Q-1: Implement a C++ program to insert an integer into an AVL tree. Ensure that the AVL tree remains balanced after the insertion.

Sample test case:

|  |
| --- |
| Input: avltree={10,20,30}  Output:  Inserted: 10, Inorder Traversal: 10  Inserted: 20, Inorder Traversal: 10 20  Inserted: 30, Inorder Traversal: 10 20 30 |

Solution:

#include <iostream>

// Define a structure for the AVL tree node

struct Node {

int key;

int height;

Node\* left;

Node\* right;

Node(int value) : key(value), height(1), left(nullptr), right(nullptr) {}

};

// Function to get the height of a node

int getHeight(Node\* node) {

return node ? node->height : 0;

}

// Function to get the balance factor of a node

int getBalance(Node\* node) {

return node ? getHeight(node->left) - getHeight(node->right) : 0;

}

// Function to perform a right rotation

Node\* rotateRight(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

y->height = std::max(getHeight(y->left), getHeight(y->right)) + 1;

x->height = std::max(getHeight(x->left), getHeight(x->right)) + 1;

return x;

}

// Function to perform a left rotation

Node\* rotateLeft(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

x->height = std::max(getHeight(x->left), getHeight(x->right)) + 1;

y->height = std::max(getHeight(y->left), getHeight(y->right)) + 1;

return y;

}

// Function to insert a key into the AVL tree

Node\* insert(Node\* root, int key) {

if (!root)

return new Node(key);

if (key < root->key)

root->left = insert(root->left, key);

else if (key > root->key)

root->right = insert(root->right, key);

else

return root;

root->height = std::max(getHeight(root->left), getHeight(root->right)) + 1;

int balance = getBalance(root);

// Perform rotations to maintain balance

if (balance > 1 && key < root->left->key)

return rotateRight(root);

if (balance < -1 && key > root->right->key)

return rotateLeft(root);

if (balance > 1 && key > root->left->key) {

root->left = rotateLeft(root->left);

return rotateRight(root);

}

if (balance < -1 && key < root->right->key) {

root->right = rotateRight(root->right);

return rotateLeft(root);

}

return root;

}

// Function to perform an inorder traversal of the AVL tree

void inorderTraversal(Node\* root) {

if (!root)

return;

inorderTraversal(root->left);

std::cout << root->key << " ";

inorderTraversal(root->right);

}

// Main function

int main() {

Node\* root = nullptr;

// Insert elements into AVL tree

int arr[] = {10, 20, 30, 40, 50, 25};

for (int key : arr) {

root = insert(root, key);

std::cout << "Inserted: " << key << ", Inorder Traversal: ";

inorderTraversal(root);

std::cout << std::endl;

}

return 0;

}

Q-2: Given the height of an AVL tree ‘h’, the task is to find the minimum number of nodes the tree can have.

Sample test case:

|  |
| --- |
| Input : H = 3  Output : N = 7 |

Solution:

#include <bits/stdc++.h>

using namespace std;

// Function to calculate the number of nodes in an AVL tree of given height

int AVLnodes(int height)

{

// Base Conditions: An AVL tree with height 0 has 1 node, and height 1 has 2 nodes.

if (height == 0)

return 1;

else if (height == 1)

return 2;

// Recursive function call using the recurrence relation of an AVL tree

return (1 + AVLnodes(height - 1) + AVLnodes(height - 2));

}

int main()

{

int H = 3; // Desired height of the AVL tree

// Calculate and print the number of nodes in an AVL tree of height H

cout << "Number of nodes in an AVL tree of height " << H << ": " << AVLnodes(H) << endl;

return 0;

}

Q-3: Implement a program to check whether Binary search tree is balanced or not.

Sample test case:

|  |
| --- |
| Input:  10  / \  8 18  / \  5 7  /  3  Output: Tree is not balanced |

Solution:

#include <bits/stdc++.h>

using namespace std;

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int d)

{

int data = d;

left = right = NULL;

}

};

// Function to calculate the height of a tree

int height(Node\* node)

{

// base case tree is empty

if (node == NULL)

return 0;

// If tree is not empty then

// height = 1 + max of left height

// and right heights

return 1 + max(height(node->left), height(node->right));

}

// Returns true if binary tree

// with root as root is height-balanced

bool isBalanced(Node\* root)

{

// for height of left subtree

int lh;

// for height of right subtree

int rh;

// If tree is empty then return true

if (root == NULL)

return 1;

// Get the height of left and right sub trees

lh = height(root->left);

rh = height(root->right);

if (abs(lh - rh) <= 1 && isBalanced(root->left)

&& isBalanced(root->right))

return 1;

// If we reach here then tree is not height-balanced

return 0;

}

int main()

{

Node\* root = new Node(10);

root->left = new Node(8);

root->right = new Node(18);

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

root->left->right = new Node(9);

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

if (isBalanced(root))

cout << "Tree is balanced";

else

cout << "Tree is not balanced";

return 0;

}