TASK 1:

#include<iostream>

using namespace std;

class TreeNode {

public:

int value;

TreeNode\* left;

TreeNode\* right;

TreeNode() {

value = 0;

left = NULL;

right = NULL;

}

TreeNode(int val) {

value = val;

left = NULL;

right = NULL;

}

};

class AVLTree {

public:

TreeNode\* root;

AVLTree() {

root = NULL;

}

bool isTreeEmpty() {

if (root == NULL) {

return true;

}

else {

return false;

}

}

// Get Height

int height(TreeNode\* r) {

if (r == NULL)

return -1;

else {

/\* compute the height of each subtree \*/

int lheight = height(r->left);

int rheight = height(r->right);

/\* use the larger one \*/

if (lheight > rheight)

return (lheight + 1);

else return (rheight + 1);

}

}

// Get Balance factor of node N

int getBalanceFactor(TreeNode\* n) {

if (n == NULL)

return -1;

return height(n->left) - height(n->right);

}

TreeNode\* rightRotate(TreeNode\* y) {

TreeNode\* x = y->left;

TreeNode\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

return x;

}

TreeNode\* leftRotate(TreeNode\* x) {

TreeNode\* y = x->right;

TreeNode\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

return y;

}

TreeNode\* insert(TreeNode\* r, TreeNode\* new\_node) {

if (r == NULL) {

r = new\_node;

cout << "Value inserted successfully" << endl;

return r;

}

if (new\_node->value < r->value) {

r->left = insert(r->left, new\_node);

}

else if (new\_node->value > r->value) {

r->right = insert(r->right, new\_node);

}

else {

cout << "No duplicate values allowed!" << endl;

return r;

}

int bf = getBalanceFactor(r);

// Left Left Case

if (bf > 1 && new\_node->value < r->left->value)

return rightRotate(r);

// Right Right Case

if (bf < -1 && new\_node->value > r->right->value)

return leftRotate(r);

// Left Right Case

if (bf > 1 && new\_node->value > r->left->value) {

r->left = leftRotate(r->left);

return rightRotate(r);

}

// Right Left Case

if (bf < -1 && new\_node->value < r->right->value) {

r->right = rightRotate(r->right);

return leftRotate(r);

}

/\* return the (unchanged) node pointer \*/

return r;

}

TreeNode\* minValueNode(TreeNode\* node) {

TreeNode\* current = node;

/\* loop down to find the leftmost leaf \*/

while (current->left != NULL) {

current = current->left;

}

return current;

}

TreeNode\* deleteNode(TreeNode\* r, int v) {

// base case

if (r == NULL) {

return NULL;

}

// If the key to be deleted is smaller than the root's key,

// then it lies in left subtree

else if (v < r->value) {

r->left = deleteNode(r->left, v);

}

// If the key to be deleted is greater than the root's key,

// then it lies in right subtree

else if (v > r->value) {

r->right = deleteNode(r->right, v);

}

// if key is same as root's key, then This is the node to be deleted

else {

// node with only one child or no child

if (r->left == NULL) {

TreeNode\* temp = r->right;

delete r;

return temp;

}

else if (r->right == NULL) {

TreeNode\* temp = r->left;

delete r;

return temp;

}

else {

// node with two children: Get the inorder successor (smallest

// in the right subtree)

TreeNode\* temp = minValueNode(r->right);

// Copy the inorder successor's content to this node

r->value = temp->value;

// Delete the inorder successor

r->right = deleteNode(r->right, temp->value);

//deleteNode(r->right, temp->value);

}

}

int bf = getBalanceFactor(r);

// Left Left Imbalance/Case or Right rotation

if (bf == 2 && getBalanceFactor(r->left) >= 0)

return rightRotate(r);

// Left Right Imbalance/Case or LR rotation

else if (bf == 2 && getBalanceFactor(r->left) == -1) {

r->left = leftRotate(r->left);

return rightRotate(r);

}

// Right Right Imbalance/Case or Left rotation

else if (bf == -2 && getBalanceFactor(r->right) <= -0)

return leftRotate(r);

// Right Left Imbalance/Case or RL rotation

else if (bf == -2 && getBalanceFactor(r->right) == 1) {

r->right = rightRotate(r->right);

return leftRotate(r);

}

return r;

}

void printInorder(TreeNode\* r) // (Left, current node, Right)

{

if (r == NULL)

return;

