#include <iostream>

#include <string>

using namespace std;

struct Node {

string keyword, meaning;

Node \*left, \*right;

int height;

Node(string key, string mean) {

keyword = key;

meaning = mean;

left = right = nullptr;

height = 1;

}

};

int max(int a, int b) {

return (a > b) ? a : b;

}

class AVLTree {

private:

Node\* root;

int height(Node\* n) {

return n ? n->height : 0;

}

int getBalance(Node\* n) {

return n ? height(n->left) - height(n->right) : 0;

}

void updateHeight(Node\* n) {

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

}

Node\* rotateRight(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

updateHeight(y);

updateHeight(x);

return x;

}

Node\* rotateLeft(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

updateHeight(x);

updateHeight(y);

return y;

}

Node\* insert(Node\* node, string key, string meaning) {

if (!node)

return new Node(key, meaning);

if (key < node->keyword)

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

else if (key > node->keyword)

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

else {

cout << "Keyword already exists. Updating meaning.\n";

node->meaning = meaning;

return node;

}

updateHeight(node);

int balance = getBalance(node);

// Balancing

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

return rotateRight(node);

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

return rotateLeft(node);

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

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

return rotateRight(node);

}

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

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

return rotateLeft(node);

}

return node;

}

Node\* minValueNode(Node\* node) {

Node\* current = node;

while (current->left != nullptr)

current = current->left;

return current;

}

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

if (!root) return root;

if (key < root->keyword)

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

else if (key > root->keyword)

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

else {

if (!root->left || !root->right) {

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

if (!temp) {

temp = root;

root = nullptr;

} else {

\*root = \*temp;

}

delete temp;

} else {

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

root->keyword = temp->keyword;

root->meaning = temp->meaning;

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

}

}

if (!root) return root;

updateHeight(root);

int balance = getBalance(root);

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

return rotateRight(root);

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

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

return rotateRight(root);

}

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

return rotateLeft(root);

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

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

return rotateLeft(root);

}

return root;

}

void inorder(Node\* root) {

if (root) {

inorder(root->left);

cout << root->keyword << " : " << root->meaning << endl;

inorder(root->right);

}

}

void reverseInorder(Node\* root) {

if (root) {

reverseInorder(root->right);

cout << root->keyword << " : " << root->meaning << endl;

reverseInorder(root->left);

}

}

Node\* search(Node\* root, string key, int& comparisons) {

comparisons++;

if (!root) return nullptr;

if (key == root->keyword) return root;

else if (key < root->keyword) return search(root->left, key, comparisons);

else return search(root->right, key, comparisons);

}

public:

AVLTree() {

root = nullptr;

}

void insert(string key, string meaning) {

root = insert(root, key, meaning);

}

void deleteKeyword(string key){

root = deleteNode(root, key);

}

void updateMeaning(string key, string newMeaning) {

int comparisons = 0;

Node\* found = search(root, key, comparisons);

if (found) {

found->meaning = newMeaning;

cout << "Meaning updated successfully.\n";

} else {

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

}

}

void displayAscending(){

cout << "\nDictionary in Ascending Order:\n";

inorder(root);

}

void displayDescending(){

cout << "\nDictionary in Descending Order:\n";

reverseInorder(root);

}

void searchKeyword(string key){

int comparisons = 0;

Node\* result = search(root, key, comparisons);

if (result)

cout << "Found: " << result->keyword << " -> " << result->meaning << " in " << comparisons << " comparisons.\n";

else

cout << "Keyword not found. Comparisons made: " << comparisons << "\n";

}

};

int main() {

AVLTree dict;

int choice;

string key, meaning;

do {

cout << "\n=== Dictionary Menu ===\n";

cout << "1. Insert keyword\n";

cout << "2. Delete keyword\n";

cout << "3. Update meaning\n";

cout << "4. Display Ascending\n";

cout << "5. Display Descending\n";

cout << "6. Search keyword\n";

cout << "7. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

cin.ignore();

switch (choice) {

case 1:

cout << "Enter keyword: ";

getline(cin, key);

cout << "Enter meaning: ";

getline(cin, meaning);

dict.insert(key, meaning);

break;

case 2:

cout << "Enter keyword to delete: ";

getline(cin, key);

dict.deleteKeyword(key);

break;

case 3:

cout << "Enter keyword to update: ";

getline(cin, key);

cout << "Enter new meaning: ";

getline(cin, meaning);

dict.updateMeaning(key, meaning);

break;

case 4:

dict.displayAscending();

break;

case 5:

dict.displayDescending();

break;

case 6:

cout << "Enter keyword to search: ";

getline(cin, key);

dict.searchKeyword(key);

break;

case 7:

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

break;

default:

cout << "Invalid choice.\n";

}

} while (choice != 7);

return 0;

}