**Name : Baljeet UID : 22BCS11134 Section : 605-B**

**Ques 1. Binary Tree Inorder Traversal.**

**Code:**

class Solution {

public:

void inorder(TreeNode\* root, vector<int>& result) {

if (!root) return;

inorder(root->left, result);

result.push\_back(root->val);

inorder(root->right, result);

}

vector<int> inorderTraversal(TreeNode\* root) {

vector<int> result;

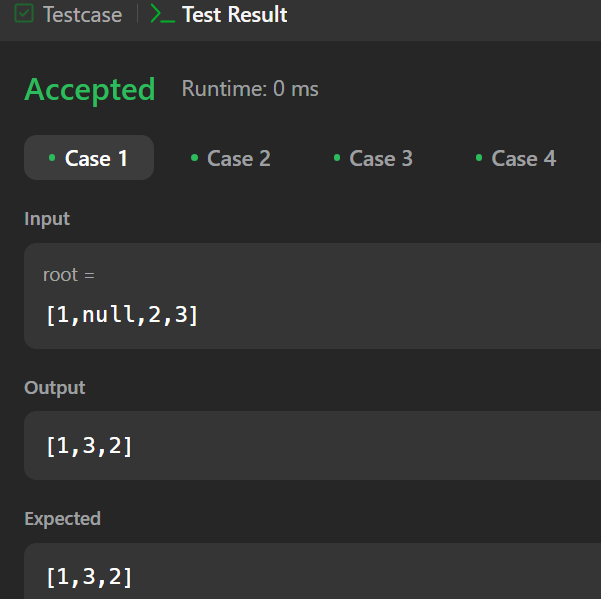
inorder(root, result);

return result;

}

};

**Output:**



**Ques 2. Symmetric Tree.**

**Code:**

class Solution {

public:

bool isMirror(TreeNode\* t1, TreeNode\* t2) {

if (!t1 && !t2) return true;

if (!t1 || !t2) return false;

return (t1->val == t2->val) &&

isMirror(t1->left, t2->right) &&

isMirror(t1->right, t2->left);

}

bool isSymmetric(TreeNode\* root) {

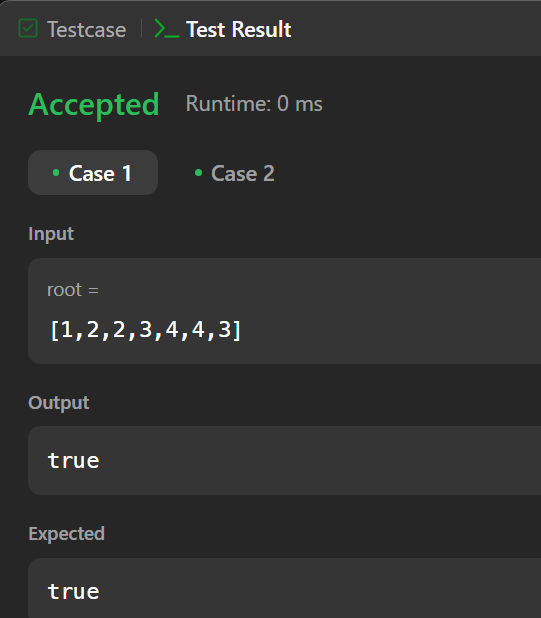
if (!root) return true;

return isMirror(root->left, root->right);

}

};

**Output:**



**Ques 3. Maximum Depth of Binary Tree.**

**Code:**

class Solution {

public:

int maxDepth(TreeNode\* root) {

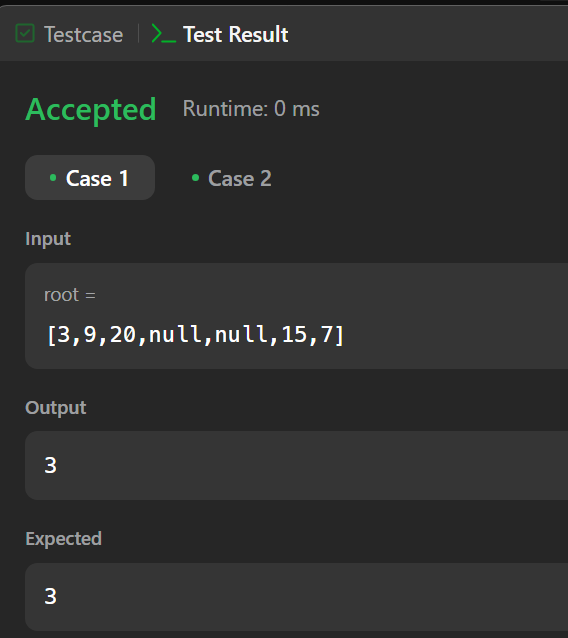
if (!root) return 0;

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

}

};

**Output:**



**Ques 4. Validate Binary Search Tree.**

**Code:**

class Solution {

public:

bool validate(TreeNode\* node, long minVal, long maxVal) {

if (!node) return true;

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

return validate(node->left, minVal, node->val) && validate(node->right, node->val, maxVal);

}

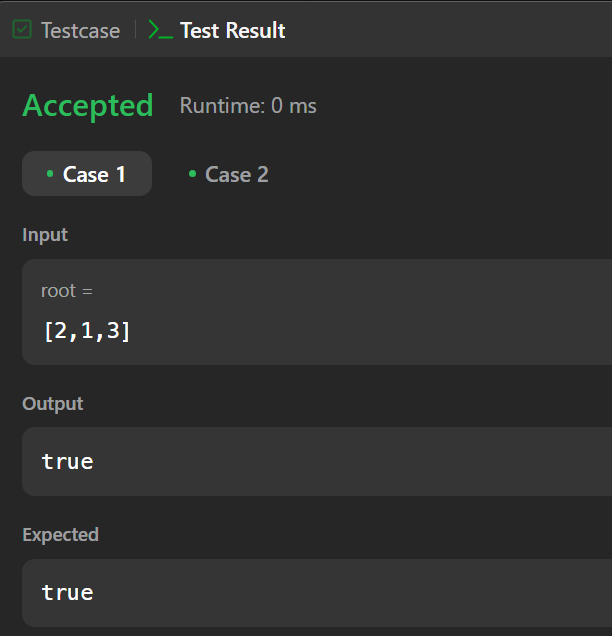
bool isValidBST(TreeNode\* root) {

return validate(root, LONG\_MIN, LONG\_MAX);

}

};

**Output:**

****

**Ques 5. Kth Smallest Element ina BST.**

**Code:**

class Solution {

public:

void inorder(TreeNode\* root, vector<int>& elements) {

if (!root) return;

inorder(root->left, elements);

elements.push\_back(root->val);

inorder(root->right, elements);

}

int kthSmallest(TreeNode\* root, int k) {

vector<int> elements;

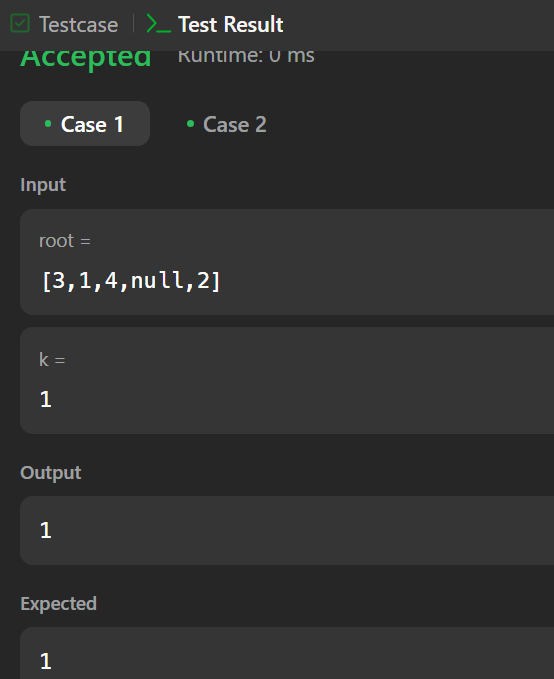
inorder(root, elements);

return elements[k - 1]; // Since k is 1-indexed

}

};

