# LAB # 09

## PROBLEM # 1

Provide a C++ implementation of AVL tree must include

1. Insert()

2. Delete()

### Code:

/\*The following program is the implementation of AVL trees insertion and deletion operations. Apart from that the program is

capable of displaying the AVL tree on the screen inverted 90 degrees.\*/

#include <iostream>

using namespace std;

const int SPACE = 5;

class TreeNode {

public:

int value;

TreeNode\* left;

TreeNode\* right;

TreeNode() {

value = 0;

left = NULL;

right = NULL;

}

TreeNode(int v) {

value = v;

left = NULL;

right = NULL;

}

};

class AVL

{

private:

TreeNode\* minValueNode(TreeNode\* node) {

TreeNode\* current = node;

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

while (current->left != NULL) {

current = current->left;

}

return current;

}

public:

//For inserting nodes in the AVL Tree

TreeNode\* insert(TreeNode\* root, int value)

{

if (root == NULL)

{

root = new TreeNode;

root->value = value;

root->left = NULL;

root->right = NULL;

return root;

}

else if (value < root->value)

{

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

root = balance(root);

}

else if (value >= root->value)

{

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

root = balance(root);

}

return root;

}

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);

}

}

//After the deletion of node same as Binary search tree, the line below performs the required rotations to

// ensure the BST is Balanced

r = balance(r);

return r;

}

//Calculates the height of the node

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);

if (lheight > rheight)

return (lheight + 1);

else return (rheight + 1);

}

}

//diff = Left height - Right height

int diff(TreeNode \* temp)

{

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

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

int b\_factor = l\_height - r\_height;

return b\_factor;

}

//This function determines the balance factor so later

//on we can perform the required rotations to balance the tree

TreeNode\* balance(TreeNode \* temp)

{

int bal\_factor = diff(temp);

if (bal\_factor > 1)

{

if (diff(temp->left) > 0)

temp = ll\_rotation(temp);

else

temp = lr\_rotation(temp);

}

else if (bal\_factor < -1)

{

if (diff(temp->right) > 0)

temp = rl\_rotation(temp);

else

temp = rr\_rotation(temp);

}

return temp;

}

//Right Right Rotation

TreeNode\* rr\_rotation(TreeNode \* node)

{

TreeNode\* temp;

temp = node->right;

node->right = temp->left;

temp->left = node;

return temp;

}

//Left Left Rotation

TreeNode\* ll\_rotation(TreeNode \* node)

{

TreeNode\* temp;

temp = node->left;

node->left = temp->right;

temp->right = node;

return temp;

}

//Left Right Rotation

TreeNode\* lr\_rotation(TreeNode \* node)

{

TreeNode\* temp;

temp = node->left;

node->left = rr\_rotation(temp);

return ll\_rotation(node);

}

//Right Left Rotation

TreeNode\* rl\_rotation(TreeNode \* node)

{

TreeNode\* temp;

temp = node->right;

node->right = ll\_rotation(temp);

return rr\_rotation(node);

}

/\*The function displays the AVL tree on the console inverted 90 degrees\*/

void displayAVL(TreeNode\* r, int space) {

if (r == NULL)

return;

space += SPACE;

displayAVL(r->right, space);

cout << endl;

for (int i = SPACE; i < space; i++)

cout << " ";

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

displayAVL(r->left, space);

};

};

int main()

{

int choice, n;

AVL T1;

TreeNode\* root = NULL;

root = T1.insert(root, 10);

root = T1.insert(root, 20);

root = T1.insert(root, 30);

root = T1.insert(root, 40);

root = T1.insert(root, 50);

while (1)

{

cout << "\n\t|||| AVL Tree Implementation ||||\t" << endl << endl;

cout << "1.Insert Element into the tree" << endl;

cout << "2.Delete Element from the tree" << endl;

cout << "3.Display Balanced AVL Tree" << endl;

cout << "4.Exit" << endl;

cout << "Enter your Choice: ";

cout << endl;

