**WEEK – 4**

**4. Write a program to perform insertion and deletion operations on AVL trees.**

**CODE:**

#include <iostream>

#include <algorithm>

using namespace std;

// Node structure for AVL tree

struct Node {

int key;

Node\* left;

Node\* right;

int height;

};

// Function to get height of a node

int height(Node\* node) {

if (node == nullptr)

return 0;

return node->height;

}

// Function to get the balance factor of a node

int getBalance(Node\* node) {

if (node == nullptr)

return 0;

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

}

// Function to create a new node

Node\* newNode(int key) {

Node\* node = new Node();

node->key = key;

node->left = nullptr;

node->right = nullptr;

node->height = 1;

return node;

}

// Function to right rotate subtree rooted with y

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

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

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

// Return new root

return x;

}

// Function to left rotate subtree rooted with x

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

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

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

// Return new root

return y;

}

// Function to insert a node into AVL tree

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

if (node == nullptr)

return newNode(key);

if (key < node->key)

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

else if (key > node->key)

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

else // Equal keys not allowed in AVL

return node;

// Update height of this ancestor node

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

// Get the balance factor to check if this node became unbalanced

int balance = getBalance(node);

// If the node becomes unbalanced, there are 4 cases

// Left Left Case

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

return rightRotate(node);

// Right Right Case

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

return leftRotate(node);

// Left Right Case

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

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

return rightRotate(node);

}

// Right Left Case

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

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

return leftRotate(node);

}

// If the node is balanced, return the unchanged node pointer

return node;

}

// Function to find the node with minimum key value in a subtree

Node\* minValueNode(Node\* node) {

Node\* current = node;

while (current->left != nullptr)

current = current->left;

return current;

}

// Function to delete a node with given key from AVL tree

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

// Perform standard BST delete

if (root == nullptr)

return root;

if (key < root->key)

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

else if (key > root->key)

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

else {

// Node to be deleted found

// Node with only one child or no child

if ((root->left == nullptr) || (root->right == nullptr)) {

Node\* temp = root->left ? root->left : root->right;

// No child case

if (temp == nullptr) {

temp = root;

root = nullptr;

} else // One child case

\*root = \*temp; // Copy the contents of the non-empty child

delete temp;

} else {

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

// in the right subtree)

Node\* temp = minValueNode(root->right);

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

root->key = temp->key;

// Delete the inorder successor

root->right = deleteNode(root->right, temp->key);

}

}

// If the tree had only one node then return

if (root == nullptr)

return root;

// Update height of the current node

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

// Get the balance factor to check if this node became unbalanced

int balance = getBalance(root);

// If this node becomes unbalanced, there are 4 cases

// Left Left Case

if (balance > 1 && getBalance(root->left) >= 0)

return rightRotate(root);

// Left Right Case

if (balance > 1 && getBalance(root->left) < 0) {

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

return rightRotate(root);

}

// Right Right Case

if (balance < -1 && getBalance(root->right) <= 0)

return leftRotate(root);

// Right Left Case

if (balance < -1 && getBalance(root->right) > 0) {

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

return leftRotate(root);

}

return root;

}

// Function to print inorder traversal of AVL tree

void inorder(Node\* root) {

if (root != nullptr) {

inorder(root->left);

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

inorder(root->right);

}

}

// Function to display menu and handle user choice

void menu(Node\* &root) {

int choice;

do {

cout << "\nAVL Tree Menu:\n";

cout << "1. Insert\n";

cout << "2. Delete\n";

cout << "3. Display\n";

cout << "4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch(choice) {

case 1: {

int key;

cout << "Enter key to insert: ";

cin >> key;

root = insert(root, key);

break;

}

case 2: {

int key;

cout << "Enter key to delete: ";

cin >> key;

root = deleteNode(root, key);

break;

}

case 3: {

cout << "Inorder traversal of AVL tree: ";

inorder(root);

cout << endl;

break;

}

case 4: {

cout << "Exiting program...\n";

exit(0);

}

default:

cout << "Invalid choice. Please try again.\n";

}

} while(choice != 4);

}

// Driver program

int main() {

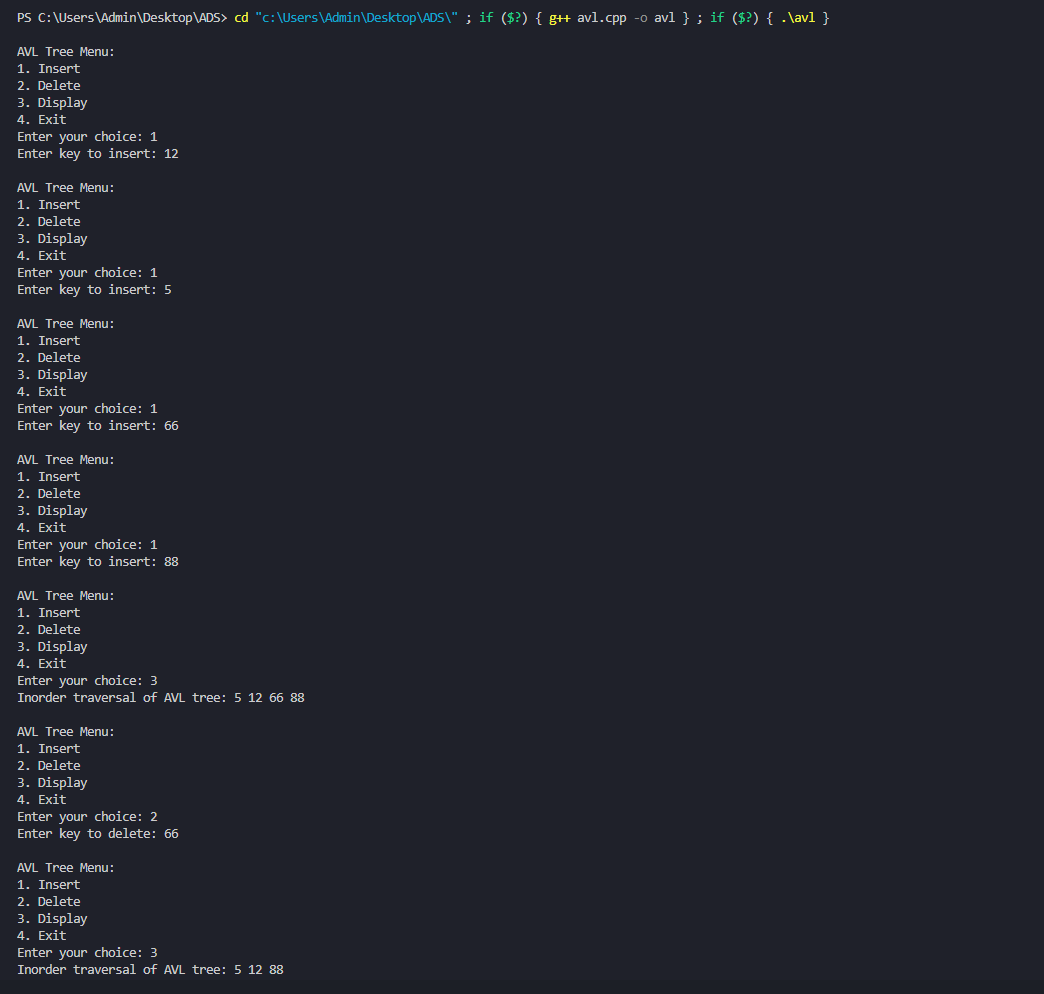
Node\* root = nullptr;

menu(root);

return 0;

}

**OUTPUT:**

****