**Output:**



**Ques 6. Binary Tree Level Order Traversal.**

**Code:**

class Solution {

public:

vector<vector<int>> levelOrder(TreeNode\* root) {

vector<vector<int>> result;

if (!root) return result;

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);

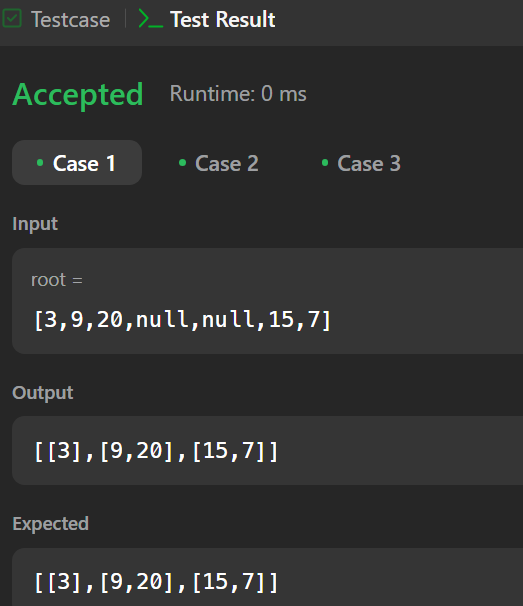
}

result.push\_back(level); }

return result; }

};

**Output:**



**Ques 7. Binary Tree Level Order Traversal II.**

**Code:**

class Solution {

public:

vector<vector<int>> levelOrderBottom(TreeNode\* root) {

vector<vector<int>> result;

if (!root) return result;

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);

}

result.push\_back(level);

}

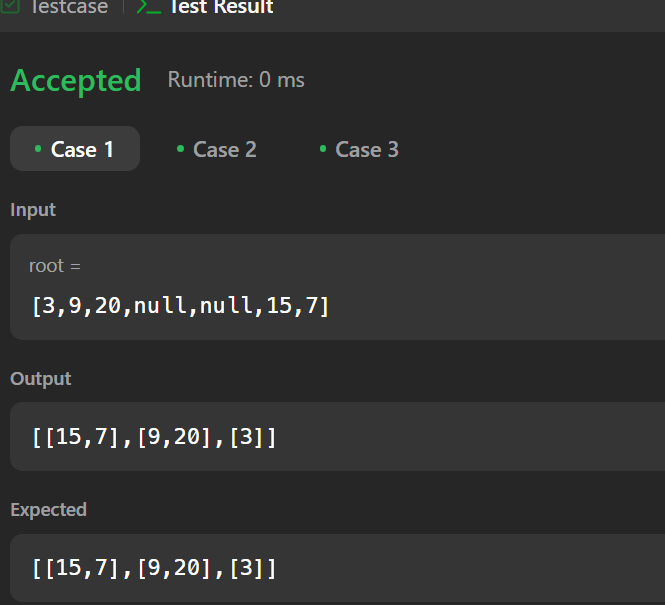
reverse(result.begin(), result.end());

return result;

}

};

**Output:**



**Ques 8. Binary Tree Zigzag Level Order Traversal.**

**Code:**

class Solution {

public:

vector<vector<int>> zigzagLevelOrder(TreeNode\* root) {

vector<vector<int>> result;

if (!root) return result;

queue<TreeNode\*> q;

q.push(root);

bool leftToRight = true;

while (!q.empty()) {

int size = q.size();

deque<int> level; // Use deque for flexible insertion

for (int i = 0; i < size; i++) {

TreeNode\* node = q.front();

q.pop();

if (leftToRight)

level.push\_back(node->val);

else

level.push\_front(node->val);

if (node->left) q.push(node->left);

if (node->right) q.push(node->right);

}

result.push\_back(vector<int>(level.begin(), level.end()));

leftToRight = !leftToRight; // Toggle direction

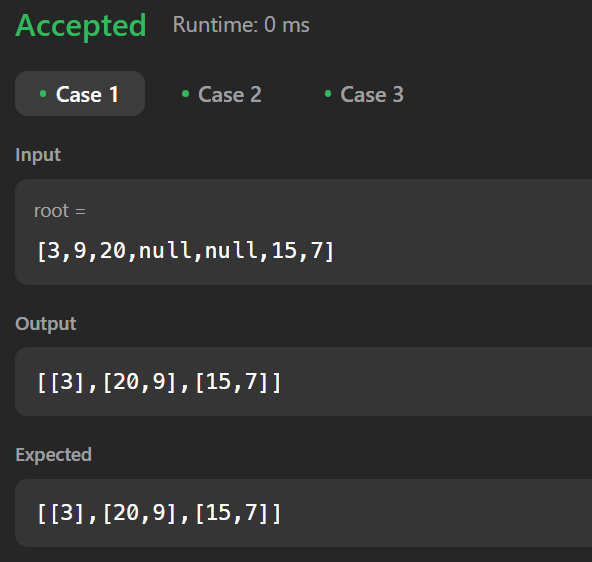
}

return result;

}

};

**Output:**

****

**Ques 9. Binary Tree Right Side View.**

**Code:**

class Solution {

public:

vector<int> rightSideView(TreeNode\* root) {

vector<int> result;

if (!root) return result;

queue<TreeNode\*> q;

q.push(root);

while (!q.empty()) {

int size = q.size();

int rightmost = 0;

for (int i = 0; i < size; i++) {

TreeNode\* node = q.front();

q.pop();

rightmost = node->val; // Store last node of the level

if (node->left) q.push(node->left);

if (node->right) q.push(node->right);

}

result.push\_back(rightmost);

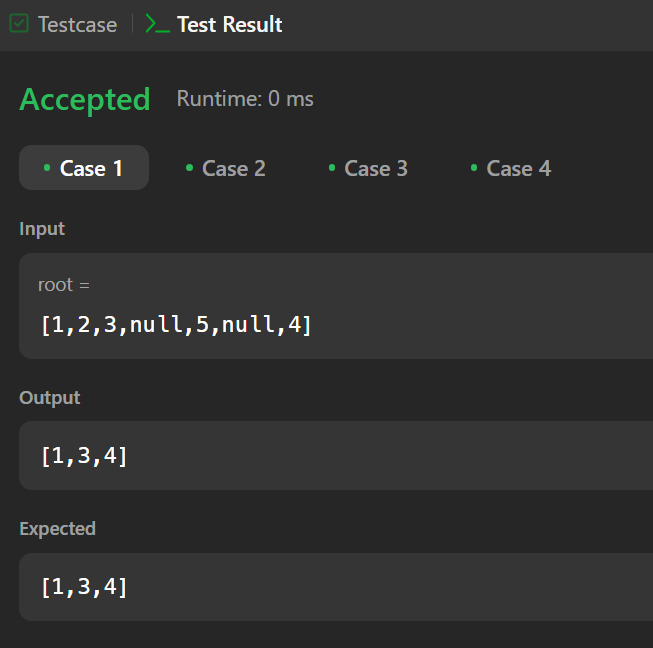
}

return result;

}

};

**Output:**

****

**Ques 10. Construct Binary Tree from Inorder and Postorder Traversal.**

**Code:**

class Solution {

public:

unordered\_map<int, int> inorderMap; // To store inorder value -> index mapping

int postIndex; // To track the current root index in postorder

TreeNode\* buildTreeHelper(vector<int>& inorder, vector<int>& postorder, int left, int right) {

if (left > right) return nullptr; // Base case

// Get the root from the postorder traversal

int rootVal = postorder[postIndex--];

TreeNode\* root = new TreeNode(rootVal);

// Get index of root in inorder array

int inorderIdx = inorderMap[rootVal];

// Build right subtree first (since postorder processes left first, we process right first)

root->right = buildTreeHelper(inorder, postorder, inorderIdx + 1, right);

root->left = buildTreeHelper(inorder, postorder, left, inorderIdx - 1);

return root;

}

TreeNode\* buildTree(vector<int>& inorder, vector<int>& postorder) {

postIndex = postorder.size() - 1;

// Store inorder indices for quick lookup

for (int i = 0; i < inorder.size(); i++) {

inorderMap[inorder[i]] = i;

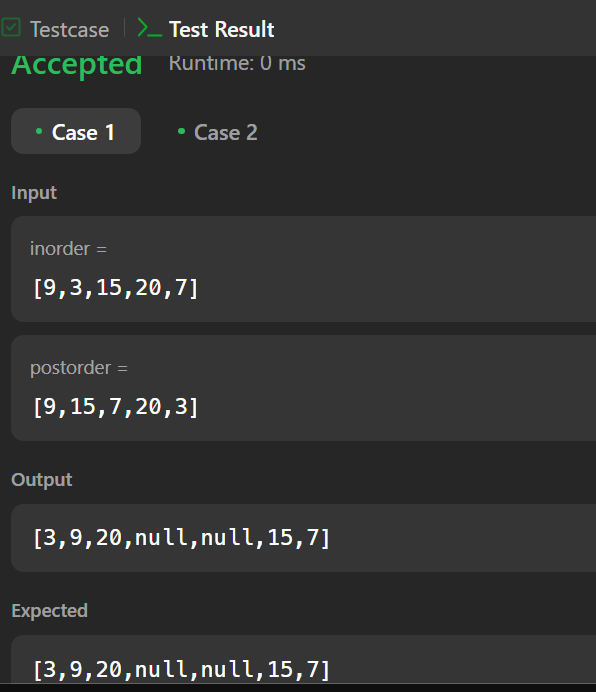
}

return buildTreeHelper(inorder, postorder, 0, inorder.size() - 1);

}

};

**Output:**



**Ques 11. Find Bottom Left Tree Value.**

**Code:**

class Solution {

public:

int findBottomLeftValue(TreeNode\* root) {

queue<TreeNode\*> q;

q.push(root);

int bottomLeft = root->val; // Initialize with root value

while (!q.empty()) {

int size = q.size();

bottomLeft = q.front()->val; // First element in this level

for (int i = 0; i < size; i++) {

TreeNode\* node = q.front();

q.pop();

// Push left child first (ensures leftmost node is encountered first)

if (node->left) q.push(node->left);

if (node->right) q.push(node->right);

}

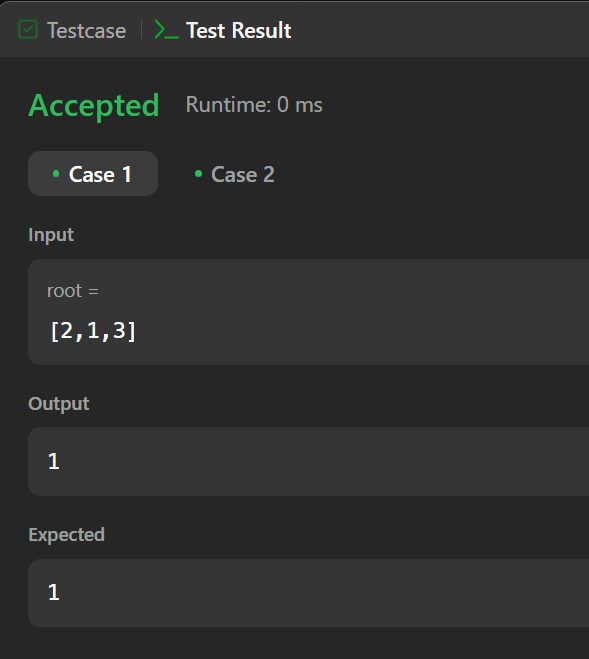
}

return bottomLeft;

}

};

**Output:**

****

**Ques 12. Binary Tree Maximum Path Sum**.

**Code:**

class Solution {

public:

int maxSum = INT\_MIN;

int maxGain(TreeNode\* node) {

if (!node) return 0;

// Compute max sum for left and right subtrees, ignoring negative sums

int leftGain = max(0, maxGain(node->left));

int rightGain = max(0, maxGain(node->right));

// Compute the maximum path sum passing through this node

int pathSum = node->val + leftGain + rightGain;

