**LAB 9**

**Name:** Faizan Tariq

**Roll:** 22F-3858

TASK 1 :  
#include <iostream>

using namespace std;

struct AvlNode {

int data;

AvlNode\* left;

AvlNode\* right;

int height;

AvlNode(int value) : data(value), left(NULL), right(NULL), height(0) {}

};

class AvlTree {

private:

AvlNode\* root;

public:

AvlTree() : root(NULL) {}

int getHeight(AvlNode\* node) {

if (node == nullptr) {

return -1;

}

return node->height;

}

int getBalanceFactor(AvlNode\* node) {

if (node == nullptr) {

return 0;

}

return getHeight(node->left) - getHeight(node->right);

}

AvlNode\* rotateRight(AvlNode\* y) { // RR rotation

AvlNode\* x = y->left;

AvlNode\* T2 = x->right;

x->right = y;

y->left = T2;

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

x->height = max(getHeight(x->left), y->height) + 1;

return x; /\* New root \*/

}

AvlNode\* rotateLeft(AvlNode\* x) {

AvlNode\* y = x->right;

AvlNode\* T2 = y->left;

y->left = x;

x->right = T2;

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

y->height = max(getHeight(y->left), x->height) + 1;

return y; /\* New root \*/

}

AvlNode\* doubleRotateWithRight(AvlNode\* K1) {

/\* RR rotation between K3 and K2 \*/

K1->right = rotateLeft(K1->right);

/\* LL rotation between K1 and K2 \*/

return rotateRight(K1);

}

AvlNode\* doubleRotateWithLeft(AvlNode\* K3) {

/\* LL rotation between K1 and K2 \*/

K3->left = rotateRight(K3->left);

/\* RR rotation between K3 and K2 \*/

return rotateLeft(K3);

}

AvlNode\* insert(int x, AvlNode\* T) {

if (T == nullptr) {

/\* Create and return a new node \*/

return new AvlNode(x);

}

if (x < T->data) {

T->left = insert(x, T->left);

}

else if (x > T->data) {

T->right = insert(x, T->right);

} /\* Else X is in the tree already; we'll do nothing \*/

T->height = 1 + max(getHeight(T->left), getHeight(T->right));

int balance = getBalanceFactor(T);

if (balance > 1) {

if (x < T->left->data) {

return rotateRight(T); // Right-Right rotation

}

else {

return doubleRotateWithLeft(T); // Right-Left rotation

}

}

if (balance < -1) {

if (x > T->right->data) {

return rotateLeft(T); // Left-Left rotation

}

else {

return doubleRotateWithRight(T); // Left-Right rotation

}

}

return T;

}

bool find(int value, AvlNode\* startNode) {

while (startNode) {

if (value < startNode->data) {

startNode = startNode->left;

}

else if (value > startNode->data) {

startNode = startNode->right;

}

else {

cout << "VALUE FOUND.\n";

return true;

}

}

cout << "Value not found.\n";

return false;

}

AvlNode\* deleteNode(int num, AvlNode\* nodePtr) {

if (nodePtr == nullptr) {

cout << "Num not found!\n";

return nullptr;

}

if (num < nodePtr->data) {

nodePtr->left = deleteNode(num, nodePtr->left);

}

else if (num > nodePtr->data) {

nodePtr->right = deleteNode(num, nodePtr->right);

}

else {

AvlNode\* tempNode = nodePtr;

if (nodePtr->left == nullptr) {

nodePtr = nodePtr->right;

}

else if (nodePtr->right == nullptr) {

nodePtr = nodePtr->left;

}

else {

tempNode = nodePtr->right;

while (tempNode->left) {

tempNode = tempNode->left;

}

tempNode->left = nodePtr->left;

nodePtr = nodePtr->right;

}

delete tempNode;