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter value to be inserted: ";

cin >> n;

root = T1.insert(root, n);

break;

case 2:

cout << "Enter value to be deleted: " << endl;

cin >> n;

root = T1.deleteNode(root, n);

break;

case 3:

if (root == NULL)

{

cout << "Tree is Empty" << endl;

continue;

}

cout << "Balanced AVL Tree:" << endl;

T1.displayAVL(root, 1);

break;

case 4:

exit(1);

break;

default:

cout << "Wrong Choice" << endl;

}

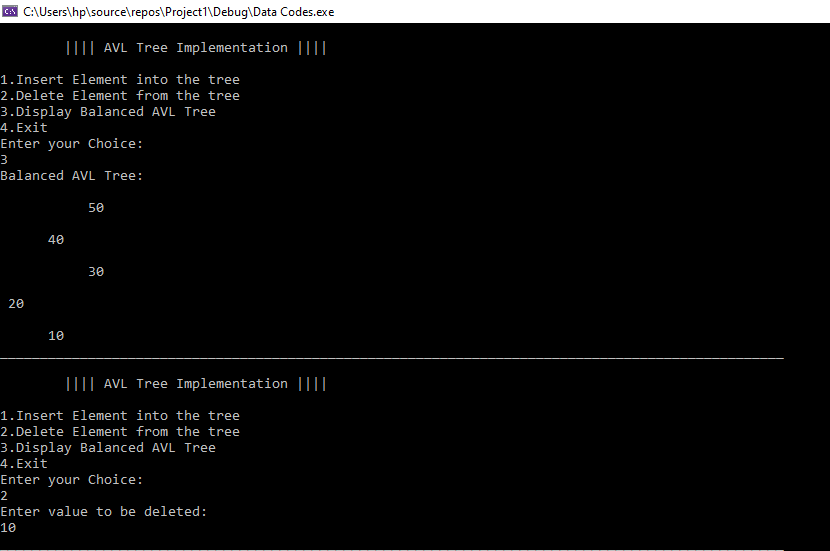
cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";

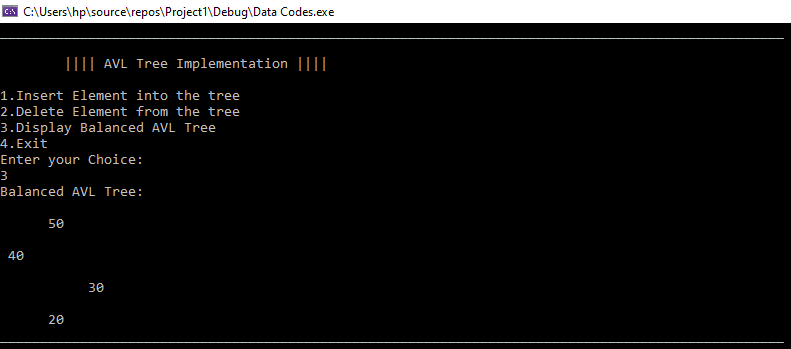
}

return 0;

}

### Screenshot:





## PROBLEM # 2

Write a recursive function to check if the provided tree is a valid AVL tree or not.

### Code:

/\*The following program is the implementation of AVL trees insertion, deletion operations also it checks if the current tree

is AVL or not. In addition to this the program is also capable of displaying the tree.\*/

#include <iostream>

using namespace std;

const int SPACE = 5;

class TreeNode {

public:

int value;

TreeNode\* left;

TreeNode\* right;

TreeNode() {

value = 0;

left = NULL;

right = NULL;

}

TreeNode(int v) {

value = v;

left = NULL;

right = NULL;

}

};

class AVL

{

private:

int h;

bool ifAVL;

TreeNode\* minValueNode(TreeNode\* node) {

TreeNode\* current = node;

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

while (current->left != NULL) {

current = current->left;

}

return current;

}

public:

AVL()

{

h = 0;

ifAVL = true;

}

//For inserting nodes in the AVL Tree

TreeNode\* insert(TreeNode\* root, int value)

{

if (root == NULL)

{

root = new TreeNode;

root->value = value;

root->left = NULL;

root->right = NULL;

return root;