/\* first recur on left child \*/

printInorder(r->left);

/\* then print the data of node \*/

cout << r->value << " ";

/\* now recur on right child \*/

printInorder(r->right);

}

TreeNode\* recursiveSearch(TreeNode\* r, int val) {

if (r == NULL || r->value == val)

return r;

else if (val < r->value)

return recursiveSearch(r->left, val);

else

return recursiveSearch(r->right, val);

}

};

int main() {

AVLTree obj;

int option = -1, val;

while (option != 0){

cout << " What operation do you want to perform? \n Select Option number.\n" << endl;

cout << "1- Insert Node" << endl;

cout << "2- Delete Node" << endl;

cout << "3- Inorder Traversal" << endl;

cout << "4- Height of Tree" << endl;

cout << "5- Balancing Factor" << endl;

cout << "0- Exit Program" << endl;

cin >> option;

TreeNode\* new\_node = new TreeNode();

switch (option) {

case 0:

break;

case 1:

cout << "AVL INSERT" << endl;

cout << "Enter VALUE of TREE NODE to INSERT in AVL Tree: ";

cin >> val;

new\_node->value = val;

obj.root = obj.insert(obj.root, new\_node);

cout << endl;

break;

case 2:

cout << "DELETE" << endl;

cout << "Enter VALUE of TREE NODE to DELETE in AVL: ";

cin >> val;

new\_node = obj.recursiveSearch(obj.root, val);

if (new\_node != NULL){

obj.root = obj.deleteNode(obj.root, val);

cout << "Value Deleted" << endl;

}

else{

cout << "Value NOT found" << endl;

}

break;

case 3:

cout << "IN-ORDER: ";

obj.printInorder(obj.root);

cout << endl;

break;

case 4:

cout << "TREE HEIGHT" << endl;

cout << "Height : " << obj.height(obj.root) << endl;

break;

case 5:

int num;

num = obj.getBalanceFactor(obj.root);

cout << endl << "Balance Factor is : " << num << endl;

break;

default:

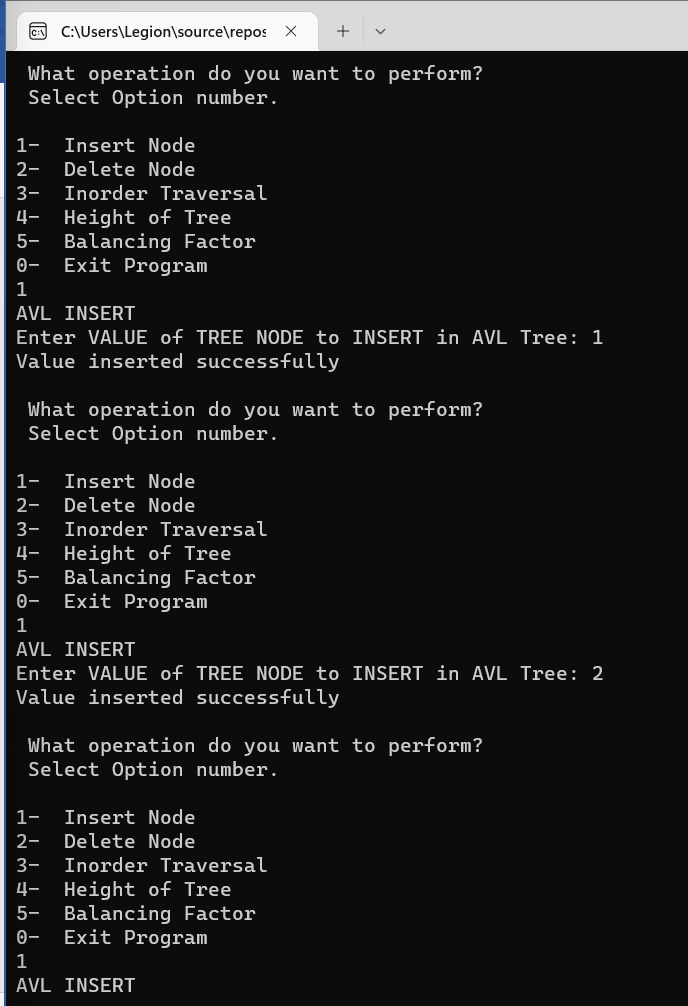
cout << "Enter Proper Option number " << endl;

}

};

return 0;

}



Text

Description automatically generated

Text

Description automatically generated