}

if (nodePtr != nullptr) {

nodePtr->height = 1 + max(getHeight(nodePtr->left), getHeight(nodePtr->right)); // calculates height of inserted node

int balance = getBalanceFactor(nodePtr);

if (balance > 1) {

if (getBalanceFactor(nodePtr->left) >= 0) {

return rotateRight(nodePtr); // Right-Right rotation

}

else {

return doubleRotateWithLeft(nodePtr); // Right-Left rotation

}

}

if (balance < -1) {

if (getBalanceFactor(nodePtr->right) <= 0) {

return rotateLeft(nodePtr); // Left-Left rotation

}

else {

return doubleRotateWithRight(nodePtr); // Left-Right rotation

}

}

}

return nodePtr;

}

void inorderDisplay(AvlNode\* p) const {

if (p != nullptr) {

inorderDisplay(p->left);

cout << p->data << " ";

inorderDisplay(p->right);

}

}

void insert(int x) {

root = insert(x, root);

}

void remove(int num) {

root = deleteNode(num, root);

}

bool find(int value) {

return find(value, root);

}

void displayInOrder() {

inorderDisplay(root);

cout << endl;

}

};

int main() {

AvlTree avl;

avl.insert(10);

avl.insert(4);

avl.insert(13);

avl.insert(3);

avl.insert(8);

avl.insert(11);

avl.insert(20);

cout << "In-order traversal: ";

avl.displayInOrder();

avl.remove(8);

cout << "In-order traversal after removing 8: ";

avl.displayInOrder();

cout << "Searching for 12: ";

avl.find(13);

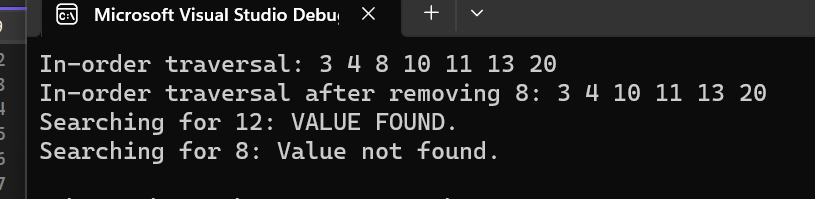
cout << "Searching for 8: ";

avl.find(8);

return 0;

}

OUTPUT :



TASK 2:

#include <iostream>

using namespace std;

struct AvlNode {

int data;

AvlNode\* left;

AvlNode\* right;

int height;

AvlNode(int value) : data(value), left(NULL), right(NULL), height(0) {}

};

class AvlTree {

private:

AvlNode\* root;

public:

AvlTree() : root(NULL) {}

int calcuateheight(AvlNode\* temp)

{

int h = 0;

if (temp != NULL)

{

int l\_height = calcuateheight(temp->left);

int r\_height = calcuateheight(temp->right);

int max\_height = max(l\_height, r\_height);

h = max\_height + 1;

}

return h;

}

int getHeight(AvlNode\* node) {

if (node == nullptr) {

return -1;

}

return node->height;

}

int getBalanceFactor(AvlNode\* node) {

if (node == nullptr) {

return 0;

}

return getHeight(node->left) - getHeight(node->right);

}

AvlNode\* rotateRight(AvlNode\* y) { // RR rotation

AvlNode\* x = y->left;

AvlNode\* T2 = x->right;

x->right = y;

y->left = T2;

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

x->height = max(getHeight(x->left), y->height) + 1;

return x; /\* New root \*/

}

AvlNode\* rotateLeft(AvlNode\* x) {

AvlNode\* y = x->right;

AvlNode\* T2 = y->left;

y->left = x;

x->right = T2;

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

y->height = max(getHeight(y->left), x->height) + 1;

return y; /\* New root \*/

}

AvlNode\* doubleRotateWithRight(AvlNode\* K1) {

/\* RR rotation between K3 and K2 \*/

K1->right = rotateLeft(K1->right);

/\* LL rotation between K1 and K2 \*/

return rotateRight(K1);

}

AvlNode\* doubleRotateWithLeft(AvlNode\* K3) {

/\* LL rotation between K1 and K2 \*/

K3->left = rotateRight(K3->left);

/\* RR rotation between K3 and K2 \*/

return rotateLeft(K3);

}

void insertNode(int num) {

if (!root) root = new AvlNode(num);

else {

AvlNode\* curNode = root;

while (1) {

if (num < curNode->data) {

if (curNode->left)

curNode = curNode->left;

else {

curNode->left = new AvlNode(num);

curNode->left->height = calcuateheight(curNode->left);

return;