}

else if (value < root->value)

{

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

root = balance(root);

}

else if (value >= root->value)

{

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

root = balance(root);

}

return root;

}

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);

}

}

//After the deletion of node same as Binary search tree, the line below performs the required rotations to

// ensure the BST is Balanced

r = balance(r);

return r;

}

//Calculates the height of the node

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);

if (lheight > rheight)

return (lheight + 1);

else return (rheight + 1);

}

}

//diff = Left height - Right height

int diff(TreeNode \* temp)

{

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

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

int b\_factor = l\_height - r\_height;

return b\_factor;

}

//The following function performs preorder traversal on the tree and checks the balancing factor for each node

//If it exceeds 1 or -1 the program it displays not AVL if not it displays AVL with the help of AvlOrNot()

void chkAVL(TreeNode\* r)

{

if (r == NULL)

return;

h = diff(r);

if (h < -1 || h > 1)

{

ifAVL = false;

return;

}

chkAVL(r->left);

chkAVL(r->right);

}

//It only displays the result and calls chkAVL

void AvlOrNot(TreeNode\* r)

{

chkAVL(r);

if (ifAVL == true)

cout << "\nThe following BST is AVL" << endl;

else

cout << "The followning tree is not AVL" << endl;

}

//This function determines the balance factor so later

//on we can perform the required rotations to balance the tree

TreeNode\* balance(TreeNode \* temp)

{

int bal\_factor = diff(temp);

if (bal\_factor > 1)

{

if (diff(temp->left) > 0)

temp = ll\_rotation(temp);

else

temp = lr\_rotation(temp);

}

else if (bal\_factor < -1)

{

if (diff(temp->right) > 0)

temp = rl\_rotation(temp);

else

temp = rr\_rotation(temp);

}

return temp;

}

//Right Right Rotation

TreeNode\* rr\_rotation(TreeNode \* node)

{

TreeNode\* temp;

temp = node->right;

node->right = temp->left;

temp->left = node;

return temp;

}

//Left Left Rotation

TreeNode\* ll\_rotation(TreeNode \* node)

{

TreeNode\* temp;

temp = node->left;

node->left = temp->right;

temp->right = node;

return temp;

}

//Left Right Rotation

TreeNode\* lr\_rotation(TreeNode \* node)

{

TreeNode\* temp;

temp = node->left;

node->left = rr\_rotation(temp);

return ll\_rotation(node);

}

//Right Left Rotation

TreeNode\* rl\_rotation(TreeNode \* node)

{

TreeNode\* temp;

temp = node->right;

node->right = ll\_rotation(temp);

return rr\_rotation(node);

}

/\*The function displays the AVL tree on the console inverted 90 degrees\*/

void displayAVL(TreeNode\* r, int space) {

if (r == NULL)

return;

space += SPACE;

displayAVL(r->right, space);

cout << endl;

for (int i = SPACE; i < space; i++)

cout << " ";

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

displayAVL(r->left, space);

};

};

int main()

{

int choice, n;

AVL T1;

TreeNode\* root = NULL;

root = T1.insert(root, 10);

root = T1.insert(root, 20);

root = T1.insert(root, 30);

root = T1.insert(root, 40);

root = T1.insert(root, 50);

while (1)

{

cout << "\n\t|||| AVL Tree Implementation ||||\t" << endl << endl;

cout << "1. Insert Element into the tree" << endl;

cout << "2. Delete Element from the tree" << endl;

cout << "3. CHECK IF AVL" << endl;

cout << "4. Display Balanced AVL Tree" << endl;

cout << "5. Exit" << endl;

cout << "Enter your Choice: ";

cout << endl;

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter value to be inserted: ";

cin >> n;

root = T1.insert(root, n);

break;

case 2:

cout << "Enter value to be deleted: " << endl;

cin >> n;

root = T1.deleteNode(root, n);

break;

case 3:

T1.AvlOrNot(root);

break;

case 4:

if (root == NULL)

{

cout << "Tree is Empty" << endl;

continue;

}

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

T1.displayAVL(root, 1);

break;

case 5:

exit(1);

break;

default:

cout << "Wrong Choice" << endl;

}

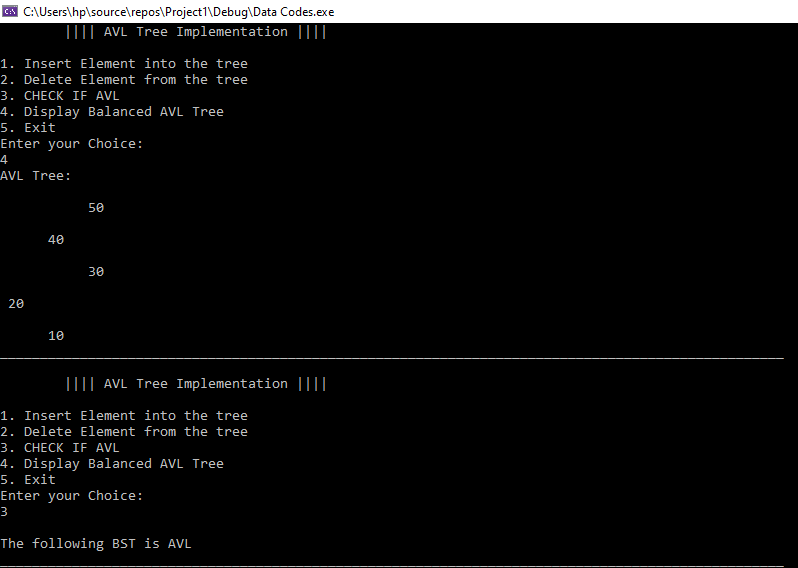
cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";

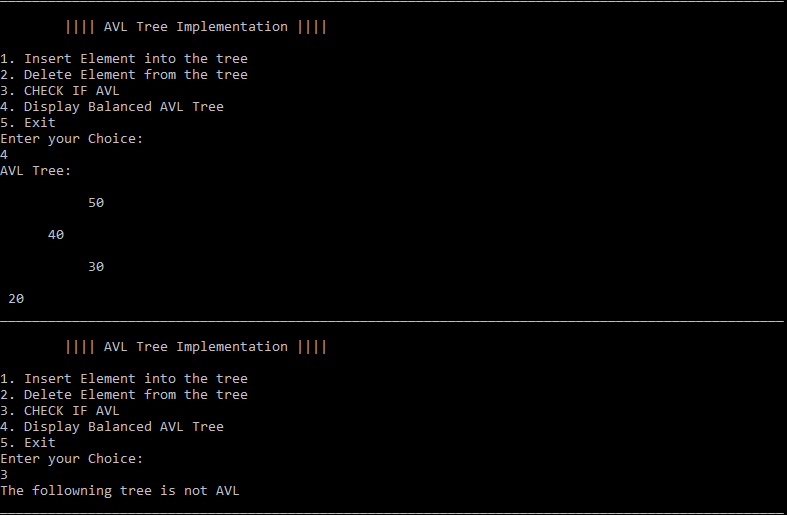
}

return 0;

}

### Screenshot:





## PROBLEM # 3

Write a C++ program to convert a provide BST into a AVL tree.

### Code:

/\*The following program simply inserts, deletes nodes from the the tree, later with a help of CONVERT\_AVL function

the BST is converted into AVL tree. In addition to this the program is also capable of displaying the tree.\*/

#include <iostream>

using namespace std;

const int SPACE = 5;

class TreeNode {

public:

int value;

TreeNode\* left;

TreeNode\* right;

TreeNode() {

value = 0;

left = NULL;

right = NULL;

}

TreeNode(int v) {

value = v;

left = NULL;

right = NULL;

}

};

class AVL

{

private:

int h;

bool ifAVL;

TreeNode\* minValueNode(TreeNode\* node) {

TreeNode\* current = node;

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

while (current->left != NULL) {

current = current->left;

}

return current;

}

public:

int numOfNodes;

AVL()

{

h = 0;

ifAVL = true;

numOfNodes = 0;