// Update global maximum sum

maxSum = max(maxSum, pathSum);

// Return max path sum including this node and at most one child

return node->val + max(leftGain, rightGain);

}

int maxPathSum(TreeNode\* root) {

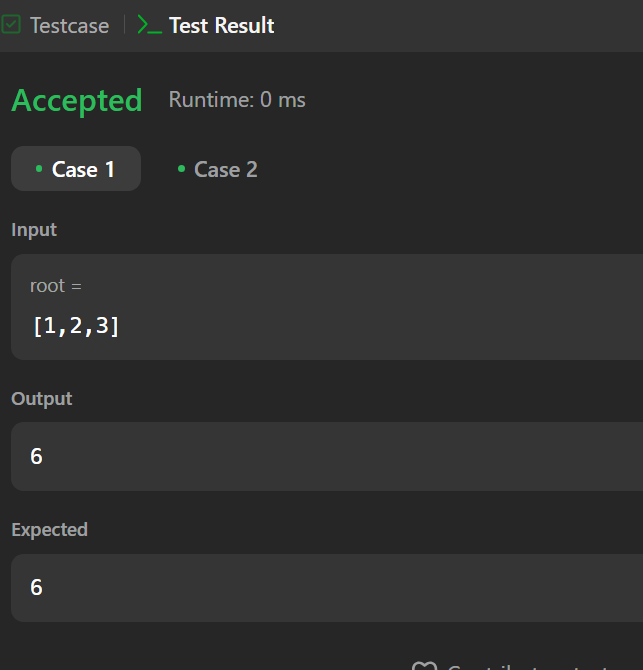
maxGain(root);

return maxSum;

}

};

**Output:**

****

**Ques 13. Vertical Order Traversal of a Binary Tree.**

**Code:**

class Solution {

public:

vector<vector<int>> verticalTraversal(TreeNode\* root) {

map<int, vector<pair<int, int>>> colMap; // {col: [(row, value)]}

queue<pair<TreeNode\*, pair<int, int>>> q; // {node, (row, col)}

q.push({root, {0, 0}}); // Start with root at (0,0)

while (!q.empty()) {

auto [node, pos] = q.front();

q.pop();

int row = pos.first, col = pos.second;

colMap[col].push\_back({row, node->val});

if (node->left) q.push({node->left, {row + 1, col - 1}});

if (node->right) q.push({node->right, {row + 1, col + 1}});

}

vector<vector<int>> result;

for (auto &[col, nodes] : colMap) {

// Sort by row first, then by value

sort(nodes.begin(), nodes.end());

vector<int> colVals;

for (auto &[row, val] : nodes) colVals.push\_back(val);

result.push\_back(colVals);

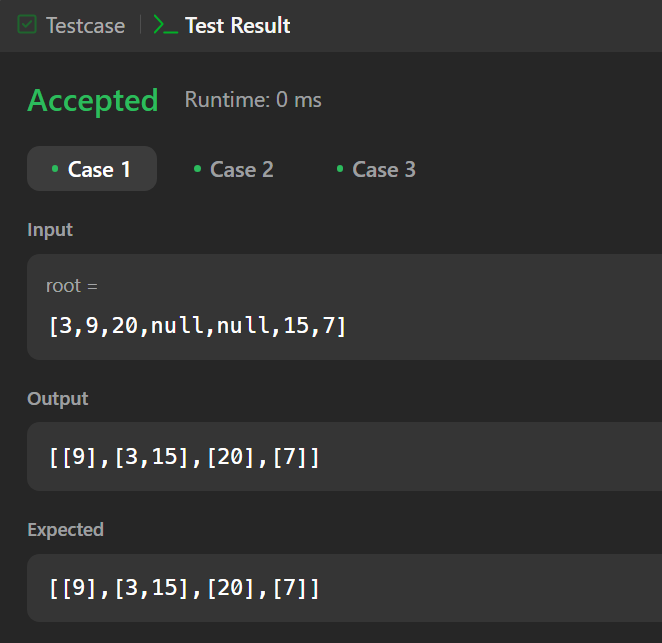
}

return result;

}

};

**Output:**

****