1.

#include <iostream>

#include <queue>

#include <vector>

#include <algorithm>

using namespace std;

// Definition of a tree node

struct TreeNode {

    int data;

    TreeNode\* left;

    TreeNode\* right;

    TreeNode(int value) {

        data = value;

        left = nullptr;

        right = nullptr;

    }

};

// Function to build a binary tree using level-order input

TreeNode\* buildBinaryTree() {

    int value;

    cout << "Enter the value for the root node: ";

    cin >> value;

    TreeNode\* root = new TreeNode(value);

    queue<TreeNode\*> q;

    q.push(root);

    while (!q.empty()) {

        TreeNode\* current = q.front();

        q.pop();

        int leftValue, rightValue;

        cout << "Enter the left child of " << current->data << " (-1 for no child): ";

        cin >> leftValue;

        if (leftValue != -1) {

            current->left = new TreeNode(leftValue);

            q.push(current->left);

        }

        cout << "Enter the right child of " << current->data << " (-1 for no child): ";

        cin >> rightValue;

        if (rightValue != -1) {

            current->right = new TreeNode(rightValue);

            q.push(current->right);

        }

    }

    return root;

}

// Function to perform an inorder traversal and store values in a vector

void inorderTraversal(TreeNode\* root, vector<int>& nodes) {

    if (root == nullptr) return;

    inorderTraversal(root->left, nodes);

    nodes.push\_back(root->data);

    inorderTraversal(root->right, nodes);

}

// Helper function to construct a balanced AVL tree from a sorted array

TreeNode\* sortedArrayToAVL(vector<int>& nodes, int start, int end) {

    if (start > end) return nullptr;

    int mid = (start + end) / 2;

    TreeNode\* root = new TreeNode(nodes[mid]);

    root->left = sortedArrayToAVL(nodes, start, mid - 1);

    root->right = sortedArrayToAVL(nodes, mid + 1, end);

    return root;

}

// Function to convert a binary tree to an AVL tree

TreeNode\* convertToAVL(TreeNode\* root) {

    vector<int> nodes;

    // Perform inorder traversal to get sorted values

    inorderTraversal(root, nodes);

    // Construct a balanced AVL tree from the sorted values

    return sortedArrayToAVL(nodes, 0, nodes.size() - 1);

}

// Function to print the tree horizontally (for visualization)

void printTreeHorizontal(TreeNode\* root, int space = 0, int levelSpace = 6) {

    if (root == nullptr) return;

    // Increase distance between levels

    space += levelSpace;

    // Print the right child first

    printTreeHorizontal(root->right, space);

    // Print the current node

    cout << endl;

    cout << string(space, ' ') << root->data;

    // Print the left child

    printTreeHorizontal(root->left, space);

}

int main() {

    // Build a binary tree

    TreeNode\* binaryTree = buildBinaryTree();

    cout << "\nOriginal Binary Tree:" << endl;

    printTreeHorizontal(binaryTree);

    // Convert to AVL tree

    TreeNode\* avlTree = convertToAVL(binaryTree);

    cout << "\nBalanced AVL Tree:" << endl;

    printTreeHorizontal(avlTree);

    return 0;

}

2.

#include <iostream>

#include <queue>

using namespace std;

// Definition of a tree node

struct TreeNode {

int data;

TreeNode\* left;

TreeNode\* right;

TreeNode(int value) {

data = value;

left = nullptr;

right = nullptr;

}

};

// Function to insert nodes level-wise based on user input

TreeNode\* buildTree() {

int value;

cout << "Enter the value for the root node: ";

cin >> value;

TreeNode\* root = new TreeNode(value);

queue<TreeNode\*> q;

q.push(root);

while (!q.empty()) {

TreeNode\* current = q.front();

q.pop();

int leftValue, rightValue;

cout << "Enter the left child of " << current->data << " (-1 for no child): ";

cin >> leftValue;

if (leftValue != -1) {

current->left = new TreeNode(leftValue);

q.push(current->left);

}

cout << "Enter the right child of " << current->data << " (-1 for no child): ";

cin >> rightValue;

if (rightValue != -1) {

current->right = new TreeNode(rightValue);

q.push(current->right);

}

}

return root;

}

// Inorder Traversal (Left, Root, Right)

void inorderTraversal(TreeNode\* root) {

if (root == nullptr) return;

inorderTraversal(root->left);

cout << root->data << " ";

inorderTraversal(root->right);

}

// Preorder Traversal (Root, Left, Right)

void preorderTraversal(TreeNode\* root) {

if (root == nullptr) return;

cout << root->data << " ";

preorderTraversal(root->left);

preorderTraversal(root->right);

}

// Postorder Traversal (Left, Right, Root)

void postorderTraversal(TreeNode\* root) {

if (root == nullptr) return;

postorderTraversal(root->left);

postorderTraversal(root->right);

cout << root->data << " ";

}

int main() {

// Build the tree

TreeNode\* root = buildTree();

// Perform and display traversals

cout << "\nInorder Traversal: ";

inorderTraversal(root);

cout << "\nPreorder Traversal: ";

preorderTraversal(root);

cout << "\nPostorder Traversal: ";

postorderTraversal(root);

return 0;

}

3.

#include <bits/stdc++.h>

using namespace std;

class Node {

public:

int data;

Node \*left;

Node \*right;

Node(int val) {

data = val;

left = nullptr;

right = nullptr;

}

};

// Returns height which is the number of edges

// along the longest path from the root node down

// to the farthest leaf node.

int height(Node \*root) {

if (root == nullptr)

return -1;

// compute the height of left and right subtrees

int lHeight = height(root->left);

int rHeight = height(root->right);

return max(lHeight, rHeight) + 1;

}

int main() {

Node \*root = new Node(1);

root->left = new Node(2);

root->right = new Node(3);

root->left->left = new Node(4);

root->left->right = new Node(5);

cout << height(root);

return 0;

}

4.

#include <bits/stdc++.h>

using namespace std;

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int value) {

data = value;

left = right = nullptr;

}

};

// Helper function to check if a tree is BST within a given range

bool isBSTUtil(Node\* node, int min, int max) {

if (node == nullptr) return true;

// If the current node's data

// is not in the valid range, return false

if (node->data < min || node->data > max) return false;

// Recursively check the left and

// right subtrees with updated ranges

return isBSTUtil(node->left, min, node->data - 1) &&

isBSTUtil(node->right, node->data + 1, max);

}

// Function to check if the entire binary tree is a BST

bool isBST(Node\* root) {

return isBSTUtil(root, INT\_MIN, INT\_MAX);

}

int main() {

Node\* root = new Node(4);

root->left = new Node(2);

root->right = new Node(5);

root->left->left = new Node(1);

root->left->right = new Node(3);

if (isBST(root)) {

cout << "True" << endl;

}

else {

cout << "False" << endl;

}

return 0;

}

5.

#include <bits/stdc++.h>

using namespace std;

class Node {

public:

int data;

Node \*left;

Node \*right;

Node \*nextRight;

Node(int val) {

data = val;

left = nullptr;

right = nullptr;

nextRight = nullptr;

}

};

// Set next right of all descendants of root.

// Assumption: root is a complete binary tree

void connectRecur(Node \*root) {

if (!root)

return;

// Set the nextRight pointer for root's left child

if (root->left)

root->left->nextRight = root->right;

// Set the nextRight pointer for root's right child

// root->nextRight will be nullptr if root is the

// rightmost child at its level

if (root->right)

root->right->nextRight =

(root->nextRight) ? root->nextRight->left : nullptr;

// Set nextRight for other nodes

// in pre-order fashion

connectRecur(root->left);

connectRecur(root->right);

}

// Sets the nextRight of root and calls connectRecur()

// for other nodes

void connect(Node \*root) {

// Set the nextRight for root

root->nextRight = nullptr;

// Set the next right for rest of the

// nodes (other than root)

connectRecur(root);

}

// Function to store the nextRight pointers in

// level-order format and return as a vector of strings

vector<string> getNextRightArray(Node \*root) {

vector<string> result;

if (!root)

return result;

queue<Node \*> q;

q.push(root);

q.push(nullptr);

while (!q.empty()) {

Node \*node = q.front();

q.pop();

if (node != nullptr) {

// Add the current node's data

result.push\_back(to\_string(node->data));

// If nextRight is nullptr, add '#'

if (node->nextRight == nullptr) {

result.push\_back("#");

}

// Push the left and right children to

// the queue (next level nodes)

if (node->left)

q.push(node->left);

if (node->right)

q.push(node->right);

}

else if (!q.empty()) {

// Add level delimiter for the next level

q.push(nullptr);

}

}

return result;

}

int main() {

Node \*root = new Node(10);

root->left = new Node(8);

root->right = new Node(2);

root->left->left = new Node(3);

connect(root);

vector<string> output = getNextRightArray(root);

for (const string &s : output) {

cout << s << ' ';

}

cout << endl;

return 0;

}