}

//For inserting nodes in the AVL Tree

TreeNode\* insert(TreeNode\* root, int value)

{

if (root == NULL)

{

root = new TreeNode;

root->value = value;

root->left = NULL;

root->right = NULL;

return root;

}

else if (value < root->value)

{

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

//root = balance(root);

}

else if (value >= root->value)

{

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

//root = balance(root);

}

numOfNodes++;

return root;

}

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

// base case

if (r == NULL) {

return NULL;

}

else if (v < r->value) {

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

}

else if (v > r->value) {

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

}

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);

}

}

//After the deletion of node same as Binary search tree, the line below performs the required rotations to

// ensure the BST is Balanced

//r = balance(r);

numOfNodes--;

return r;

}

TreeNode\* CONVERT\_AVL(TreeNode\* r)

{

return r = balance(r);

}

//Calculates the height of the node

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);

if (lheight > rheight)

return (lheight + 1);

else return (rheight + 1);

}

}

//diff = Left height - Right height

int diff(TreeNode \* temp)

{

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

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

int b\_factor = l\_height - r\_height;

return b\_factor;

}

//The following function performs preorder traversal on the tree and checks the balancing factor for each node

//If it exceeds 1 or -1 the program it displays not AVL if not it displays AVL with the help of AvlOrNot()

void chkAVL(TreeNode\* r)

{

if (r == NULL)

return;

h = diff(r);

if (h > -1 || h > 1)

{

ifAVL = false;

return;

}

chkAVL(r->left);

chkAVL(r->right);

}

//It only displays the result and calls chkAVL

bool AvlOrNot(TreeNode\* r)

{

chkAVL(r);

return ifAVL;

}

//This function determines the balance factor so later

//on we can perform the required rotations to balance the tree

TreeNode\* balance(TreeNode \* temp)

{

int bal\_factor = diff(temp);

if (bal\_factor > 1)

{

if (diff(temp->left) > 0)

temp = ll\_rotation(temp);

else

temp = lr\_rotation(temp);

}

else if (bal\_factor < -1)

{

if (diff(temp->right) > 0)

temp = rl\_rotation(temp);

else

temp = rr\_rotation(temp);

}

return temp;

}

//Right Right Rotation

TreeNode\* rr\_rotation(TreeNode \* node)

{

TreeNode\* temp;

temp = node->right;

node->right = temp->left;

temp->left = node;

return temp;

}

//Left Left Rotation

TreeNode\* ll\_rotation(TreeNode \* node)

{

TreeNode\* temp;

temp = node->left;

node->left = temp->right;

temp->right = node;

return temp;

}

//Left Right Rotation

TreeNode\* lr\_rotation(TreeNode \* node)

{

TreeNode\* temp;

temp = node->left;

node->left = rr\_rotation(temp);

return ll\_rotation(node);

}

//Right Left Rotation

TreeNode\* rl\_rotation(TreeNode \* node)

{

TreeNode\* temp;

temp = node->right;

node->right = ll\_rotation(temp);

return rr\_rotation(node);

}

/\*The function displays the AVL tree on the console inverted 90 degrees\*/

void displayAVL(TreeNode\* r, int space) {

if (r == NULL)

return;

space += SPACE;

displayAVL(r->right, space);

cout << endl;

for (int i = SPACE; i < space; i++)

cout << " ";

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

displayAVL(r->left, space);

};

};

int main()

{

int choice, n;

AVL T1;

TreeNode\* root = NULL;

root = T1.insert(root, 10);

root = T1.insert(root, 20);

root = T1.insert(root, 30);

root = T1.insert(root, 40);

root = T1.insert(root, 50);

while (1)

{

cout << "\n\t|||| BST Tree ||||\t" << endl << endl;

cout << "1. Insert Element into the tree" << endl;

cout << "2. Delete Element from the tree" << endl;

cout << "3. CONVERT TO AVL" << endl;

cout << "4. Display TREE" << endl;

cout << "5. Exit" << endl;

cout << "Enter your Choice: ";

cout << endl;

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter value to be inserted: ";

cin >> n;

root = T1.insert(root, n);

break;

case 2:

cout << "Enter value to be deleted: " << endl;

cin >> n;

root = T1.deleteNode(root, n);

break;

case 3:

cout << "\n Tree is Successfully converted into AVL!" << endl;

for(int i = 0; i < T1.numOfNodes; i++)

root = T1.CONVERT\_AVL(root);

break;

case 4:

if (root == NULL)

{

cout << "Tree is Empty" << endl;

continue;

}

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

T1.displayAVL(root, 1);

break;

case 5:

exit(1);

break;

default:

cout << "Wrong Choice" << endl;

}

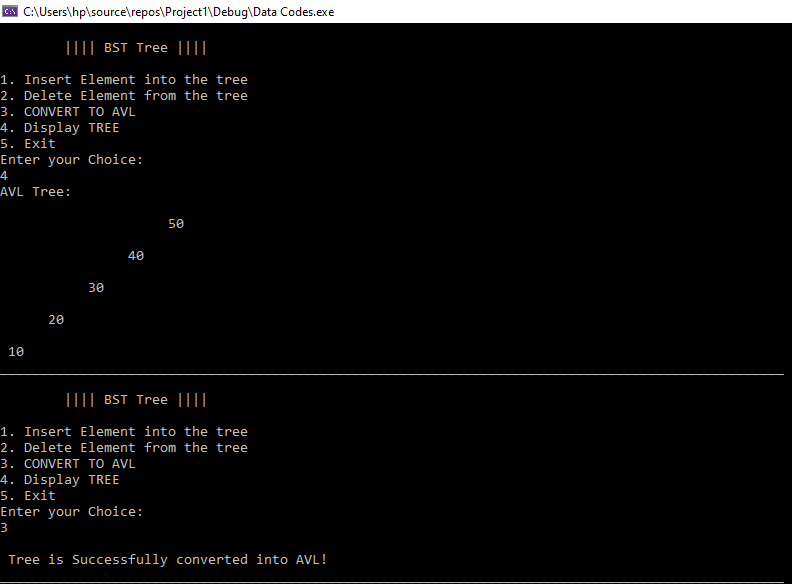
cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";

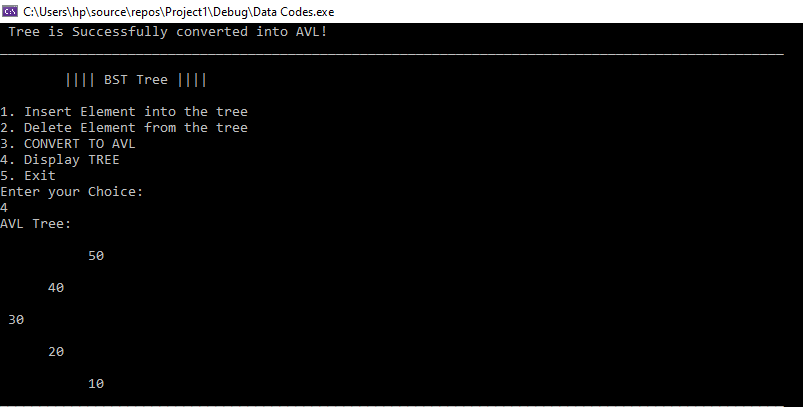
}

return 0;

}

### Screenshot:





## PROBLEM # 4

Write a C++ function which is passed two AVL trees T1 and T2, where the largest key

in T1 is less than the smallest key in T2, Join(T1, T2) returns an AVL tree containing the

union of the elements in T1 and T2.