}

}

else if (num > curNode->data) {

if (curNode->right) {

curNode = curNode->right;

}

else {

curNode->right = new AvlNode(num);

curNode->right->height = calcuateheight(curNode->right);

return;

}

}

else {

cout << "\nDUPLICATE NUMBER FOUND IN TREE.\n";

}

}

}

}

void inorderDisplay(AvlNode\* p) const {

if (p != nullptr) {

inorderDisplay(p->left);

cout << p->data << " ";

inorderDisplay(p->right);

}

}

void displayInOrder() {

inorderDisplay(root);

cout << endl;

}

bool isAVL(AvlNode\* Root, bool& avl) {

if (Root == nullptr) {

avl = true;

return true;

}

int balanceFactor = getBalanceFactor(Root);

if (balanceFactor < -1 || balanceFactor > 1) {

avl = false;

return false;

}

return isAVL(Root->left , avl) && isAVL(Root->right, avl);

}

AvlNode\* getRoot() {

return root;

}

AvlNode\* insert(int x, AvlNode\* T) {

if (T == nullptr) {

/\* Create and return a new node \*/

return new AvlNode(x);

}

if (x < T->data) {

T->left = insert(x, T->left);

}

else if (x > T->data) {

T->right = insert(x, T->right);

} /\* Else X is in the tree already; we'll do nothing \*/

T->height = 1 + max(getHeight(T->left), getHeight(T->right));

int balance = getBalanceFactor(T);

if (balance > 1) {

if (x < T->left->data) {

return rotateRight(T); // Right-Right rotation

}

else {

return doubleRotateWithLeft(T); // Right-Left rotation

}

}

if (balance < -1) {

if (x > T->right->data) {

return rotateLeft(T); // Left-Left rotation

}

else {

return doubleRotateWithRight(T); // Left-Right rotation

}

}

return T;

}

};

int main() {

AvlTree avl , avl2;

avl.insertNode(1);

avl.insertNode(2);

avl.insertNode(3);

avl.insertNode(4);

avl.insertNode(5);

avl.insertNode(6);

bool is\_AVL = false;

avl.isAVL(avl.getRoot(), is\_AVL);

if (is\_AVL) {

cout << "Your Tree is AVL.\n";

}

else {

cout << "Your tree is not avl.\n";

}

cout << "\nInserting AVL TREE :\n";

avl2.insert(10, avl2.getRoot());

avl2.insert(4, avl2.getRoot());

avl2.insert(13, avl2.getRoot());

avl2.insert(3, avl2.getRoot());

avl2.insert(8, avl2.getRoot());

avl2.insert(20, avl2.getRoot());

avl2.insert(11, avl2.getRoot());

is\_AVL = false;

avl2.isAVL(avl2.getRoot(), is\_AVL);

if (is\_AVL) {

cout << "Your Tree is AVL.\n";

}

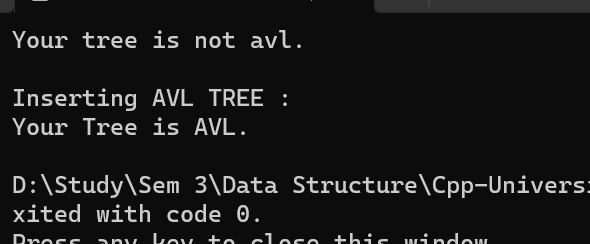
else {

cout << "Your tree is not avl.\n";

}

return 0;

}

OUTPUT:  


TASK 3:  
#include <iostream>

using namespace std;

struct AvlNode {

int data;

AvlNode\* left;

AvlNode\* right;

int height;

AvlNode(int value) : data(value), left(NULL), right(NULL), height(0) {}

};

class AvlTree {

private:

AvlNode\* root;

public:

AvlTree() : root(NULL) {}

int calcuateheight(AvlNode\* temp)

{

int h = 0;

if (temp != NULL)

{

int l\_height = calcuateheight(temp->left);

int r\_height = calcuateheight(temp->right);

int max\_height = max(l\_height, r\_height);

h = max\_height + 1;

}

return h;

}

int getHeight(AvlNode\* node) {

if (node == NULL) {

return -1;

}

return node->height;

}

int getBalanceFactor(AvlNode\* node) {

if (node == NULL) {

return -1;

}

return getHeight(node->left) - getHeight(node->right);

}

AvlNode\* rotateRight(AvlNode\* y) { // RR rotation

AvlNode\* x = y->left;

AvlNode\* T2 = x->right;

x->right = y;

y->left = T2;

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

x->height = max(getHeight(x->left), y->height) + 1;

return x; /\* New root \*/

}

AvlNode\* rotateLeft(AvlNode\* x) {

AvlNode\* y = x->right;

AvlNode\* T2 = y->left;

y->left = x;

x->right = T2;

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

y->height = max(getHeight(y->left), x->height) + 1;

return y; /\* New root \*/

}

AvlNode\* doubleRotateWithRight(AvlNode\* K1) {

/\* RR rotation between K3 and K2 \*/

K1->right = rotateLeft(K1->right);

/\* LL rotation between K1 and K2 \*/

return rotateRight(K1);

}

AvlNode\* doubleRotateWithLeft(AvlNode\* K3) {

/\* LL rotation between K1 and K2 \*/

K3->left = rotateRight(K3->left);

/\* RR rotation between K3 and K2 \*/

return rotateLeft(K3);

}

void insertNode(int num) {

if (!root) root = new AvlNode(num);

else {

AvlNode\* curNode = root;

while (1) {

if (num < curNode->data) {

if (curNode->left)

curNode = curNode->left;

else {

curNode->left = new AvlNode(num);

curNode->left->height = calcuateheight(curNode->left);

return;

}

}

else if (num > curNode->data) {

if (curNode->right) {

curNode = curNode->right;

}

else {

curNode->right = new AvlNode(num);

curNode->right->height = calcuateheight(curNode->right);

return;

}

}

else {

cout << "\nDUPLICATE NUMBER FOUND IN TREE.\n";

}

}

}

}

AvlNode\* convertBSTtoAVL(AvlNode\* p) {

if (p != NULL) {

p->left = convertBSTtoAVL(p->left);

p->right = convertBSTtoAVL(p->right);

int balance = getBalanceFactor(p);

cout << "\nData :" << p->data << " Balancing factor :" << balance <<endl;

if (balance > 1) {

if (p->data < p->left->data) {

p = rotateRight(p);

}

else {

p = doubleRotateWithLeft(p);

}

}

if (balance < -1) {

if (p->data > p->right->data) {

p = rotateLeft(p);

}

else {

p = rotateLeft(p);

}

}

}

return p;

}

void inorderDisplay(AvlNode\* p) const {

if (p != NULL) {

inorderDisplay(p->left);

cout << p->data << " ";

inorderDisplay(p->right);

}

}

void displayInOrder() {

inorderDisplay(root);

cout << endl;

}

bool isAVL(AvlNode\* Root, bool& avl) {

if (Root == NULL) {

avl = true;

return true;

}

int balanceFactor = getBalanceFactor(Root);

if (balanceFactor < -1 || balanceFactor > 1) {

avl = false;

return false;

}

return isAVL(Root->left , avl) && isAVL(Root->right, avl);

}

AvlNode\* getRoot() {

return root;

}

AvlNode\* insert(int x, AvlNode\* T) {

if (T == NULL) {

return new AvlNode(x);

}

if (x < T->data) {

T->left = insert(x, T->left);

}

else if (x > T->data) {

T->right = insert(x, T->right);

}

T->height = 1 + max(getHeight(T->left), getHeight(T->right));

int balance = getBalanceFactor(T);

if (balance > 1) {

if (x < T->left->data) {

return rotateRight(T); // Right-Right rotation

}

else {

return doubleRotateWithLeft(T); // Right-Left rotation

}

}

if (balance < -1) {

if (x > T->right->data) {

return rotateLeft(T); // Left-Left rotation

}

else {

return doubleRotateWithRight(T); // Left-Right rotation

}

}

return T;

}

void setRoot(AvlNode\* newRoot) {

root = newRoot;

}

};

int main() {

AvlTree avl;

avl.insertNode(1);

avl.insertNode(2);

avl.insertNode(3);

avl.insertNode(4);

avl.insertNode(5);

avl.insertNode(6);

bool is\_AVL = false;

avl.isAVL(avl.getRoot(), is\_AVL);

if (is\_AVL) {

cout << "Your Tree is AVL.\n";

}

else {

cout << "Your tree is not avl.\n";

}

cout << "Converting BST to AVL.\n";

avl.setRoot(avl.convertBSTtoAVL(avl.getRoot()));

is\_AVL = false;

avl.isAVL(avl.getRoot(), is\_AVL);

if (is\_AVL) {

cout << "Your Tree is AVL.\n";

}

else {

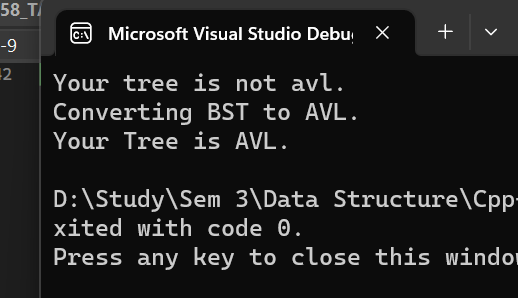
cout << "Your tree is not avl.\n";

}

return 0;

}

OUTPUT:



TASK 4:

#include <iostream>

using namespace std;

struct AvlNode {

int data;

AvlNode\* left;

AvlNode\* right;

int height;

AvlNode(int value) : data(value), left(NULL), right(NULL), height(0) {}

};

class AvlTree {

private:

AvlNode\* root;

public:

AvlTree() : root(NULL) {}

int getHeight(AvlNode\* node) {

if (node == nullptr) {

return -1;

}

return node->height;

}

int getBalanceFactor(AvlNode\* node) {

if (node == nullptr) {

return 0;

}

return getHeight(node->left) - getHeight(node->right);

}

AvlNode\* rotateRight(AvlNode\* y) { // RR rotation

AvlNode\* x = y->left;

AvlNode\* T2 = x->right;

x->right = y;

y->left = T2;

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

x->height = max(getHeight(x->left), y->height) + 1;

return x; /\* New root \*/

}

AvlNode\* rotateLeft(AvlNode\* x) {

AvlNode\* y = x->right;

AvlNode\* T2 = y->left;

y->left = x;

x->right = T2;

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

y->height = max(getHeight(y->left), x->height) + 1;

return y; /\* New root \*/

}

AvlNode\* doubleRotateWithRight(AvlNode\* K1) {

/\* RR rotation between K3 and K2 \*/

K1->right = rotateLeft(K1->right);

/\* LL rotation between K1 and K2 \*/

return rotateRight(K1);

}

AvlNode\* doubleRotateWithLeft(AvlNode\* K3) {

/\* LL rotation between K1 and K2 \*/

K3->left = rotateRight(K3->left);

/\* RR rotation between K3 and K2 \*/

return rotateLeft(K3);

}

AvlNode\* insert(int x, AvlNode\* T) {

if (T == nullptr) {

/\* Create and return a new node \*/

return new AvlNode(x);

}

if (x < T->data) {

T->left = insert(x, T->left);

}

else if (x > T->data) {

T->right = insert(x, T->right);

} /\* Else X is in the tree already; we'll do nothing \*/

if (x == T->data) {

cout << "Already Exists.\n";

return T;

}

T->height = 1 + max(getHeight(T->left), getHeight(T->right));

int balance = getBalanceFactor(T);

if (balance > 1) {

if (x < T->left->data) {

return rotateRight(T); // Right-Right rotation

}

else {

return doubleRotateWithLeft(T); // Right-Left rotation

}

}

if (balance < -1) {

if (x > T->right->data) {

return rotateLeft(T); // Left-Left rotation

}

else {

return doubleRotateWithRight(T); // Left-Right rotation

}

}

return T;

}

bool find(int value, AvlNode\* startNode) {

while (startNode) {

if (value < startNode->data) {

startNode = startNode->left;

}

else if (value > startNode->data) {

startNode = startNode->right;

}

else {

cout << "found";

return true;

}

}

return false;

}

AvlNode\* deleteNode(int num, AvlNode\* nodePtr) {

if (nodePtr == nullptr) {

cout << "Num not found!\n";

return nullptr;

}

if (num < nodePtr->data) {

nodePtr->left = deleteNode(num, nodePtr->left);

}

else if (num > nodePtr->data) {

nodePtr->right = deleteNode(num, nodePtr->right);

}

else {

AvlNode\* tempNode = nodePtr;

if (nodePtr->left == nullptr) {

nodePtr = nodePtr->right;

}

else if (nodePtr->right == nullptr) {

nodePtr = nodePtr->left;

}

else {

tempNode = nodePtr->right;

while (tempNode->left) {

tempNode = tempNode->left;

}

tempNode->left = nodePtr->left;

nodePtr = nodePtr->right;

}

delete tempNode;

}

if (nodePtr != nullptr) {

nodePtr->height = 1 + max(getHeight(nodePtr->left), getHeight(nodePtr->right)); // calculates height of inserted node

int balance = getBalanceFactor(nodePtr);

if (balance > 1) {

if (getBalanceFactor(nodePtr->left) >= 0) {

return rotateRight(nodePtr); // Right-Right rotation

}

else {

return doubleRotateWithLeft(nodePtr); // Right-Left rotation

}

}

if (balance < -1) {

if (getBalanceFactor(nodePtr->right) <= 0) {

return rotateLeft(nodePtr); // Left-Left rotation

}

else {

return doubleRotateWithRight(nodePtr); // Left-Right rotation

}

}

}

return nodePtr;

}

void inorderDisplay(AvlNode\* p) const {

if (p != nullptr) {

inorderDisplay(p->left);

cout << p->data << " ";

inorderDisplay(p->right);

}

}

void insert(int x) {

root = insert(x, root);

}

void remove(int num) {

root = deleteNode(num, root);

}

void displayInOrder() {

inorderDisplay(root);

cout << endl;

}

AvlNode\* getRoot() {

return root;

}

int calcuateheight(AvlNode\* temp)

{

int h = 0;

if (temp != NULL)

{

int l\_height = calcuateheight(temp->left);

int r\_height = calcuateheight(temp->right);

int max\_height = max(l\_height, r\_height);

h = max\_height + 1;

}

return h;

}

void insertNode(int num) {

if (!root) root = new AvlNode(num);

else {

AvlNode\* curNode = root;

while (1) {

if (num < curNode->data) {

if (curNode->left)

curNode = curNode->left;

else {

curNode->left = new AvlNode(num);

curNode->left->height = calcuateheight(curNode->left);

return;

}

}

else if (num > curNode->data) {

if (curNode->right) {

curNode = curNode->right;

}

else {

curNode->right = new AvlNode(num);

curNode->right->height = calcuateheight(curNode->right);

return;

}

}

else {

cout << "\nDUPLICATE NUMBER FOUND IN TREE.\n";

}

}

}

}

void joinTrees(AvlNode\* avl1, AvlNode\* avl2) {

if (avl1 != nullptr) {

inorderDisplay(avl1->left);

insertNode(avl1->data);

inorderDisplay(avl1->right);

}

if (avl2 != nullptr) {

inorderDisplay(avl2->left);

insertNode(avl2->data);

inorderDisplay(avl2->right);

}

}

};

int main() {

AvlTree avl1 , avl2 , result;

avl1.insert(10);

avl1.insert(4);

avl1.insert(13);

avl1.insert(3);

avl2.insert(8);

avl2.insert(11);

avl2.insert(20);

cout << "\nIn-order traversal of avl1 : \n";

avl1.displayInOrder();

cout << "\nIn-order traversal of avl2 :\n";

avl2.displayInOrder();

cout << "Resultant Tree :" << endl;

result.joinTrees(avl1.getRoot(), avl2.getRoot());

result.displayInOrder();

return 0;

}

OUTPUT:

