**Dsa Project : Avl Tree operations menue ( maker )**

**Grp members:**

* **Hamdah Iqbal L1F22BSDS0018**
* **M Arif Rubbanni L1f22BSDS0010**
* **M Moiz Shabbir L1F22BSDS0023**
* **Syed Rizam Ali Rizvi L1F22BSDS0007**

**Code:**

#include <iostream>

using namespace std;

struct Node {

int key;

Node\* left;

Node\* right;

int height;

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

};

int getHeight(Node\* N);

void updateHeight(Node\* N);

int getBalance(Node\* N);

Node\* rotateRight(Node\* root);

Node\* rotateLeft(Node\* root);

Node\* minValueNode(Node\* node);

Node\* maxValueNode(Node\* node);

Node\* balance(Node\* root);

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

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

Node\* search(Node\* root, int key);

void inorder(Node\* root);

void preorder(Node\* root);

void postorder(Node\* root);

int height(Node\* root);

int diameter(Node\* root);

Node\* successor(Node\* root, int key);

Node\* predecessor(Node\* root, int key);

void printMenu();

int getHeight(Node\* N) {

if (!N) return 0;

return N->height;

}

void updateHeight(Node\* N) {

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

}

int getBalance(Node\* N) {

if (!N) return 0;

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

}

Node\* rotateRight(Node\* root) {

Node\* newRoot = root->left;

Node\* rightChild = newRoot->right;

newRoot->right = root;

root->left = rightChild;

updateHeight(root);

updateHeight(newRoot);

return newRoot;

}

Node\* rotateLeft(Node\* root) {

Node\* newRoot = root->right;

Node\* leftChild = newRoot->left;

newRoot->left = root;

root->right = leftChild;

updateHeight(root);

updateHeight(newRoot);

return newRoot;

}

Node\* minValueNode(Node\* node) {

Node\* current = node;

while (current && current->left)

current = current->left;

return current;

}

Node\* maxValueNode(Node\* node) {

Node\* current = node;

while (current && current->right)

current = current->right;

return current;

}

Node\* balance(Node\* root) {

updateHeight(root);

int balanceFactor = getBalance(root);

if (balanceFactor > 1) {

cout << "Node " << root->key << " is unbalanced: \n";

if (getBalance(root->left) >= 0) {

cout << "Balance = " << balanceFactor << " (LL case)\n";

return rotateRight(root);

}

else {

cout << "Balance = " << balanceFactor << " (LR case)\n";

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

return rotateRight(root);

}

}

if (balanceFactor < -1) {

cout << "Node " << root->key << " is unbalanced: \n";

if (getBalance(root->right) <= 0) {

cout << "Balance = " << balanceFactor << " (RR case)\n";

return rotateLeft(root);

}

else {

cout << "Balance = " << balanceFactor << " (RL case)\n";

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

return rotateLeft(root);

}

}

return root;

}

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;

return balance(root);

}

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

if (!root) return root;

if (key < root->key)

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

else if (key > root->key)

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

else {

if (!root->left) {

Node\* temp = root->right;

delete root;

return temp;

}

else if (!root->right) {

Node\* temp = root->left;

delete root;

return temp;

}

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

root->key = temp->key;

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

}

if (!root) return root;

return balance(root);

}

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

if (!root || root->key == key) return root;

if (key > root->key) return search(root->right, key);

return search(root->left, key);

}

void inorder(Node\* root) {

if (root) {

inorder(root->left);

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

inorder(root->right);

}

}

void preorder(Node\* root) {

if (root) {

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

preorder(root->left);

preorder(root->right);

}

}

void postorder(Node\* root) {

if (root) {

postorder(root->left);

postorder(root->right);

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

}

}

int height(Node\* root) {

if (!root) return 0;

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

}

int diameter(Node\* root) {

if (!root) return 0;

int lh = height(root->left);

int rh = height(root->right);

int ld = diameter(root->left);

int rd = diameter(root->right);

return max(lh + rh + 1, max(ld, rd));

}

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

Node\* current = search(root, key);

if (!current) return nullptr;

if (current->right) return minValueNode(current->right);

Node\* successor = nullptr;

Node\* ancestor = root;

while (ancestor != current) {

if (current->key < ancestor->key) {

successor = ancestor;

ancestor = ancestor->left;

}

else

ancestor = ancestor->right;

}

return successor;

}

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

Node\* current = search(root, key);

if (!current) return nullptr;

if (current->left) return maxValueNode(current->left);

Node\* predecessor = nullptr;

Node\* ancestor = root;

while (ancestor != current) {

if (current->key > ancestor->key) {

predecessor = ancestor;

ancestor = ancestor->right;

}

else

ancestor = ancestor->left;

}

return predecessor;

}

void printMenu() {

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

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

cout << "1. Insert a key\n";

cout << "2. Delete a key\n";

cout << "3. Search for a key\n";

cout << "4. Find successor of a key\n";

cout << "5. Find predecessor of a key\n";

cout << "6. Find minimum value node\n";

cout << "7. Find maximum value node\n";

cout << "8. Inorder traversal\n";

cout << "9. Preorder traversal\n";

cout << "10. Postorder traversal\n";

cout << "11. Height of the tree\n";

cout << "12. Diameter of the tree\n";

cout << "0. Exit\n";

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

cout << "Enter your choice: ";

}

int main() {

Node\* root = nullptr;

int choice, key;

do {

printMenu();

cin >> choice;

switch (choice) {

case 1:

while (true) {

cout << "Enter keys to insert (enter -1 to stop): ";

cin >> key;

if (key == -1) break;

root = insert(root, key);

cout << "AVL Tree after insertion (inorder): ";

inorder(root);

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

}

break;

case 2:

while (true) {

cout << "Enter key to delete: (enter -1 to stop): ";

cin >> key;

if (key == -1) break;

root = deleteNode(root, key);

cout << "AVL Tree after deletion (inorder): ";

inorder(root);

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

}

break;

case 3:

while (true) {

cout << "Enter key to search: (enter -1 to stop): ";

cin >> key;

if (key == -1) break;

if (search(root, key))

cout << "Key " << key << " found in the AVL tree\n\n------------------------------------------------------------------------\n\n";

else

cout << "Key " << key << " not found in the AVL tree\n\n------------------------------------------------------------------------\n\n";

}

break;

case 4:

while (true) {

cout << "Enter key to find successor: (enter -1 to stop): ";

cin >> key;

if (key == -1) break;

{

Node\* succ = successor(root, key);

if (succ)

cout << "Successor of " << key << " is " << succ->key << "\n\n------------------------------------------------------------------------\n\n";

else

cout << "No successor found.\n\n\n------------------------------------------------------------------------\n\n";

}

}

break;

case 5:

while (true) {

cout << "Enter key to find predecessor: (enter -1 to stop): ";

cin >> key;

if (key == -1) break;

{

Node\* pred = predecessor(root, key);

if (pred)

cout << "Predecessor of " << key << " is " << pred->key << ".\n\n\n------------------------------------------------------------------------\n\n";

else

cout << "No predecessor found.\n\n\n------------------------------------------------------------------------\n\n";

}

}

break;

case 6:

{

Node\* minNode = minValueNode(root);

if (minNode)

cout << "Minimum value node in AVL tree: " << minNode->key << "\n\n------------------------------------------------------------------------\n\n";

else

cout << "AVL tree is empty.\n\n\n------------------------------------------------------------------------\n\n";

}

break;

case 7:

{

Node\* maxNode = maxValueNode(root);

if (maxNode)

cout << "Maximum value node in AVL tree: " << maxNode->key <<"\n\n------------------------------------------------------------------------\n\n";

else

cout << "AVL tree is empty.\n\n\n------------------------------------------------------------------------\n\n";

}

break;

case 8:

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

inorder(root);

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

break;

case 9:

cout << "Preorder traversal of AVL tree:\n";

preorder(root);

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

break;

case 10:

cout << "Postorder traversal of AVL tree:\n";

postorder(root);

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

break;

case 11:

cout << "Height of the AVL tree: " << height(root) <<"\n\n------------------------------------------------------------------------\n\n";

break;

case 12:

cout << "Diameter of the AVL tree: " << diameter(root) <<"\n\n------------------------------------------------------------------------\n\n";

break;

case 0:

cout << "Exiting...Goodbye!\n\n\n------------------------------------------------------------------------\n\n";

break;

default:

cout << "Invalid choice...Please enter again.\n\n\n------------------------------------------------------------------------\n\n";

}

} while (choice != 0);

return 0;

}