### Code:

/\*The following program takes two AVL trees and merges them into a single tree and

displays them on the console in a preorder manner

\*/

#include <iostream>

using namespace std;

const int SPACE = 5;

class TreeNode {

public:

int value;

TreeNode\* left;

TreeNode\* right;

TreeNode() {

value = 0;

left = NULL;

right = NULL;

}

TreeNode(int v) {

value = v;

left = NULL;

right = NULL;

}

};

class AVL

{

private:

int h;

TreeNode\* minValueNode(TreeNode\* node) {

TreeNode\* current = node;

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

while (current->left != NULL) {

current = current->left;

}

return current;

}

public:

int numOfNodes;

AVL()

{

h = 0;

numOfNodes = 0;

}

//For inserting nodes in the AVL Tree

TreeNode\* insert(TreeNode\* root, int value)

{

if (root == NULL)

{

root = new TreeNode;

root->value = value;

root->left = NULL;

root->right = NULL;

return root;

}

else if (value < root->value)

{

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

root = balance(root);

}

else if (value >= root->value)

{

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

root = balance(root);

}

numOfNodes++;

return root;

}

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);

}

}

//After the deletion of node same as Binary search tree, the line below performs the required rotations to

// ensure the BST is Balanced

r = balance(r);

numOfNodes--;

return r;

}

//Calculates the height of the node

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);

if (lheight > rheight)

return (lheight + 1);

else return (rheight + 1);

}

}

//diff = Left height - Right height

int diff(TreeNode\* temp)

{

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

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

int b\_factor = l\_height - r\_height;

return b\_factor;

}

//This function determines the balance factor so later

//on we can perform the required rotations to balance the tree

TreeNode\* balance(TreeNode\* temp)

{

int bal\_factor = diff(temp);

if (bal\_factor > 1)

{

if (diff(temp->left) > 0)

temp = ll\_rotation(temp);

else

temp = lr\_rotation(temp);

}

else if (bal\_factor < -1)

{

if (diff(temp->right) > 0)

temp = rl\_rotation(temp);

else

temp = rr\_rotation(temp);

}

return temp;

}

//Right Right Rotation

TreeNode\* rr\_rotation(TreeNode\* node)

{

TreeNode\* temp;

temp = node->right;

node->right = temp->left;

temp->left = node;

return temp;

}

//Left Left Rotation

TreeNode\* ll\_rotation(TreeNode\* node)

{

TreeNode\* temp;

temp = node->left;

node->left = temp->right;

temp->right = node;

return temp;

}

//Left Right Rotation

TreeNode\* lr\_rotation(TreeNode\* node)

{

TreeNode\* temp;

temp = node->left;

node->left = rr\_rotation(temp);

return ll\_rotation(node);

}

//Right Left Rotation

TreeNode\* rl\_rotation(TreeNode\* node)

{

TreeNode\* temp;

temp = node->right;

node->right = ll\_rotation(temp);

return rr\_rotation(node);

}

/\*The function displays the AVL tree on the console inverted 90 degrees\*/

void displayAVL(TreeNode\* r, int space) {

if (r == NULL)

return;

space += SPACE;

displayAVL(r->right, space);

cout << endl;

for (int i = SPACE; i < space; i++)

cout << " ";

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

displayAVL(r->left, space);

};

};

TreeNode\* MergeTrees(TreeNode\* t1, TreeNode\* t2)

{

if (!t1)

return t2;

if (!t2)

return t1;

t1->value = t1->value + t2->value;

t1->left = MergeTrees(t1->left, t2->left);

t1->right = MergeTrees(t1->right, t2->right);

return t1;

}

int main()

{

int tNodes = 0;

AVL T1, T2, T3;

TreeNode\* r1 = NULL;

TreeNode\* r2 = NULL;

r1 = T1.insert(r1, 50);

r1 = T1.insert(r1, 40);

r1 = T1.insert(r1, 60);

r2 = T2.insert(r2, 80);

r2 = T2.insert(r2, 70);

r2 = T2.insert(r2, 90);

tNodes = T1.numOfNodes + T2.numOfNodes;

cout << "Tree 1: " << endl;

T1.displayAVL(r1,5);

cout << "\n\n\n";

cout << "Tree 2: " << endl;

T2.displayAVL(r2, 5);

cout << "\n\n\n";

TreeNode\* r3 = MergeTrees(r1, r2);

cout << "MERGED TREE: " << endl;

T3.displayAVL(r3, 5);

cout << endl << endl;

system("pause");

return 0;

}

### Screenshot:

