3]  
**Output:** true

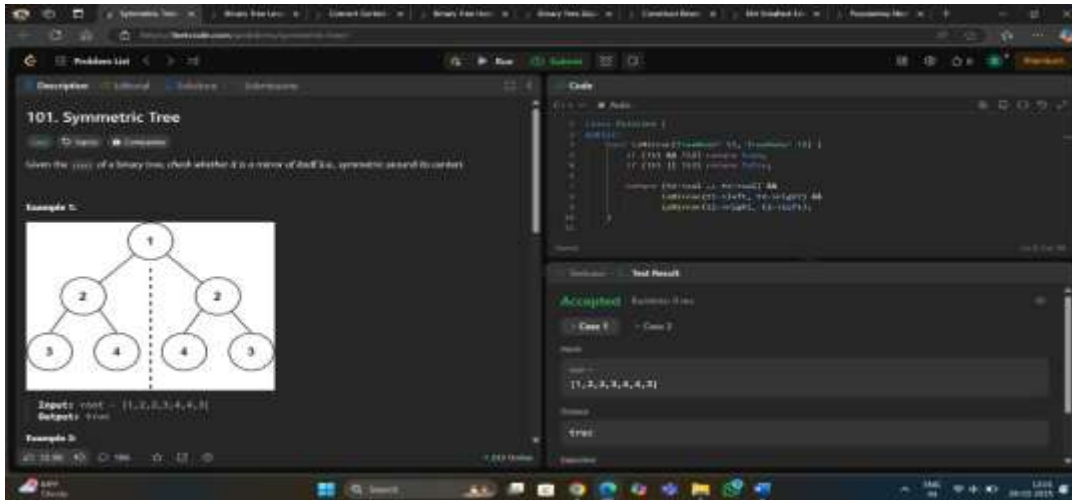
```

1 class Solution {
2 public:
3     bool isValidBST(TreeNode* root) {
4         long long minVal = LONG_MIN, long long maxVal =
5             LONG_MAX;
6         return isValid(root, minVal, maxVal);
7     }
8     bool isValid(TreeNode* root, long long minVal, long long maxVal) {
9         if (!root) return true;
10        if (root->val <= minVal || root->val >= maxVal)
11            return false;
12        return isValid(root->left, minVal, root->val) &&
13            isValid(root->right, root->val, maxVal);
14    }
15 };
  
```

**Test Result:** Accepted

**Case 1:** Input: root = [2,1,3], Output: true

## 14.Code 3 :



**101. Symmetric Tree**

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

**Example 1:**

```

graph TD
    1((1)) --- 2L((2))
    1 --- 2R((2))
    2L --- 3((3))
    2L --- 4((4))
    2R --- 4((4))
    2R --- 3((3))
  
```

Input: root = [1,2,2,3,4,4,3]  
Output: true

**Example 2:**

```

graph TD
    1((1)) --- 2L((2))
    1 --- 3((3))
    2L --- 4((4))
    2L --- 5((5))
    3 --- 6((6))
    3 --- 7((7))
  
```

Input: root = [1,2,3,4,5,6,7]  
Output: false

```

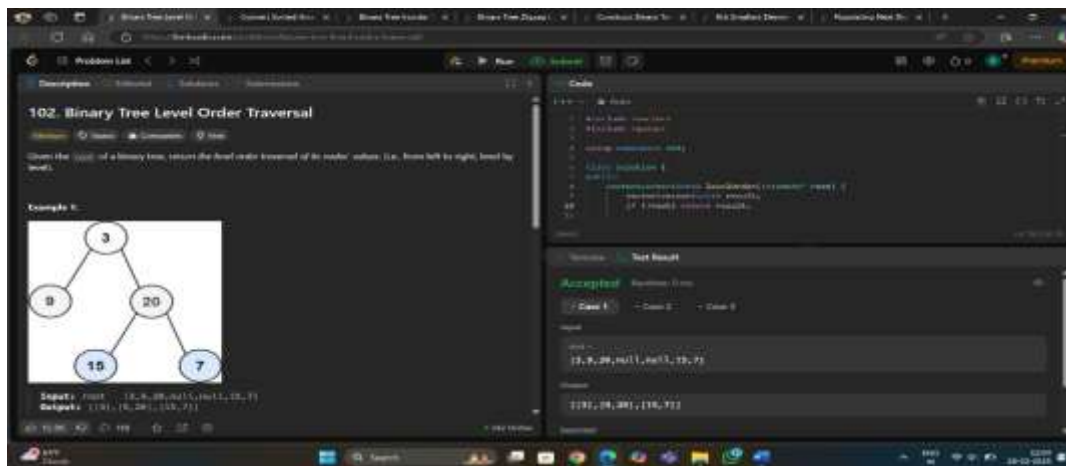
C++
1 // Definition for a binary tree node.
2 struct TreeNode {
3     int val;
4     TreeNode *left;
5     TreeNode *right;
6     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
7 };
8
9 bool isSymmetric(TreeNode* root) {
10    return isMirror(root, root);
11 }
12
13 bool isMirror(TreeNode* t1, TreeNode* t2) {
14    if (t1 == NULL && t2 == NULL) return true;
15    if (t1 == NULL || t2 == NULL) return false;
16    if (t1->val != t2->val) return false;
17    return isMirror(t1->left, t2->right) && isMirror(t1->right, t2->left);
18 }
  
```

Test Result: Accepted

Case 1: [1,2,2,3,4,4,3]

Output: true

## 15.Code 4 :



**102. 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).

**Example 1:**

```

graph TD
    3((3)) --- 9((9))
    3 --- 20((20))
    9 --- 15((15))
    20 --- 7((7))
  
```

Input: root = [3,9,20,null,null,15,7]  
Output: [[3],[9,20],[15,7]]

```

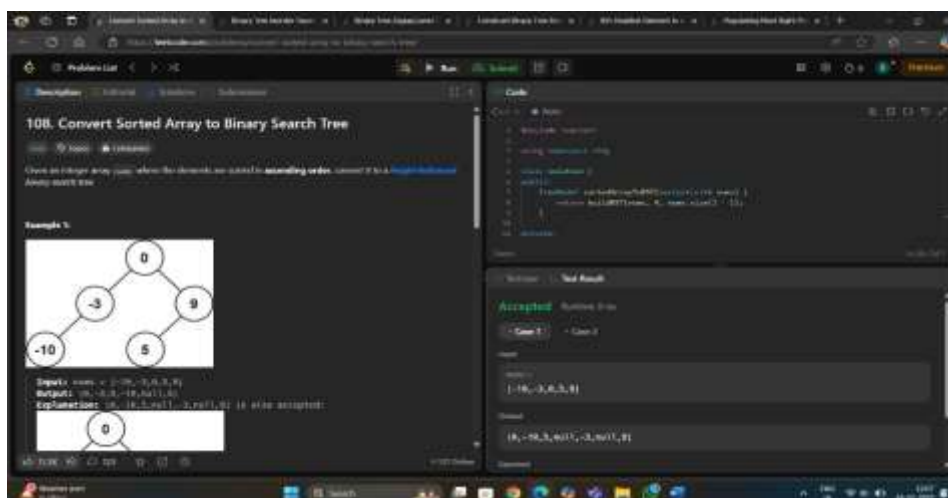
C++
1 // Definition for a binary tree node.
2 struct TreeNode {
3     int val;
4     TreeNode *left;
5     TreeNode *right;
6     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
7 };
8
9 vector<vector<int>> levelOrder(TreeNode* root) {
10    if (root == NULL) return {};
11    vector<vector<int>> result;
12    queue<TreeNode*> q;
13    q.push(root);
14    while (!q.empty()) {
15        int size = q.size();
16        vector<int> level;
17        for (int i = 0; i < size; i++) {
18            TreeNode* node = q.front();
19            q.pop();
20            level.push_back(node->val);
21            if (node->left) q.push(node->left);
22            if (node->right) q.push(node->right);
23        }
24        result.push_back(level);
25    }
26    return result;
27 }
  
```

Test Result: Accepted

Case 1: [3,9,20,null,null,15,7]

Output: [[3],[9,20],[15,7]]

## 16.Code 5 :



**108. 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.

**Example 1:**

```

graph TD
    0((0)) --- -3((-3))
    0 --- 9((9))
    -3 --- -10((-10))
    9 --- 5((5))
  
```

Input: nums = [-10,-3,0,5,9]  
Output: [0,-3,9,-10,5]  
Explanation: [0,-3,9,-10,5] is also accepted: [0,-10,5,-3,9]

```

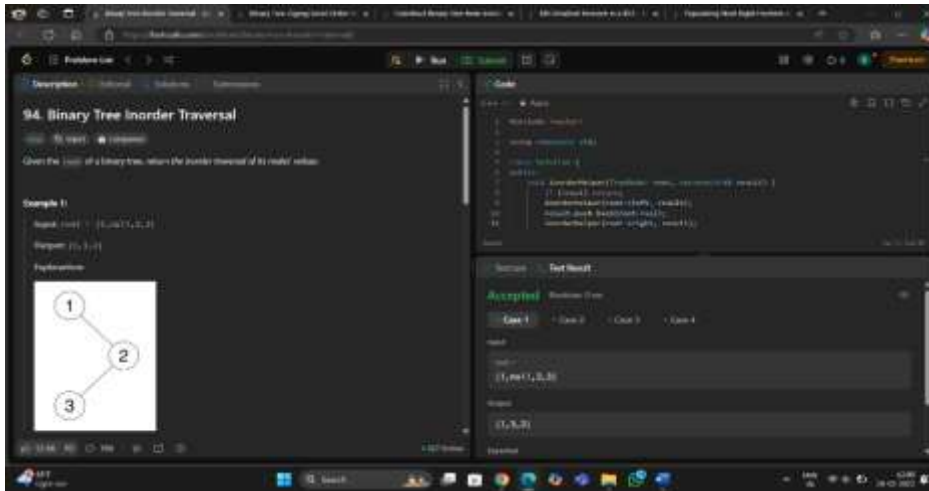
C++
1 // Definition for a binary tree node.
2 struct TreeNode {
3     int val;
4     TreeNode *left;
5     TreeNode *right;
6     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
7 };
8
9 TreeNode* sortedArrayToBST(vector<int> &nums) {
10    return helper(nums, 0, nums.size() - 1);
11 }
12
13 TreeNode* helper(vector<int> &nums, int left, int right) {
14    if (left > right) return NULL;
15    int mid = (left + right) / 2;
16    TreeNode* node = new TreeNode(nums[mid]);
17    node->left = helper(nums, left, mid - 1);
18    node->right = helper(nums, mid + 1, right);
19    return node;
20 }
  
```

Test Result: Accepted

Case 1: [-10,-3,0,5,9]

Output: [0,-3,9,-10,5]

## 17.Code 6 :



**94. Binary Tree Inorder Traversal**

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

**Example 1:**  
Input: *root* = [1, null, 2, 3]  
Output: [1, 3, 2]

**Diagram:**  
A binary tree with root node 1. Node 1 has a left child 3 and a right child 2.

```

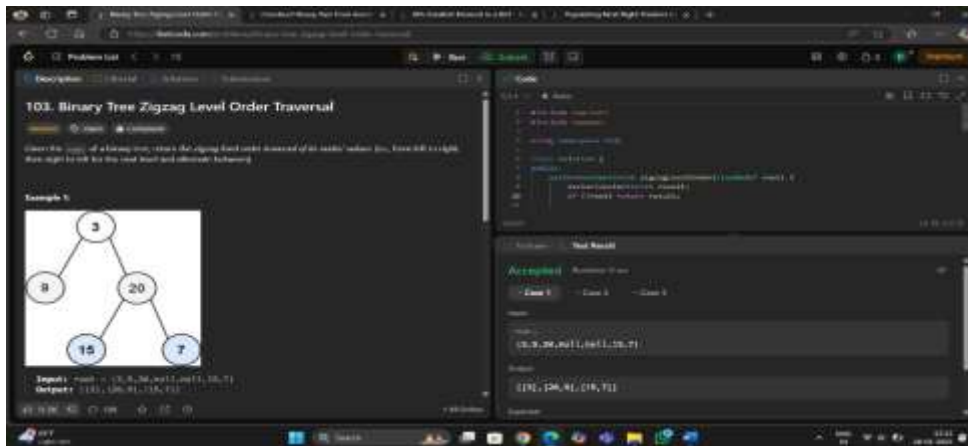
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        inorderTraversalHelper(root, result);
        return result;
    }
private:
    void inorderTraversalHelper(TreeNode* root, vector<int> &result) {
        if (root == nullptr) return;
        inorderTraversalHelper(root->left, result);
        result.push_back(root->val);
        inorderTraversalHelper(root->right, result);
    }
};

```

**Test Result:** Accepted

**Case 1:** Input: [1, null, 2, 3], Output: [1, 3, 2]

## 18.Code 7 :



**103. Binary Tree Zigzag Level Order Traversal**

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

**Example 1:**  
Input: *root* = [3, 9, 20, 15, 7]  
Output: [[3], [9, 20], [15, 7]]

**Diagram:**  
A binary tree with root node 3. Node 3 has a left child 9 and a right child 20. Node 9 has a left child 15, and node 20 has a right child 7.

```

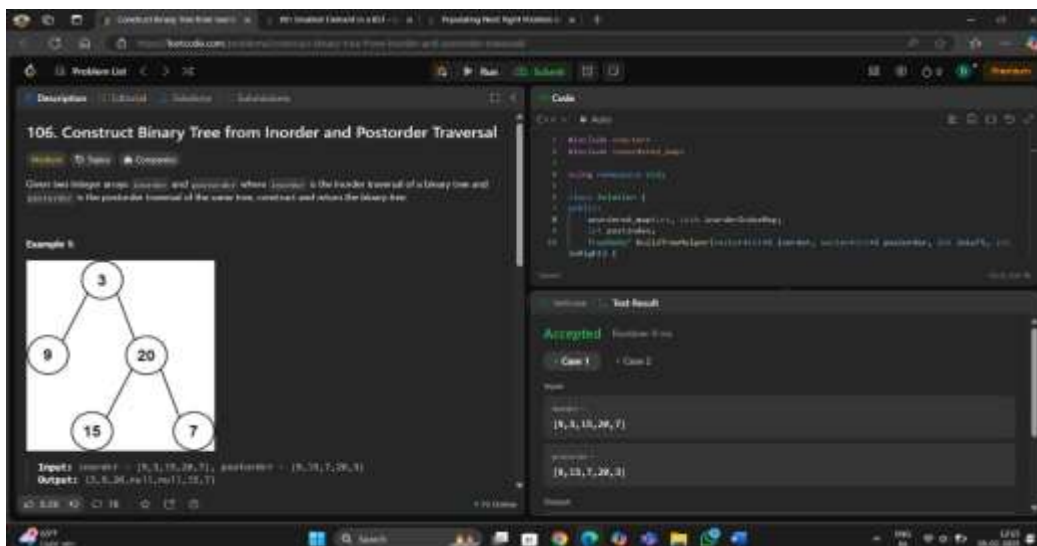
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> result;
        zigzagLevelOrderHelper(root, 0, result);
        return result;
    }
private:
    void zigzagLevelOrderHelper(TreeNode* root, int level, vector<vector<int>> &result) {
        if (root == nullptr) return;
        if (level < result.size()) {
            result[level].push_back(root->val);
        } else {
            result.push_back({root->val});
        }
        zigzagLevelOrderHelper(root->left, level + 1, result);
        zigzagLevelOrderHelper(root->right, level + 1, result);
    }
};

```

**Test Result:** Accepted

**Case 1:** Input: [3, 9, 20, 15, 7], Output: [[3], [9, 20], [15, 7]]

## 19.Code 8 :



**106. 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.

**Example 1:**  
Input: *inorder* = [9, 3, 15, 20, 7], *postorder* = [9, 15, 20, 7, 3]  
Output: [3, 9, 15, 20, 7]

**Diagram:**  
A binary tree with root node 3. Node 3 has a left child 9 and a right child 20. Node 9 has a left child 15, and node 20 has a right child 7.

```

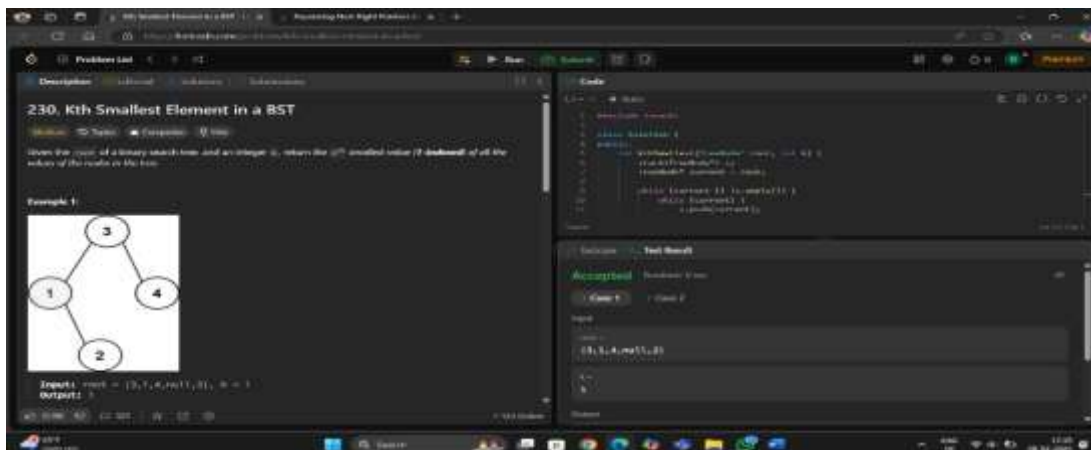
class Solution {
public:
    TreeNode* buildTree(vector<int> &inorder, vector<int> &postorder) {
        return buildTreeHelper(inorder, 0, inorder.size() - 1, postorder, 0, postorder.size() - 1);
    }
private:
    TreeNode* buildTreeHelper(vector<int> &inorder, int inStart, int inEnd, vector<int> &postorder, int postStart, int postEnd) {
        if (inStart > inEnd || postStart > postEnd) return nullptr;
        int rootVal = postorder[postEnd];
        int rootIndex = inStart;
        while (inorder[rootIndex] != rootVal) rootIndex++;
        int leftInStart = inStart, leftInEnd = rootIndex - 1;
        int leftPostStart = postStart, leftPostEnd = rootIndex - inStart - 1;
        int rightInStart = rootIndex, rightInEnd = inEnd;
        int rightPostStart = rootIndex - inStart, rightPostEnd = postEnd - 1;
        TreeNode* root = new TreeNode(rootVal);
        root->left = buildTreeHelper(inorder, leftInStart, leftInEnd, postorder, leftPostStart, leftPostEnd);
        root->right = buildTreeHelper(inorder, rightInStart, rightInEnd, postorder, rightPostStart, rightPostEnd);
        return root;
    }
};

```

**Test Result:** Accepted

**Case 1:** Input: [9, 3, 15, 20, 7], [9, 15, 20, 7, 3], Output: [3, 9, 15, 20, 7]

## 20.Code 9:



**230. 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.

**Example 1:**

```

graph TD
    3((3)) --- 1((1))
    3((3)) --- 4((4))
    1((1)) --- 2((2))
  
```

**Inputs:** root = [3,1,4,null,2], k = 3  
**Output:** 3

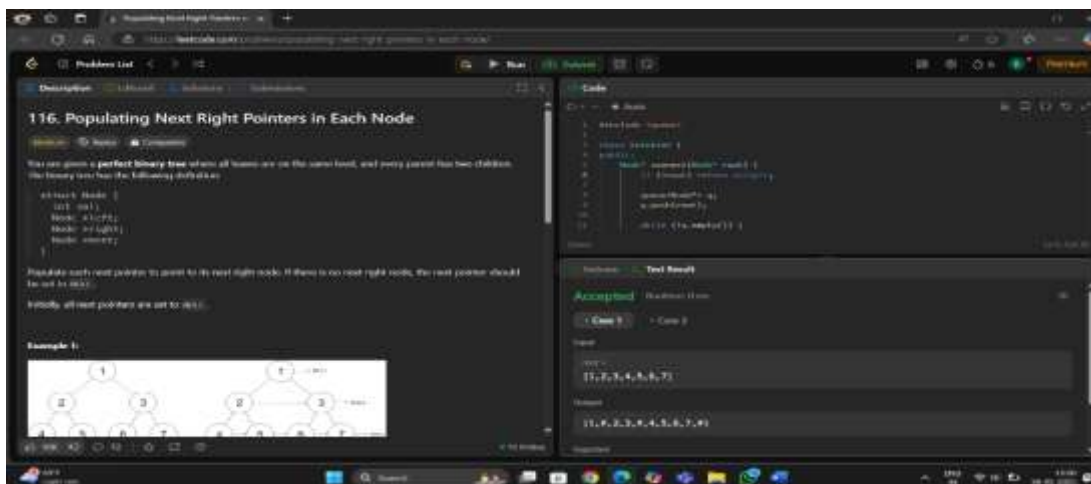
```

classmate
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        return kthSmallest(root, 0, 0);
    }
    int kthSmallest(TreeNode* root, int k, int count) {
        if (root == null) return 0;
        int left = kthSmallest(root->left, k, count);
        if (left > 0) return left;
        count++;
        if (count == k) return root->val;
        return kthSmallest(root->right, k, count);
    }
};
  
```

**Test Result:** Accepted

**Input:** [3,1,4,null,2], 3  
**Output:** 3

## 21.Code 10 :



**116. 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.

**Example 1:**

```

graph LR
    1((1)) --- 2((2))
    1((1)) --- 3((3))
    2((2)) --- 4((4))
    2((2)) --- 5((5))
    3((3)) --- 6((6))
    3((3)) --- 7((7))
  
```

**Input:** [1,2,3,4,5,6,7]  
**Output:** [1,2,3,4,5,6,7]

```

classmate
class Solution {
public:
    Node* connect(Node* root) {
        if (root == null) return null;
        root->next = root->right;
        return root;
    }
};
  
```

**Test Result:** Accepted

**Input:** [1,2,3,4,5,6,7]  
**Output:** [1,2,3,4,5,6,7]

## 22.Learning Outcome :

1. Concepts: Depth-first search (DFS), Recursion
2. A valid BST requires every node's left child to be smaller and right child to be larger.
3. A tree is symmetric if its left subtree is a mirror of the right subtree.
4. Concepts: Recursion, Divide & Conquer, Balanced BST
5. Concepts: Inorder traversal, Recursion, Iterative (Stack)
6. Concepts: Tree Reconstruction, Recursion, Hash Map Optimization